University of Zurich[UZH]

Master Project – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of a Collaborative, Lightweight Malware Analysis Sandbox using Container Virtualization

Raffael Mogicato
Zurich, Switzerland
Student ID: 18-742-767

Adrian Zermin
Zurich, Switzerland
ID: 20-731-956

Supervisor: Jan von der Assen, Muriel Franco
Date of Submission: February 06, 2023

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Project
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

Malware ist weiterhin eine relevante Bedrohung für Cyber-physische Systeme. Deshalb bleiben die Entwicklung neuer Ansätze zu Analyse von Malware sowie das Erstellen relevanter und nützlicher Datensätze von grosser Bedeutung. Im Kontext der dynamischen Analyse von Malware sind gut gepflegte, open-source Lösungen rar. Deshalb erforscht diese Arbeit die Anwendung von Container-basierten Ansätzen im Kontext der dynamischen Malware Analyse als Alternative zu weitverbreiteten, Hypervisor-basierten Lösungen. Hierzu wird *SecBox*, eine kollaborative, containerbasierte, leichtgewichtige Sandbox Plattform zur dynamischen Analyse von Malware vorgestellt sowie evaluiert. Die Plattform bietet Zugang zu einer Auswahl von Malware-Proben, ermöglicht eine echtzeitige Interaktion mit der Sandbox und liefert interaktive Visualisierungsmöglichkeiten. Die Lösung verfügt über einen mehrstufigen Analyseprozess, der es Benutzern ermöglicht, bösartiges Verhalten mithilfe einer Referenzinstanz zu erkennen. Zusätzlich werden weiterführende Analysen durch den Export von Systemaufruf- und Netzwerkpaketdaten unterstützt. Die Ergebnisse der durchgeführten Evaluation bestätigen, dass die vorgeschlagene Plattform ein nützliches Werkzeug für Analysten und Sicherheitsforscher ist, um Erkenntnisse und Daten zu spezifischen Malware-Proben zu gewinnen.

Abstract

Malware as an attack vector for cyber-physical systems has remained a relevant security threat, thus the development of novel analysis approaches and the generation of relevant and useful execution data remains critical. In the context of dynamic analysis of malware, well-maintained, open-source solutions are scarce. This work explores the applications of containerization in the context of dynamic malware analysis as an alternative to traditional approaches leveraging hypervisor-based virtualization. To do this, the development of a prototypical implementation of *SecBox* as well as the acquisition of the theoretical background and related work required for its implementation is described. *SecBox*, a collaborative, container-based, lightweight dynamic malware analysis sandbox platform is subsequently presented and evaluated. The platform provides access to a selection of malware samples, allows real-time interaction with the sandbox, and provides interactive visualization capabilities. It features a multi-step analysis process that allows users to discern malicious behavior using a baseline instance as a reference. Additionally, downstream analysis tasks are supported through the export of system call and network packet data. The results of the conducted evaluation confirm the proposed platform is a useful tool for analysts and security researchers to generate insights and data regarding specific malware samples.

Acknowledgments

We would like to first and foremost thank our supervisor, Jan von der Assen, for his continued extensive guidance, mentorship, and advice during the process of this work. He continued to go the extra mile to provide our work with the exposure it deserves and submit it for publication.

This work would not have been possible without Prof Dr Burkhard Stiller providing us with the invaluable opportunity to perform research at the Communication Systems Group, as well as the necessary resources to make our vision a reality.

On a more personal note, we would like to thank Janik Lüchinger for reviewing this work and providing his unique insights into the topic of cyber security.

Contents

Chapter 1

Introduction

Cybercrime has been increasing at an alarming rate. In 2015, cyber crimes resulted in 3 trillion USD of damages worldwide. This number is projected to be 8 trillion USD in 2023 [1]. If this was a single nation, this would make it the third largest economy behind the USA and China. Such cybercrimes often involve malware - malicious software that can cause harm to a software system [2]. Famous examples of recent malware-based attacks include ransomware such as Ryuk [3] and supply-chain attacks such as the SolarWinds [4] and log4j [5] incidents. Over 1300 million distinct malware samples have been identified by AV-Test [6]. Such a large number of diverse malware highlights the need for analysis tools that are equally diverse in their approach to helping analysts prevent and identify potential attacks. However, while large-profile cases such as the SolarWinds incident [4] garner extensive media coverage, at a much smaller scale, the Swiss economy and its small and medium enterprises (SMEs) are affected by a large variety of increasingly devious malware [7]. This gives rise to the need for varied and innovative solutions that allow users to gain insights into malware behavior and to develop effective mitigation strategies.

1.1 Motivation

Dynamic analysis is one approach to analyzing malware. The malware's behavior is observed in a secure operating environment. By observing the malicious behavior, analysts hope to identify ways to improve detection or gain knowledge about new malware trends. The most common approach for dynamic malware analysis is to run malware in a VM (virtual machine) or emulator-based sandbox as highlighted in Section 3. There are many tools available that use this approach. However, many of these solutions have limitations for malware research and analysis. Cuckoo Sandbox [8] for example is currently unmaintained, ANY.RUN [9], JoeSandbox [10], and many other industry solutions have extensive analysis capabilities but are neither open-source nor free. Thus, deriving valuable insights into the relevant threats remains challenging, especially for SMEs with limited access to adequate funding and thus the required product licenses. More importantly, VMs often come with a considerable overhead, since for each VM a guest operating system (OS) is

required to be run [11]. For emulators, the footprint is even larger, as it involves the emulation of hardware components which requires system resources to do so. Another key issue of VM and emulator-based sandboxes is the fact that execution time differences between bare metal machines and virtualized ones allow malware to detect that they are inside a virtualized environment [12]. A potential solution for these issues is containers, which have gained little attention as sandboxes from both industry and research. Containers use the concept of OS virtualization, allowing a kernel to serve multiple separated user spaces. This greatly reduces their footprint [11] and thus increases their efficiency, but results in weaker isolation when compared to VMs or emulators due to the shared kernel of containers. This weaker isolation has been one of the main reasons why containers have seen little attention in research. One solution by Khalimov, Benahmed, Hussain, *et al.* [12] uses seccomp (secure computing mode) profiles to block certain (potentially malicious) system calls. In recent years there has been an industry effort to harden containers, most notably Google's gVisor [13] which implements the Linux system call interface with its own kernel. This allows containers to keep their near-native performance while combining it with the strong isolation which is typical for VMs.

1.2 Description of Work

This thesis describes the development of a prototypical implementation of *SecBox* as well as the acquisition of the theoretical background and related work required for its implementation. The prototype shall explore the feasibility of container-based sandboxing for malware analysis. The goal is to create a sandbox platform that enables security analysts as well as researchers from the cyber security domain to gain valuable insights into malware and its characteristics. The solution should be lightweight, allow for collaboration among peers, and be open-source. This is in contrast to existing solutions, where the most common approach to sandboxing uses comparatively heavyweight VMs or emulators. Many solutions are also unmaintained, closed-source, or require a commercial license. The prototype approaches the problem of dynamic malware analysis with the toolset provided by recent advances in containerization technology, namely gVisor [13]. With the goal of requirements elicitation for a dynamic malware analysis solution in mind, the problem space is explored on two fronts. First, an understanding of the current threat landscape is achieved both on a theoretical level by highlighting underlying theory, and on a practical level by analyzing prominent pieces of malware and their attack vectors in Section 2.2. Secondly, an in-depth analysis of existing dynamic analysis solutions available is performed in Chapter 3. Having established an understanding of the problem space, the design and implementation approach of *SecBox* is presented in Chapters 4 and 5 respectively. In order to evaluate the prototype, three case studies were conducted in Chapter 6 regarding the implemented solution. The thesis is concluded with a summary of what was achieved, the insights gained, and suggesting future work in the promising domain of container-based sandboxing.

Chapter 2

Background

This chapter will focus on establishing a common theoretical understanding of the terms and concepts relevant to the solution presented herein. First, an overview of malware and malware analysis is established, followed by the introduction of general sandboxing and virtualization concepts.

2.1 Malware

Malware, or malicious software, is a malignant piece of code that aims to intentionally cause harm or limit the functionality of a software system [2]. The threat posed by malware has been continuously present throughout the history of software, with an ongoing cat-and-mouse game between malware and its detection and mitigation tools [14]. The following section will give an introduction to malware and aims to provide a common understanding of the concepts surrounding malware.

Malware can be classified according to behavior, propagation mechanism, intended damage function, or exploited vulnerabilities. Due to the diverse and modular nature of malware, creating a classification scheme has proven challenging, but the following terms have persisted and cover many common behaviors of malware and are widely used in the cyber security community [15]:

- A **Virus** is a self-replicating malicious program that infects computer networks and host computers and remains passive as executable, making it reliant on user interaction. Depending on its complexity, a virus may be able to modify its own replicated copies [16], [15].
- A **Worm** is a piece of malicious code that is able to replicate and distribute itself actively through distributed systems and networks [15].
- **Trojan (Horses)** are software tools that are disguised to look like legitimate pieces of software, which when executed ensure remote access to an attacker to allow an attacker to damage the victim in any way they desire [16], [15].

- **Rootkits** try to hide its existence from security software such as antivirus software through sophisticated means and provide continuous root access to an attacker [15].
- **Spyware** allows an attacker to gain insights about the victim system and user behavior, such as keystrokes, passwords and other sensitive data [15] [17].
- **Adware** delivers advertisements to a victim, e.g., as pop-up ads and within other legitimate software. It is used to generate revenue by displaying unwanted advertisements [15].
- **Ransomware** infects a computer or network and holds its data or operation for ransom. This is often done through data encryption and subsequent ransom demands through various channels [15].
- **Crypto Miners** are pieces of malware that use hijacked resources in order to mine cryptocurrencies on behalf of criminals [18].
- **Bots** are pieces of software that allow an adversary to remotely take control of a host system in order to perform a specific operation. They often form so-called *botnets*, operated by a bot master or organized as a peer-to-peer system in order to mount a coordinated form of attack [15].

In the next section, these very general terms are explored in a more fine-grained manner and a systematic approach to malware classification is highlighted.

2.2 Survey on Malware-Based Attack Vectors

Attack vectors are any means by which an adversary can attain access to or control over a cyber asset in order to achieve some malicious effect [19]. The following sections highlight some of the most common attack vectors used by malware in the current cyber threat landscape. For the use case covered here, these attack vectors are explained by having an in-depth look at different specific malware, such as the Mirai botnet, and the attack vectors which they leverage in their operation.

2.2.1 Malware Component Analysis

While many common terms exist around the topic of malware and its analysis, there is no generally agreed-upon framework that describes comparative malware analysis. Pandey and Mehtre [20] provide a general malware analysis framework featuring 12 classes of activities, while Kok, Abdullah, Jhanjhi, *et al.* [21] researched ransomware using a software lifecycle-focused approach and, Polychronakis, Mavrommatis, and Provos [22] again use a lifecycle approach focused on web-based malware. The framework used to introduce some of the most common malware behaviors was proposed by Alrawi, Lever, Valakuzhy, *et al.* [23], where a comparative study of Internet of Things (IoT), desktop, and mobile malware was performed. Alrawi, Lever, Valakuzhy, *et al.* [23] propose the following 5 components of malware which are also depicted in Figure 2.1:

- **Infection vector** refers to how the malware attacks a system.
- **Payload** is the dropped malware code after exploitation.
- **Persistence** allows the malware to install (itself) on a system.
- **Capabilities** are the functions in the malware code.
- **Command and Control (C&C) Infrastructure** specifies how malware communicates with the operator.

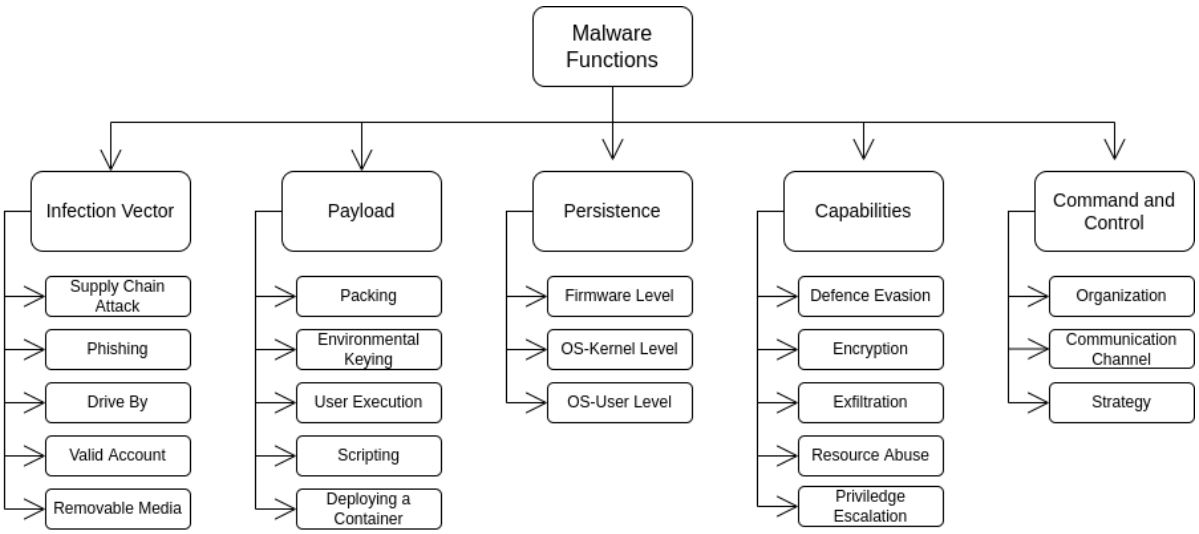


Figure 2.1: Malware Components and their Covered Exemplary Functions

Infection Vectors

There are many different pathways through which malware can infect a host system, differing in degree of sophistication, automation, and effectiveness. Such a method of delivering a malware payload to a host is called an *Infection Vector* [23]. In the following, some of the most common infection vectors by which malware enters a system are highlighted.

Supply Chain Attacks target software developers and suppliers, aiming to compromise their software build processes, source code, or update mechanisms to distribute malware. Increasingly popular, these attacks have been facilitated by the popularity of open-source software and package managers [24]. Examples include the recently highly discussed Log4J vulnerability [5] or the SolarWinds incident in 2020 [4].

Phishing refers to deceiving a target through electronic means with the goal of extracting information from or delivering a payload to a target. Phishing can take place through various means, such as email, instant messaging, dedicated websites, or even through social media [25]. Phishing can be executed in an extremely targeted manner, coined *spearphishing*. In this approach, a specific person or organization is targeted by leveraging previously obtained personal information [26].

Drive By Compromise refers to an attacker gaining access to a host due to the victim visiting a certain website while browsing the World Wide Web. The malicious content can take various forms, from approaches such as hosting entire websites to deliver malicious content, having a legitimate advertisement provider display malicious content on behalf of the attacker, or injecting malicious code into legitimate websites through various techniques, such as Cross-Site-Scripting (XSS) attacks [27]. However, all of these approaches focus on exploiting client-side software upon visiting a website [26].

An attack may be able to acquire a *Valid Account* to a host system. These valid credentials can then be abused by an attacker to intentionally infect a system with malware and launch further attacks on surrounding systems. Credentials can be acquired through various means. Some device manufacturers set default credentials for their devices, making them vulnerable to guessing attacks. This problem is extremely prevalent in Internet of Things (IoT) devices since many users do not alter the default credentials [28]. Similarly, devices that use easy-to-guess or often-used passwords are vulnerable, no matter the platform, be it mobile, PC, or IoT [23]. Another means of acquiring valid credentials is a *Brute Force Attack*, which is a repetitive interaction with an authentication service in order to try out different login credentials in consecutive attempts. Brute force attacks can take place in a distributed manner, against multiple targets, and in parallel. There are also more subtle variations that only attempt a login using common credentials, or target-specific combinations based on previously obtained information about the target, such as *dictionary attacks* [29].

Removable Media (DVDs, external storage, etc.) represent another means of infecting a system. By replicating malware onto removable media and leveraging *Autorun* features, malware can be disguised as a legitimate file or exploit a vulnerability that allows for its execution [26].

Payload

This component describes the form in which malware is delivered to a host system. The following paragraphs show common approaches to manipulating the formal characteristics of malware to allow it to fulfill its intended (malicious) purpose.

Packing is one of the most prevalently adopted set of obfuscation techniques used by Windows malware. It describes a class of techniques that allow a program to inject malicious code into a live process at run time by unpacking the payload from the compressed data [30].

Environmental Keying refers to guarding a piece of code against execution based on certain environmental variables in order to constrain execution to only specific target environments. This is done by having an encrypted payload check for target-specific values during decryption and having decryption fail if performed on an unintended target [26].

User Execution techniques rely on user interaction to execute a payload. These are often coupled with phishing techniques. Malware is embedded into malicious files, links, or images in order to execute upon user interaction [26].

Scripting interpreters allow an adversary to execute commands and binaries. They also provide common interfaces across many platforms. Examples of such interpreters include but are not limited to Unix Shell (macOS, Linux), PowerShell (Windows), Python (cross-platform, Python interpreter required), JavaScript (Client-Side), etc. These scripts can be either part of the initial payload delivered through an infection vector or provided after infection by the C&C infrastructure [26].

Deploying a Container may prove to be beneficial to an attacker since containers provide an encapsulated execution environment under the sole control of the attacker. They are also easy to deploy and may be used to evade defensive measures in place [26].

Persistence

Persistence components are the result of an adversary trying to maintain their hold on an infected host past an eventual shut-down of the system. This is due to the fact that malware usually operates in volatile memory. Persistence of malware can be established at three distinct levels.

Firmware level persistence refers to the malware modifying device or component firmware in order to establish persistence pre-OS boot. During device startup, various services are loaded, which can be hijacked before the OS even takes control [26], e.g., bootkits [31].

At the *OS-Kernel* level, malware often relies on automated script execution that is triggered by the OS at different stages of the boot process, such as the logon script in Windows [26] or by modifying the `initrd` or `initramfs` file system in Linux environments [32].

Malware can also achieve persistence at the *OS-User* level, which is also the most common approach to achieving persistence. Again, many different approaches exist and can be leveraged by malware, depending on the achieved level of authority on the system and hardware or software characteristics of the system. Common examples include the alteration of a writable file system [32], browser extension manipulation [33], and many more.

Capabilities

Capabilities refer to the core functions and modules present in modern malware. Malware can contain a combination of the following capabilities, all of which are designed to fulfill a piece of malware's malicious purpose.

Defense Evasion is a group of techniques designed to prevent malware activities from being detected by defensive techniques, tools, or cyber security professionals. This includes active detection avoidance as well as incapacitation of security mechanisms on a target device. Defense evasion tends to be highly specific to the device or platform [26], which makes an analysis of standard evasion techniques ineffective. However, in order to provide more concrete examples, this paper will provide specific examples of defense evasion techniques in Subsection 2.2.2.

Encryption functions refer to the encryption of data or systems in order to hinder a target from operating as intended. The most common use case of this function comes in the shape of ransomware, where after encryption, a ransom demand is made in exchange for decryption. However, this approach can also be used to completely render the system inaccessible permanently and cause permanent data or system access loss [21].

Exfiltration of data is another goal for which an adversary may employ malware. This type of function allows an adversary to steal data from a target's network and typically transfer it to the operator's C&C structure. Exfiltration can happen in an automated or manual fashion over many different channels, including physical media, Bluetooth, over a web service, such as GitHub, or, in a cloud environment, via a scheduled transfer of data to another cloud account under an adversary's control.

Malware may have the goal of utilizing a host's resources, such as network, processing power, or storage, for malicious purposes. This class of functions is called *Resource Abuse*. There are two main use cases where a target's resources are leveraged by malware: *Denial of Service* and *Cryptojacking* attacks.

(Distributed) Denial of Service (*(D)DoS*) attacks are intentional attempts to deny legitimate users from using a specific network resource by using one (or multiple) infected device(s). There are two main approaches to achieving this. The first approach, a *vulnerability* attack, focuses on confusing a network protocol or application by sending incorrectly formatted packages. The second approach tries to exhaust a specific resource, either on the server's side (Central Processing Unit (CPU) capacity, sockets, memory, etc.) or on the connection level by exhausting network capacity. Historically, most DoS attacks have been distributed in nature [34]. Cryptojacking on the other hand was sparked by the spike in the value of cryptocurrencies in recent years and describes the abuse of a compromised resource in order to complete proof of works (PoWs). The compromised resources are organized in so-called mining pools. The most commonly illicitly mined cryptocurrencies are Monero and Bitcoin [18].

Privilege Escalation consists of functions that allow an adversary to reach a higher level of permissions. While exploration and intelligence gathering might be possible with low-level permissions, in order to achieve their desired outcome, malware needs to achieve the required permissions. To this end, malware often takes advantage of system vulnerabilities such as misconfigurations and exploits in order to gain elevated access. In the context of this paper, the most relevant form of privilege escalation is an *Escape to Host*, which refers to an application breaking out of a container and gaining root access to the host machine [26].

Command and Control Infrastructure

Command and Control (C&C) tactics are ways by which malware communicates with its operator and vice versa. The goal is to achieve consistent and secure communication that is not detectable by defense mechanisms, such as antivirus (AV) software, in order to allow an operator to control a compromised device consistently.

The *Organization* of C&C architectures can be either peer-to-peer [35] or centralized, with a centralized architecture being the more prevalent one [20].

While standard web protocols such as Hypertext Transfer Protocol (Secure) (HTTP(S)) remain popular, the diversity of *Communication Channels* in malware has been increasing due to the increase in classic network-traffic analysis tools used by cyber security professionals. This has led to malware communicating with their C&C architecture through social media (such as Twitter, YouTube and Facebook) [36], SMS [26], Web services (such as GitHub and Google Documents) and other non-standard and hard to detect communication channels [36].

Malware employs many *Strategies* in order to mask the communication with its operator. The traffic is often designed to mimic traffic occurring during normal system operation. The methods employed vary depending on the victim's defenses and the desired stealth level, but common examples include data encoding, insertion of junk data, or dynamic resolution for communication with its command and control infrastructure [26].

2.2.2 Malware Families and their Behavior

With the number of distinct malware reaching well over 1300 million according to AV-Test [6], it has become clear that the threat of malware cannot be handled by addressing each malware individually. Malware continues to evolve, with many versions of the same malware affecting systems at the same time, code being shared among threat actors (TA), and the trend of Malware-as-a-Service (MaaS), all contributing to making the malware landscape extremely convoluted. For this reason, security researchers introduced the concept of malware families, which describe groups of samples of malware that share large sections of their code base and thus behave in similar fashions. In the following sections, four families of malware are showcased that plague different platforms and different types of targets that have gained notoriety due to the damage done, their perseverance, or their deviousness.

The Mirai Botnet

Mirai is a piece of malware that started infecting IoT devices in the fall of 2016. Its propagation resulted in a botnet that proceeded to launch some of the biggest DDoS attacks in volume on record. Among its targets were the Krebson Security Website [37] and OVH [28], [38]. Other than the sheer volume of the attacks the botnet was able to mount, another thing made the case of Mirai special in the realm of malware attacks: namely the fact that shortly after the aforementioned attacks, the Mirai source code was released online [39]. This allowed security researchers to identify Mirai's functionality and infer that it was a more evolved version of the previously studied BASHLITE (also known as gafgyt, LizardStresser, or Torlus) malware [40]. However, this also led to other malware creators integrating the Mirai malware into their own malware or adapting its source code to leverage other exploits. The most notable of these was an attack against Deutsche Telekom AG [41].

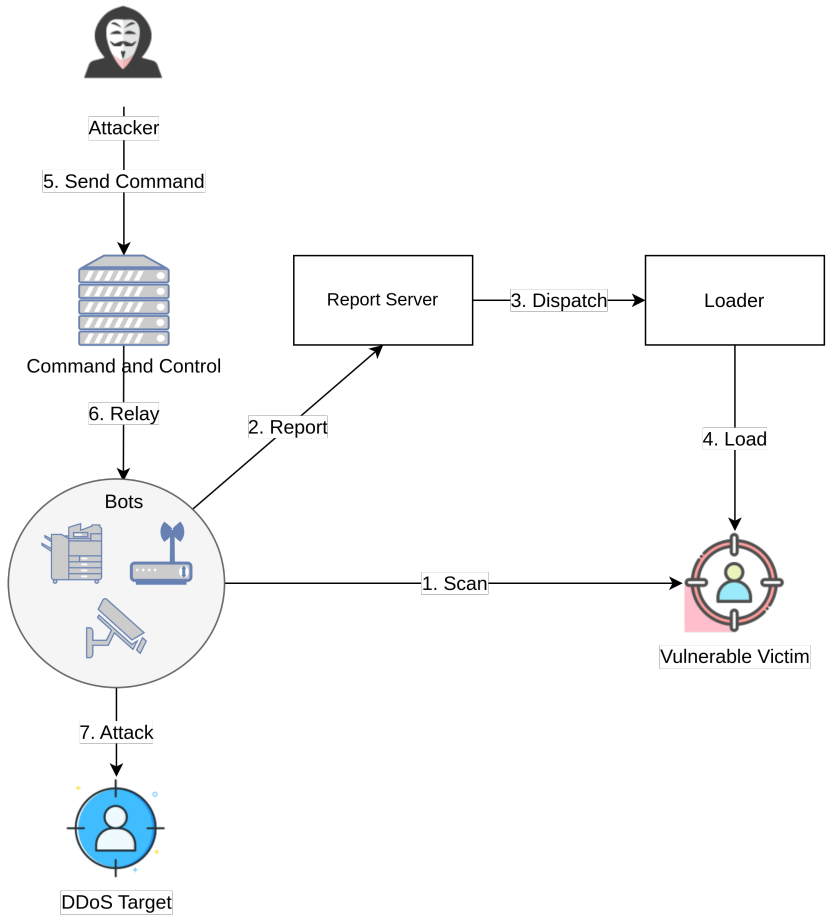


Figure 2.2: Mirai Operation adapted from [28]

Figure 2.2 presents how the Mirai malware and botnet operate. In the first step (1), Mirai performs a rapid scan of the available IP (Internet Protocol) address space, excluding a blacklist of hard-coded exempted IP addresses. This is often done using standard networking tools, such as `arp-scan` [42]. The scan happens by asynchronously sending transmission control protocol (TCP) SYN probes to an IP address on ports 23 and 2323. If a potential victim is identified, Mirai applies a valid account approach as its infection vector, by performing a dictionary attack of 10 username and password combinations from a hard-coded list of 62 commonly used credentials. If a login attempt succeeds, the login information, and the address of the victim, are sent to a hard-coded report server (2). In (3), a loader program infects these devices and in the fourth step it drops an architecture-specific payload.

Mirai does not attempt to achieve persistence, but rather has the capability to evade potential defensive measures by hiding its presence, renaming its run process, and deleting the downloaded binary. At this point, the device is fully compromised, scanning for other potential victims, and waiting for commands from the C&C architecture. The devices targeted by Mirai were mostly consumer-grade routers, security cameras, and more. However, the spectrum of targeted devices evolved after the release of the original source code to include other IoT devices as well, diversifying the threat. An attacker

could then decide to launch an attack through their C&C structure (steps 5. and 6.), which sparks the bots to launch a DDoS attack on the desired victim. The protocols used by Mirai for its DDoS attacks, its infection vector, and the communication with the C&C architecture have expanded during its lifetime to include HTTP, User Datagram Protocol (UDP), and many more [28].

Emotet aka Geodo, Heodo

Emotet, also known as Geodo or Heodo is a trojan that first appeared with the name Geodo in German-speaking countries in June 2014 [43]. Emotet targets Windows systems in the banking or public sector. In 2018, such an attack was named one of the most costly threats affecting governments, private and public sectors [44]. The success of the Emotet malware may be attributed to its modular nature, allowing it to evolve to utilize many different techniques by incorporating new functional components.

Emotet most commonly uses spearphishing in the form of sophisticated emails with malicious files or URLs as its infection vector. Due to its worm-like nature, the emails might originate from trusted sources, allowing the malware to spread more easily. Its payload is often integrated as malicious Visual Basic for Applications (VBA) code into Microsoft Office files [45] or as malicious scripts (Powershell, Windows Command Shell) executing and downloading the Emotet malware. After infection, the malware executes several capabilities. It attempts to evade defense measures in place by injecting its code into running processes, obfuscating its executed code, changing its file timestamps (so-called *timestomping*), or even detecting potential sandboxes through user interaction checks. Emotet additionally has been known to attempt a valid accounts approach on surrounding network devices using a dictionary attack, exfiltrating email contact information for further propagation as well as browser credentials. Lastly, Emotet tries to escalate its privileges through various exploits in the Windows kernel [46].

Emotet has been observed to establish persistence at either the OS-kernel level through automatic execution upon boot or at the user level by altering the writable file system or job scheduling [47]. Communication with its C&C server is often encrypted using RSA keys and usually happens via HTTP requests [46]. In recent years, despite its many potential malicious functional components, Emotet has evolved to become malware that focuses mostly on infecting systems and subsequently downloading other malware to the victim's system, such as Trickbot [48], Dridex [49] or Ryuk [48]. Due to its high profile, the Emotet malware infrastructure was taken down in a global effort in early 2021 [50].

Ryuk

Ryuk is ransomware that first appeared in 2018 used to target enterprises that are able to pay its relatively large ransom demand of 15-bitcoin (BTC). The malware is not widely distributed due to the fact that it is mostly leveraged in targeted attacks against single entities. Its source code bears similarities to the HERMES malware [51].

Ryuk is often delivered onto a previously infected system by other malware, such as Emotet or TrickBot [48]. The payload is delivered through a so-called dropper that writes 32-, and 64-bit Ryuk payloads to system-specific folders. These files are then executed through PowerShell scripts, deleting the original dropper to hide its presence. The malware has several capabilities that are performed upon execution. After a sleep period, over 180 processes that are related to defensive measures in place in a system are stopped. Its main capability, the encryption of data and devices, is performed by injecting code into running processes and using symmetric and asymmetric encryption techniques to encrypt files on the victim's system. These files are appended with the `.RYK` file extension and a ransom note is created either as a `.html` or `.txt` file. Connected devices are turned on using the Wake-on-LAN feature and subsequently encrypted [51].

Ryuk's code is obfuscated in order to prevent static analysis from security researchers and to prevent detection by system defense measures. Persistence is established on the OS-kernel level by leveraging autostart execution or at the user level by creating a registry [52]. Ryuk has been reported to use commercial tools such as Cobalt Strike [53] to communicate with its operators [3].

FluBot, FedexBanker, and Cabassous

FluBot malware is a modern Android banking trojan that originated in late 2020 and started out targeting Android mobile devices in Spain. The malware proved to be extremely infectious, coining its name. Its developers continuously released new versions of the malware that countered efforts by law enforcement and security specialists to prevent its spread. Subsequent versions added new features, infection vectors, obfuscation techniques as well as changes to its C&C architecture [54].

The malware most commonly leverages phishing SMS as its infection vector, mimicking legit applications such as shipping services, FlashPlayer, or voicemail services [54]. The malware would then rely on user execution to download onto a victim's device, with the user also providing the necessary permissions to FluBot (*i.e.,* internet access, phone calls, reading SMS, etc.). In earlier versions, the malware used APKProtector [55] to obfuscate and pack its payload, switching to a custom packer in later versions. FluBot also uses a specific kind of environmental keying in order to target users from certain countries by whitelisting devices and checking the system language of the infected device. Persistence is established as a regular user-level app. The malware focused on the exfiltration of contact data and banking credentials from the target device and abused an infected device to further spread the malware to send phishing SMS to potential targets. The malware calls its C&C server through RSA-encrypted HTTP requests using a domain generation algorithm [26]. Later versions of the malware also allowed the operator to directly send commands to infected devices, e.g., blocking incoming notifications, uninstalling certain apps, or disabling play protect [56]. Flubot's Russia-based infrastructure was taken down in a joint effort of international law enforcement agencies in July 2022 [57], however, whether this means the end of FluBot remains to be seen [58].

2.3 Malware Analysis

The general goal of malware analysis is to understand the behavior of a given malware sample, often with the intention to improve detection techniques or react to new malware trends. To prevent such an analysis, malware authors in turn employ evasion techniques. This leads to an arms race between malware analysts and attackers, where the former continually improve upon malware analysis tools and techniques, while the latter create new methods to evade analysis [59]. In most literature, malware analysis approaches are classified into *static* and *dynamic analysis*, with some researchers also considering a third category: *memory analysis*.

2.3.1 Static analysis

Static analysis focuses on analyzing malware by decompiling it and inspecting its source code. This source code can then reveal information about the malware's (malicious) intent. Hence, malware authors aim to obfuscate their source code through techniques such as polymorphism, metamorphism, and packing [60]. Static analysis techniques can also be applied to retrieve information from the compiled binary [61]. However, once compiled, a lot of information gets lost (*e.g.,* variables or the size of data structures) [59]. This highlights the limitations of static analysis: Attackers obfuscate their source code so that it is not readily available for analysis. This means that malware analysts must deal with the intricate challenges of analyzing binaries. While unsophisticated malware can be detected by static analysis, more complex malware is often able to evade simple static analysis. As a result, their malicious behavior can only be observed once they are executed [61].

2.3.2 Dynamic analysis

In *dynamic analysis*, the malware is executed in order to analyze its behavior at run time. From a high-level perspective, two approaches for dynamic analysis can be distinguished: *in-the-box* and *out-of-the-box* [62]. In the former, all the tools for malware analysis are installed in the same OS. According to Chakkaravarthy, Sangeetha, and Vaidehi [61], in-the-box approaches are self-surgical and more efficient, due to the fact that the anti-malware and debugging tools are installed on the same OS on which the malware runs. However, this makes the approach prone to result in vulnerabilities due to its direct kernel access. Consequently, kernel access can be exploited by malware (e.g., through rootkits) to stop the analysis. In the out-of-the-box approach, the OS with the malware is isolated from the analysis tools, increasing security [61].

The aforementioned malware analysis arms race also poses a problem for dynamic analysis. Dynamic analysis is limited to only observing artifacts that stem from the execution. This property is exploited by malware that tries to detect analysis tools or platforms: if the malware suspects that it is in an environment that is being analyzed, it either terminates or exhibits non-malicious behavior [61], thus evading analysis.

Memory analysis is a third category that focuses on memory. When malware infects a system, it usually leaves artifacts in the Random Access Memory (RAM) of the infected system. First, a snapshot of the physical memory image at one or more point(s) in time is taken, which is then analyzed [63]. The advantage is that even obfuscated malware often leaves traces in the memory. Rathnayaka and Jamdagni [64] propose an approach in which the memory is dumped after malware is executed in a sandbox and then analyzed. They also combine this approach with static analysis.

A combination of these three approaches is often used for in-depth analysis, as malware naturally tries to evade detection and analysis.

2.4 Sandboxing

A widespread problem in IT security is how to deal with the execution of untrusted applications in a given environment. It is impossible to determine whether any given application is safe to be executed; to quote Thompson [65, p. 763] "You can't trust code that you did not totally create yourself.". A possible solution is to run such applications in a secure environment, *i.e.,* a sandbox, thus decreasing the risk of a security breach by restricting the application's calls to the actual environment [66].

With this in mind, it becomes fairly obvious why sandboxes have been widely used for malware analysis. They allow analysts to run malware in a secure and isolated environment, which they equip with tools to monitor the malware's behavior at run time. This of course is well-suited for dynamic malware analysis, as it allows the malware to be analyzed without fearing its spread. Of course, this means that a sandbox actually needs to be secure: This is often achieved through hardware virtualization or full-system emulation, a few examples of which are Cuckoo [8], Any.Run [9], and LiSa [67]. Kruegel [68] highlights the three goals of a sandbox for dynamic analysis:

- Visibility: a sandbox should see as much as possible from the execution of a program.
- Resistance to detection: malware should not be able to easily detect that it is inside a sandbox, otherwise it might obfuscate its presence.
- Scalability: The solution should run numerous samples without samples interfering with each other.

They also emphasize the fact that the challenge is not to build a sandbox, but rather to build a good one. A lot of malware employs anti-VM or anti-sandbox techniques [69], and some malware even targets VM-specific vulnerabilities to attack other VMs or escape from the VM [70]. Due to such techniques, building a sandbox that fulfills the above goals remains challenging.

Sandboxes run into a major challenge when they aim to analyze malware, as malware authors integrate anti-analysis behavior into their malware to evade analysis. This issue is based on the assumption that systems of interest can be identified based on characteristics

that differentiate them from a regular execution environment, *i.e.,* an environment where no analysis is performed. This relates to the resistance to detection of a given sandbox. Anti-virtualization, or anti-VM, techniques represent a subset of general anti-analysis techniques.

There are other anti-analysis technique categories, such as anti-debugging, anti-disassembly, or obfuscation [71]. However, the most relevant ones for dynamic analysis are anti-VM techniques. Branco, Barbosa, and Neto [71] found that 81.40% out of 4'030'945 analyzed malware samples use anti-VM techniques. Chen, Andersen, Mao, *et al.* [72] estimates that 40% of all malware shows less malicious behavior when executed in a VM or with a debugger attached. This indicates how widespread such behavior is.

These techniques can also be abstracted into the term environment awareness: Guibernau [73] names five techniques that allow malware to determine whether it is executed in a sandbox environment or to detect the presence of forensic tools.

The first is **system architecture**, which consists of finding information about hardware components, system specifications, or system footprints. An example of this is hardware IDs like the processor ID, which may be undefined in a VM. The **system background** can deliver further hints, as VMs may reveal their presence in MAC addresses or registry keys. Still, such artifacts can often be circumvented through clever design. More difficult to deal with are techniques that use **time-based detection** or **user-based detection**. The former techniques may use extended sleep, *i.e.,* not showing activity for a set amount of time to fool analysts. The latter may look for user interaction to determine whether it is in a sandbox.

This also led to the rise of user behavior emulation [74] as an anti-anti-analysis technique. The final environment-aware technique described by Guibernau [73] is **network-based detection**, where malware checks whether it can connect to the internet or which ports are open. Mitre [26], a knowledge-base of adversary tactics, have a similar classification approach, where they distinguish between system checks, user activity-based checks, and time-based evasion [75]. Overall, this highlights the need for sandboxes to continually evolve to adapt to the ever-changing anti-VM and anti-sandboxing techniques that exist.

2.5 Virtualization

With the increasing popularity of the cloud, virtualization technologies have gained significant attention as they allow physical resources to be split into virtualized systems, thus enabling their shared usage. Virtualization is used in various scenarios, such as server consolidation, debugging, testing, software migration, and sandboxing [76]. Especially for sandboxes, security-related concepts are highly relevant. For a comprehensive understanding of such concepts, the x86 processor security-ring model must first be discussed. This widely used type of processor employs a concept called security rings. These rings define the privilege level of instructions that are run in the processor [11] [77]. They are labeled from ring level 0 (kernel space) to ring level 3 (user space). Ring levels 1 and 2 are not used by traditional OSs. Programs get access from the user space to the kernel

space via system calls. Most OSs have a kernel space containing drivers and fundamental components which communicate directly with the hardware, and a user space containing the applications that run on the OS.

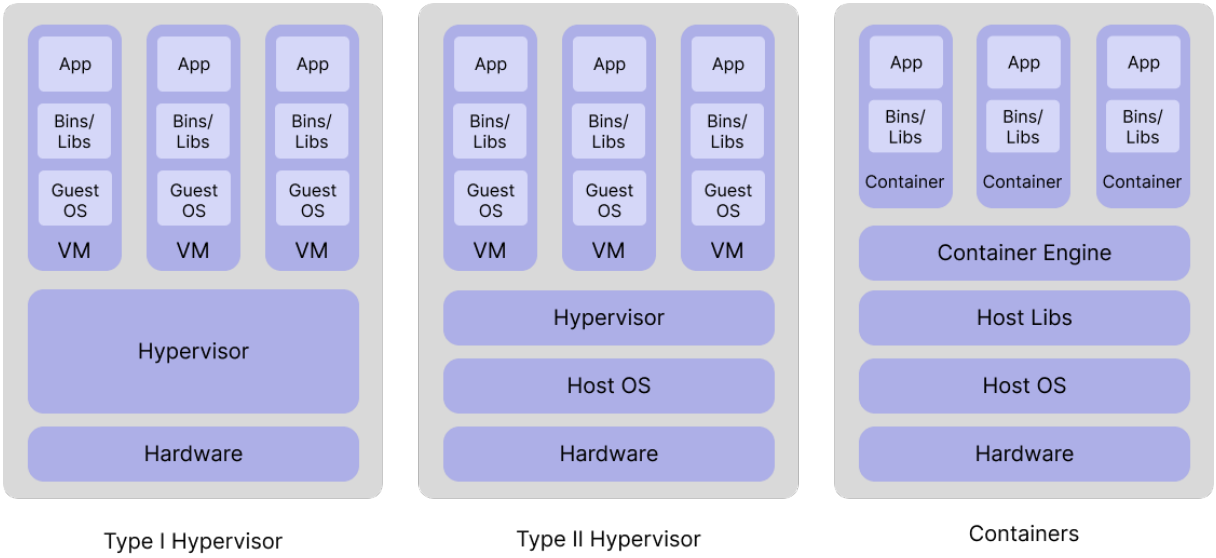


Figure 2.3: Comparison of Hypervisor-based and Container-based Virtualization

These rings, combined with the hypervisor, provide isolation between VMs. Hypervisors, also referred to as Virtual Machine Monitors (VMM), provide a layer between the guest OS and the hardware. They are usually implemented as separate hypervisor kernels or as Linux Kernel Modules [11]. Two types of hypervisors can be differentiated: Type I hypervisors run on bare metal, while type II hypervisors run on a guest OS as a user-level application. This difference can be seen in Figure 2.3. In either case, the hypervisor communicates directly (type I) or indirectly (type II) with the hardware and makes it shareable between VMs, by scheduling CPU time and allocating physical memory to each VM. Consequentially, the guest OS is run on a less privileged ring level. The notable exception is hardware-assisted virtualization, where the hypervisor is run in a host domain, commonly referred to as ring level -1, allowing the guest OS to be run on ring 0. This kind of virtualization decreases the complexity of the hypervisor and overhead [11].

In hypervisor-virtualization, a software program runs on the underlying hardware, the hypervisor only controls the access of different VMs to said hardware. Emulators are software programs that simulate how a different program or a piece of hardware functions [68]. This makes them interesting for sandboxing since they can fully emulate systems. Full-system emulation allows emulators to simulate hardware (most importantly CPU and physical memory), allowing an entire OS to be installed on top of the emulator. This not only helps to obfuscate the existence of a sandbox but also allows it to see every instruction that is issued inside of the sandbox [68]. However, this also comes with great cost, as emulation is resource intensive because the entire hardware system needs to be emulated in software. An example of a processor emulator that can be run as a full-system emulator is QEMU [78]. QEMU can also be used on many hypervisors, such as KVM and XEN [11] (both of which are hardware-assisted) to provide hardware emulation. It supports the emulation of various processor architectures such as x86, ARM, PowerPC, and Sparc [76].

2.5.1 Containers

Containers represent virtualization at the OS level. The central idea is to provide a VM-like environment without the overhead of running another kernel and without simulating the surrounding hardware to enable fast deployment of software [79]. Containers can achieve near bare metal performance of the CPU, memory, disk, and network [80]. Unlike hypervisor-based virtualization, container-based virtualization does not use a hypervisor, as depicted in Figure 2.3. Instead of virtualization on the hardware level, it happens directly on the OS level. This means containers share an OS kernel, increasing the performance as no OSs are run redundantly. However, the isolation is weaker when compared to VMs, due to the shared kernel. There are many products available that leverage container technology, such as Docker [81], Linux Containers (LXC) [79], or gVisor [13]. All three are Linux-based. gVisor is especially interesting for this use case, as it provides an additional layer of isolation through a user space kernel. While it implements only a limited number of system calls to the host kernel, it provides strong isolation.

Linux Containers

In LXC [79], containerization is achieved through the use of certain Linux kernel features [11]. `Cgroups` or control groups can be used to group processes together as well as to limit and monitor their respective resources [82]. `Namespaces` wrap a system's global resources in a software representation that provides the illusion of an isolated instance of the respective resource to all member processes of the namespace. There are currently eight namespace types available on Linux [83]. `Capabilities` offer the option to provide certain superuser privileges to threads in a fine-grained manner [84]. `seccomp-bpf` allows restricting the system calls that a process can make [85].

Despite these powerful tools, there remain a number of security issues present in container-based systems. Firstly, the container management software usually runs as a privileged process, making it a prime target for privilege escalation [86]. Also, container-based systems often use publicly available container image registries, such as Docker Hub [87], which facilitate malware distribution through infected images. Lastly, the potential of a container breaking the isolation (e.g., [88]) represents a reoccurring problem, with new kernel vulnerabilities being discovered at a high frequency [89]. One noteworthy development in the realm of container security is presented by Sun, Safford, Zohar, *et al.* [90] who suggest the extension of the eight available namespaces by a so-called security namespace. This would allow containers to monitor and mitigate internal threats similarly to VMs by applying autonomous security control.

Docker

Docker [81] is a platform building on LXC that attempts to simplify developer interaction with the aforementioned concepts and make the deployment of containers easier and more accessible. Figure 2.4 shows the basic functionality docker provides. Docker introduces a few new concepts, such as images, which are read-only templates with instructions for

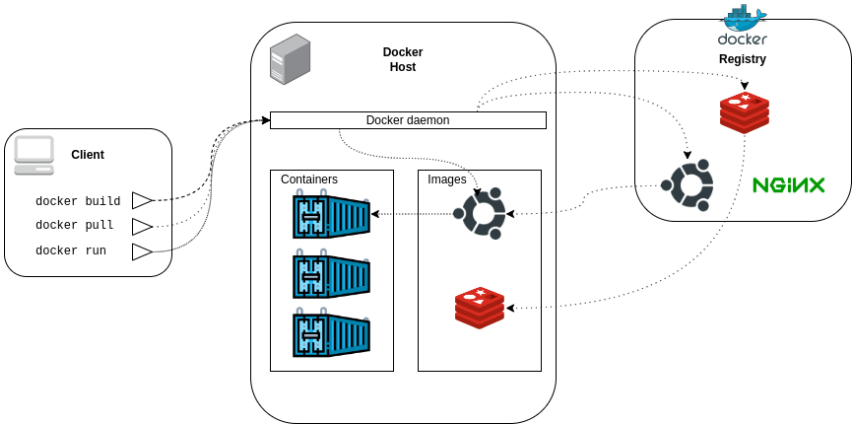


Figure 2.4: The Docker Platform Figure Modeled after [91], [92], [93], [94]

creating a docker container. Images are defined by a so-called *Dockerfile*, which is similar to a script for setting up a container. Many of these images are hosted on a public registry. Docker also allows its users to host their own private registries. Images are managed by *Docker daemon* or *dockerd* which also manage the containers, networks, and available volumes [91].

gVisor

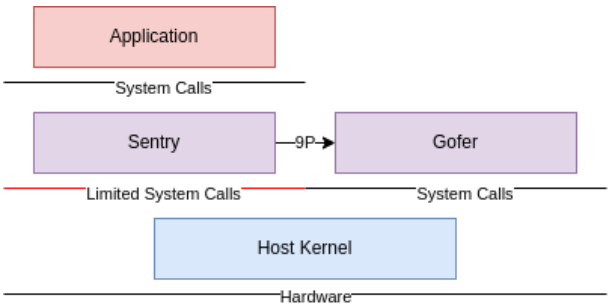


Figure 2.5: gVisor Architecture [13]

gVisor [13] is a user-level kernel that is implemented from the ground up in Go. It offers a large portion of the Linux system call interface. For an amd64 architecture running Linux, out of 349 system calls, 261 are either fully or partially supported. On an arm64 architecture, out of 292 system calls, 225 are (partially) available. gVisor also implements the Open Container Initiative (OCI) [95] runtime and thus can be used with both Kubernetes and Docker. gVisor provides a sandboxed container environment to ensure system security despite running potentially malicious or untrusted applications in the cloud.

In Figure 2.5, a high-level architecture of gVisor is shown. The *sentry* implements the user-level kernel. Since the sentry is itself a user-level application, it does not pass the system calls made by the application to the host kernel. It does however require some system calls to be made to the host kernel in order to enable its operation. The *gofer* is

a process being started for each container and providing access to file system resources to the sentry. The gofer communicates with the sentry over the 9P protocol [13].

Chapter 3

Related Work

To gain an overview of existing solutions, this chapter will first highlight sandboxes that implement either VM or emulator-based solutions. In the second part, existing approaches to container-based sandboxes are listed and explored. However, it is noteworthy that these are not the only options for malware analysis. For example, binary emulation does not recreate a whole environment but only emulates functions, system calls, and operating subsystems [96]. This represents an approach to making dynamic analysis less resource intensive. Tools that use sandboxing are more resource costly, but also provide more granular analysis data [96].

3.1 Methodology

There exist numerous sandboxes that use emulation or virtualization. In order to gain an overview, a list was compiled of popular sandbox malware analysis tools from industry and research to give the reader an overview. Note that many of these tools, especially commercial tools, use a combination of different techniques and technologies to analyze malware. Table 3.1 is based on a systematic literature research conducted by searching for the terms *sandbox*, *malware*, and *analysis* on Google Scholar, followed by a backward literature search. Due to the large number of tools that exist, this analysis considers 14 relevant sandboxes, with a focus on more recent and open-source sandboxes.

To identify differences between existing solutions, a number of properties are introduced to determine where there is room for improvement. This list is incomplete, partially due to the fact that closed-sourced solutions do not provide in-depth descriptions of their architecture. Furthermore, a stronger focus was placed on solutions that have been maintained during the last five to ten years, given that the malware landscape is ever-evolving, which leads to rapidly outdated tools. However, some examples of older sandboxes, most notably CWSandbox [97] and Ether [98], have been included in order to be as thorough as possible.

The first of these properties is the OS of the sandbox, *i.e.,* the OS on which the targeted malware should run. In Table 3.1, it is visible that there are solutions for various platforms,

the most common target platforms are Windows and Linux. All listed tools use either VMs or QEMU to implement their sandbox. In regard to business models, there are two main approaches: generally, the systems are either closed-source and require a subscription (industry), or they are open-source. The former often also offers a basic free model with limited analysis capabilities. A general issue is that many of these analysis tools eventually become unmaintained due to various reasons. This problem is more acute with open-source software, as there often is little to no financial incentive to keep development going. A project was determined to be unmaintained if it is either explicitly stated by the developers or - in the case of open-source solutions - if no issues have been closed on GitHub for more than 12 months. Furthermore, an issue with closed-source software is that analysts have little information about the inner workings of the software. This makes it challenging to properly assess the quality and find potential weaknesses of a sandbox. A further property is whether the sandbox supports network routing. This allows users to (securely) analyze network traffic. This is especially interesting for malware that possesses a C&C structure, as analyzing the network (e.g., by simulating internet access) helps with the identification of said structure. Finally, the solutions were classified whether they support interaction from the user. Many sandboxes employ a fully-automated analysis to facilitate the analysis process. While this increases usability, especially for users who are less versed in technical details, it can also limit the analysis capabilities. This is due to the fact some malware only shows malicious behavior when user interaction is present.

3.2 Survey on Sandboxes for Malware Analysis

3.2.1 Hypervisor- and Emulator-based Sandboxes

The findings of the systematic literature research have been summarized in this section.

Cuckoo Sandbox

Cuckoo Sandbox [8] is one of the most noteworthy sandbox analysis tools, as it has seen widespread usage for dynamic malware analysis [99] [100] [101]. On Google Scholar, searching "Cuckoo Sandbox" gives over 3'200 results, showing the attention this sandbox has received from academia. The sandbox has two main components: the first component is the Cuckoo host, which is responsible for the guest and analysis management. The second one is the analysis guest, which is an agent that runs in an isolated environment with the malware. The behavior of the sample is then reported to the host via an isolated, virtual network [8].

The project is currently unmaintained, however, there are plans for a complete rewrite of the tool in Python 3 [102]. There are a number of works included in the list that at their core are simply extensions of Cuckoo sandboxes, namely malwr [103] and CAPE [104]. Malwr is an online platform that allows users to upload samples without installing a sandbox on their own system. CAPE is based on the Cuckoo Sandbox but has seen its

own development and is still maintained by its community. CAPE also offers an online platform for malware analysis [105].

Cuckoo is compatible with a number of VM technologies such as Virtualbox, Xen, and VMware [106]. One of the main issues of Cuckoo sandboxes is their detectability. Ferrand [107] names a number of approaches to reduce the visibility of Cuckoo, however, complete obfuscation of the sandbox environment remains infeasible. Still, obfuscating the virtual environment as much as possible is valuable, as malware can check the execution environment for sandbox-related artifacts [73]. Cuckoo follows an in-the-box approach since an agent module has to be installed inside the environment [106].

Drakvuf

Unlike Cuckoo's in-the-box approach, Drakvuf [108] leverages virtual machine introspection (VMI), which enables it to analyze processes without installing an agent on the host machine. This out-of-the-box approach enables it to collect data by tracing system calls and kernel functions at the hypervisor level. The motivation behind this tool stems from papers such as [109] that highlight the importance of data from the kernel-level rather than just the user-level. They found machine learning (ML) malware classification models to be more effective when trained on kernel-level data. Besides the ability to trace kernel functions, Drakvuf also claims higher stealthiness due to the lack of a host-injected agent and higher throughput (number of processed samples per time period) when compared to Cuckoo [108].

Industry Solutions

There are various sandboxes for malware analysis that were developed by the industry. JoeSandbox [10] is a product family developed for automated dynamic malware analysis. It offers various types of sandboxes, covering many popular operating systems and using various types of sandboxes, such as VMs, emulators, and physical machines. JoeSandbox has subscriptions for both cloud and on-premise solutions. Falcon Sandbox [110] is another tool that, in addition to on-premise and cloud solutions, also offers a free community platform for malware analysis [111]. The third tool is Any.Run [9], a cloud-only solution. This tool is limited to Windows but is one of the few tools that support real-time interaction from the user. A general issue with all of these solutions is the fact that they are closed-source. This makes it challenging to properly evaluate the architecture and efficiency of these tools with anything but black-box approaches. Furthermore, they also come with a relatively high price – especially when one considers the competing free solutions.

Research Solutions

Over the years, researchers have come up with various approaches to sandboxing. Willems, Holz, and Freiling [97] created CWSandbox which uses API hooking, *i.e.,* it intercepts

calls to Windows API functions to record which functions have been called and to block them if necessary. The CWSandbox itself is executed inside a Windows VM, so there is still virtualization used to achieve the malware analysis. A major weakness of this approach is that it is easily detectable and even escapable [112]. Dinaburg, Royal, Sharif, *et al.* [98] propose an improvement by using hardware virtualization extensions, namely Intel VT using a Xen Hypervisor running Windows as a guest OS. The resulting system is called Ether. This makes the Sandbox harder to detect, while still logging the executed instructions, memory writes and system calls made by a guest process.

With the rise of the IoT, researchers have come up with sandboxes that focus on Linux architectures. These tools often use full-system emulation, mostly with QEMU, rather than hypervisor-based virtualization. This enables analysis tools to emulate different CPU architectures, which is especially useful when the malware that is to be analyzed targets IoT devices that have a different hardware architecture than the machine that performs the analysis. One of the Linux-based systems that use VMs is Limon [113]. Since it is VM-based, it can only analyze malware for i386 and x86-64 architectures. Uhřıek [67] proposes LiSa, which uses full-system emulation and thus can also emulate ARM architectures that are often targeted by IoT malware. ElfDigest implements LiSa sandboxes as SaaS. There have also been sandboxes that specifically target IoT malware. Notably, IoTBox [114] was introduced as the sandbox component of IoTPot - a honeypot for IoT malware. A honeypot has several similar properties to a sandbox, namely its monitorability, and is a system component intended to be compromised by malware and cyber attacks in order to alert its operators to potential threats. IoTPot also supports various different architectures. There is also V-Sandbox [115] which makes it possible to simulate a C&C server in addition to the support of various architectures. Finally, Tamer [116] uses dynamic binary instrumentation to monitor system call instructions in the guest system and VMI to identify the process of target malware. Just like Limon or V-Sandbox, it can use simulated C&C servers. However, Tamer does not support multiple CPU architectures.

3.2.2 Container-based Sandboxes

As described so far, there are various existing solutions to the problem of dynamic malware analysis using sandboxes. Most of them either use system emulation or hypervisor-based virtualization to create a secure sandbox. While these types of sandboxes are secure and able to isolate the environment, they are also resource-intensive, since a whole operating system with all its overhead has to be run. Furthermore, the existence of a whole environment also introduces a lot of noise [96], which may hinder analysis capabilities.

Still, there is one virtualization technique that has barely been used for sandboxes, despite its popularity in other areas: containers. Khalimov, Benahmed, Hussain, *et al.* [12] explore a prototype that uses a container-based sandbox as an alternative to hypervisor-based ones. Their groundwork explores different ways to recreate a regular execution environment as closely as possible. They isolate the container by using Secure Computing Mode (seccomp), which is a kernel feature of Linux that blocks (or allows) specified system calls issued by processes according to the seccomp profile of the container. Rather

Table 3.1: Overview of Malware Analysis Sandboxes

	OS of Sandbox	Sandbox Type	Business Model	Maintained	Network Routing	Real Time Interaction
Cuckoo	Windows Linux macOS Android	VM	Open-Source, Free	No	Internet InetSim Tor VPN	No
malwr	Windows Linux macOS Android	VM (Uses Cuckoo Sandbox)	SaaS, Free	No	No	No
CAPE(v2)	Windows Linux macOS Android	VM (Extension of Cuckoo Sandbox)	Open-Source, SaaS Free	Yes	Internet InetSim Tor VPN SOCKS	No
CWSandbox	Windows	API hooking (in a VM)	n/a	No	No	No
Any.Run	Windows	VM	SaaS, Subscription, Free Community SaaS	Yes	VPN, Fakenet, Geolocation	Yes
JoeSandbox	Windows Linux macOS Android iOS	VM, hooking, emulation and bare metal	SaaS and On-Premise, Subscription	Yes	VPN, SSL Proxy, Localized Internet Anonymization (Geolocation)	No
LiSa	Linux	QEMU	Open-Source, Free	No	VPN	No
ElfDigest	Linux	QEMU (Uses LiSa)	Closed-Source, Free	Yes	No	No
Ether	Windows	VM	Open-Source, Free	No	No	
Falcon Sandbox	Windows Linux macOS, Android	n/a, most likely VM	Closed-Source, Subscription, Free Community SaaS	Yes	Tor, Simulated network traffic	No
Limon	Linux	VM	Open-Source, Free	No	No	No
V-Sandbox	Linux	QEMU	Open-Source, Free	No	Virtual network	No
Drakvuf	Windows Linux	VM (VM Introspection)	Open-Source, Free	Yes	DNS proxy	No
Tamer	Linux	QEMU	Open-Source, Free	Yes	Closed network	No
SecBox	**Linux**	**Container**	**Open-Source, Free**	**Yes**	**No**	**Yes**

than virtualizing at a hardware level, containers virtualize at the OS level and are much more lightweight than VMs. Containers share the OS kernel [11], which means that the isolation is much weaker than when a hypervisor is used. This is one of the main reasons why containers have not been adapted for use as sandboxes, as malware can exploit kernel vulnerabilities to escape the sandbox.

Both VMs and emulators such as QEMU introduce detectable artifacts [117]. This means that malware can find out whether it is executed in a virtualized environment through various artifacts. These artifacts can generally be divided into OS artifacts and hardware artifacts. Kedrowitsch, Yao, Wang, *et al.* [118] describe that malware can tell that they are in a virtualized environment through the presence of specific files, the names of drivers and processes, or the configuration of the OS. Hardware artifacts describe indicators for a virtualized environment from the instruction execution, like increased execution time or hardware configurations found only in virtualized environments, e.g., hardware identifiers that are specific to virtual environment. Kedrowitsch, Yao, Wang, *et al.* [118] built a honeypot from containers, finding that VMs can be easily identified by hardware-level detection techniques, such as CPU clock sampling, while Linux containers possess a similar profile to "bare metal", defeating such detection techniques. Furthermore, they found that Linux containers are suitable to defeat methods that try to detect in-host monitoring tools due to LXC usage of kernel namespaces. However, the containers introduce a number of new artifacts that can be detected by malware, for example, the aforementioned name spaces or permissions.

Khalimov, Benahmed, Hussain, *et al.* [12] put together a summary of detectable artifacts and, if possible, explained how their implementation circumvented them. These artifacts are depicted in Figure 3.1. Severity describes the accuracy of detection, and complexity describes how hard their obfuscation was. If obfuscation was not possible, their complexity was considered to be high. The two most glaring artifacts are the Linux File System and the seccomp profile, both of which could not be obfuscated by them.

A further consideration that must be made is the attack surface. VMs generally have a smaller attack surface: according to [119], type II hypervisors can only be attacked through the hypervisor itself. Containers have a larger attack surface. It consists of the container engine (e.g., Docker daemon) and the host operating system [119]. Especially the latter poses a big problem. Operating systems tend to have large codebases, consequentially they are also vulnerable.

3.3 Discussion

A key takeaway from the investigation of existing sandbox solutions is that both industry and research have mostly focused on VMs and emulators for sandboxes. Containers have rarely been implemented as sandboxes due to their comparatively weak isolation. Furthermore, the need that such a solution should be both open-source and free was identified. Sharing the code publically expedites the process of finding possible weaknesses in the proposed solution. It also facilitates maintaining the solution and allows for easy adaptation, as no money needs to be paid.

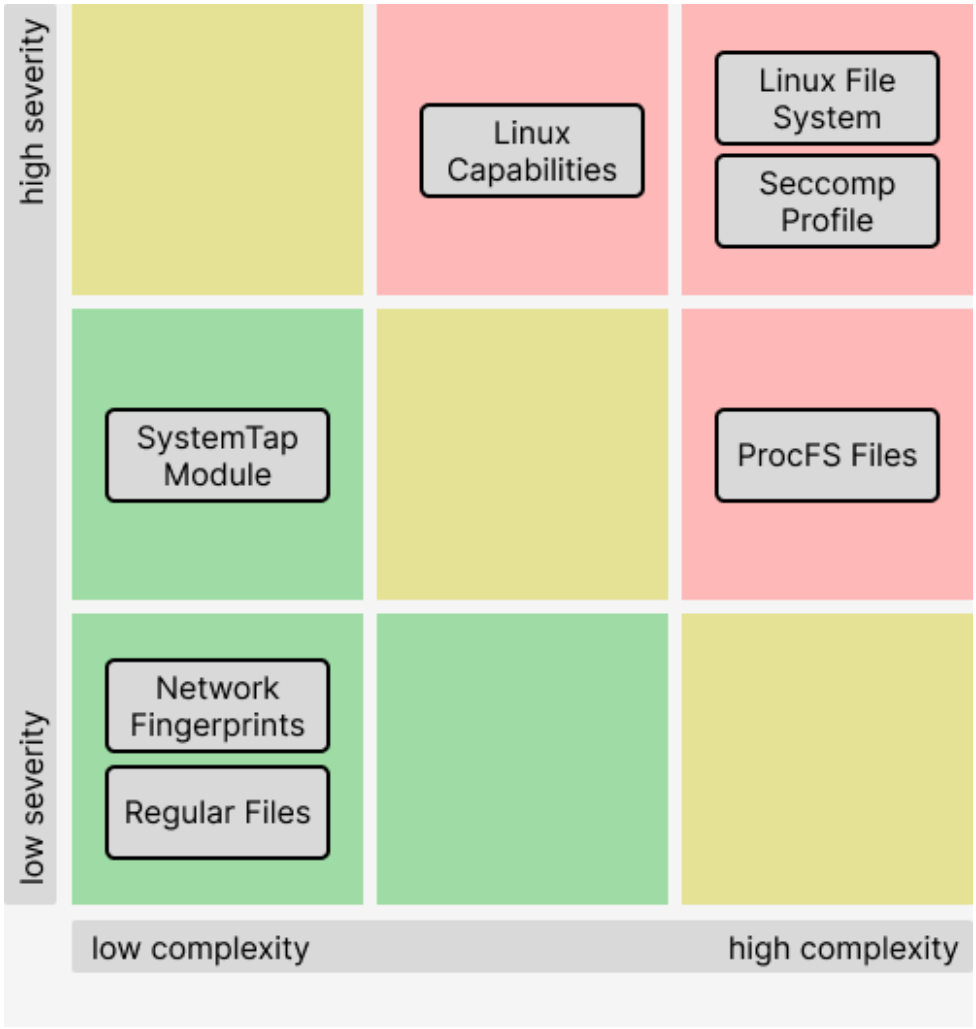


Figure 3.1: Evaluation of Artifacts [12]

To provide in-depth analysis, it should also provide be able to observe network traffic caused by the malware. The implementation should also be extensible in order to provide network routing, for secure analysis. Finally, a feature that most solutions lack is real-time interaction. This is due to the fact that most solutions focus on automated analysis, as this allows for the analysis of many samples and requires less knowledge from the user. Real-time interaction allows for greater exploration capabilities for malware analysts, as it allows them to directly see the effect of commands issued to the sandbox.

Due to the fact that the most popular container engines are Linux-based, it makes sense to also use it as a host OS. In the following chapter, the considerations that went into the design of the container-based sandbox solution are presented.

Chapter 4

Design

This section describes the design process of the proposed solution by first explaining our vision for the system and then describing identified requirements. The requirements are grouped into quality requirements and functional requirements. The following section provides insight into the choices that went into our design, most important of all the choice of containerization technology. Lastly, the final architecture is presented.

4.1 Approach

SecBox aims to incorporate concepts of various previously described sandboxes in addition to implementing ideas that are - to our knowledge - unique to *SecBox*. Before diving further into design details, a definition of the intended user(s) is needed. Users are malware analysts or security researchers, either in industry or academia. They possess a good technical understanding of the Linux platform in general and malware behavior more specifically. Additionally, they are familiar with basic networking and computer architecture concepts. The proposed system is also intended to be used within a team of such users.

Users interact via a web application with the sandbox itself. Through this web application, dynamic analysis processes can be started and ended. A unique feature of the *SecBox* approach is the fact that, for every dynamic analysis process, two separate sandboxes are started. One sandbox contains the malicious file, while the other one acts as a baseline. The former is referred to as the infected sandbox and the latter one as the healthy one. This allows users to have a comparison to filter out noise and thus identify which behavior is caused by the malware, and not by the normal running system. A further key feature is the separation of the analysis process into two phases. The first phase is called live-analysis, and the second one is post-analysis. Furthermore, the system should have an out-of-the-box approach, so the design should be able to observe malicious behavior without injecting an agent into the sandbox.

During live-analysis, users are shown an overview of the running sandbox, with various system resource consumption metrics informing the user of what is happening inside

the sandbox. This allows analysts to determine if malicious behavior has already been observed by comparing the healthy and infected sandboxes. Another important feature is the possibility to interact with the sandboxes. Users should be able to give commands to either one or both of the sandboxes allowing them to execute malware at will and modify the sandbox to fit their needs. This design decision enables users to elicit malicious behavior from a piece of malware. The sandbox should have an isolated network stack, in order to separate network traffic between hosts and sandboxes. However, this means that the sandbox can access the internet. While internet access allows for enhanced analysis capabilities (*e.g.,* of the C&C structure), the malware may use this internet access in a malicious manner. Still, this modular design allows for the implementation of network routing which may be used to limit network access.

In post-analysis, more in-depth metrics should be available to the user. These metrics are mostly displayed in graphical form. Users can then select relevant charts to add them to a report. In the generated report, users should be able to annotate charts with comments. This way, multiple users can collaborate during the post-analysis by sharing their thoughts on recorded metrics. Reports can then be saved and shown in a dashboard so that users can find their own past reports and reports created by other users. However, all visualizations have limitations. Due to this, users should also be able to download the raw data collected during live-analysis. This not only allows analysis to use already existing analysis tools but also creates the possibility of creating fingerprints from the executing malware - including a non-malicious fingerprint from the healthy sandbox. Such fingerprinting may prove valuable for future work, e.g., training a machine-learning model for automatic malware detection.

4.2 Requirements

Above all, the proposed solution should allow a user to perform dynamic malware analysis by providing a sandboxed environment to its users. Kruegel [68] specifies that a sandbox must achieve three goals: visibility, resistance to detection, and scalability. First, a sandbox has to see as much as possible of the execution of a program. Otherwise, it might miss relevant activity and consequentially users cannot make solid deductions about the presence or absence of malicious behavior. The metrics considered for the solution were:

- Network Traffic
- Memory Accesses
- System Calls
- File System Modifications
- System Resource Usage (CPU, RAM, Disk)

Second, a sandbox has to perform monitoring in a fashion that makes it difficult to detect. Otherwise, it is easy for malware to identify the presence of the sandbox and, in

Table 4.1: Specified Functional Epics

Epic	Priority	User Story	Definition of Done
Process Setup	1	As a user I want to be able to set up a Linux-based environment in order to perform dynamic malware analysis of a specified malware sample.	There exists a full-fledged dialogue for the set-up of a malware analysis process.
Execution and Live-Analysis	1	As a user I want to be able to execute a specified malware in a sandboxed environment, interact with it and see various live graphs representing the state of the sandbox.	After the analysis process has begun, a user is able to interact with the sandboxed environment in order to execute the specified malware and interact with the system. The user receives feedback about the sandboxed environment through various live graphs.
Post Analysis	2	As a user I want to be able to create and annotate a collection of insightful graphs of the data collected during the previous analysis process.	After the analysis process a user is presented with an interactive dashboard allowing them to add and delete specific graphs. The dashboard can be saved for future use.
Previous Reports	3	As a user I want to be able to view, edit and delete previous reports generated through the previously specified analysis process.	There is a history screen of previous reports available to the user.
MITRE Rule Definition	4	As a user I want to be able to view, edit and delete pattern matching rules for system calls based on the previously introduced malware components Figure 2.1	There exists a rule creation screen, allowing a user to specify a system call pattern rule. The specified rules are integrated into the existing Live and Post Analysis processes through additional graphs.

response, alter its behavior to evade detection. The notion of resistance to detection can be expressed through the number of artifacts that are present due to the containerization technology [12], *e.g.,* seccomp profiles that are set for the respective Linux container. The third goal captures the desire to run many samples through a sandbox so that the execution of one sample does not interfere with the execution of subsequent malware programs. Also, scalability means that it must be possible to analyze many samples in an automated fashion. The solution should also remain lightweight to leverage the efficiency advantage container-based solutions have over VM-based solutions. Additionally, the system should be flexible, meaning not only should the implementation be easy to expand and build on, but also should the system itself be easy to set up on new hardware.

4.2.1 Functional Requirements

Next, functional requirements are explored and specified through user stories. Due to the scope of the solution and in the interest of brevity, Table 4.1 provides an overview of the specified epics, while more fine-grained specifications are available to the interested reader in the Appendix B.1. For prioritization, a MoSCoW prioritization scheme [120] was used, with priority 1 corresponding to Must Have epics and priority 4 corresponding to Won't Have epics.

The epics process setup, execution, live-analysis, and post-analysis cover the main use cases supported by our solution. The "Previous Reports" epic expands on the main use case with additional useability functionality and allows for persistent results. The epic "MITRE Rule definition" further expands the live and post-analysis capabilities of *SecBox* by defining patterns on system calls occurring during live-analysis the user wants to be able to recognize. One example of such a pattern would be a read system call within the `/proc` directory, which could correspond to the malware trying to gather victim host information [121]. This way, malware behavior could be classified into MITRE techniques during live- and post-analysis in a way that is configurable by the user. This epic was scoped out during the design process due to the overhead required to implement it effectively and will be revisited in the future work Section 7.1.

4.3 Design Choices

During the design process of a software system, there are many design choices being made, both explicitly and implicitly. Among the most important design decisions is the choice of system structure or architecture. For the *SecBox* approach a three-component architecture was chosen in order to allow users to collaborate while allowing the solution to scale horizontally. The client and backend structure allows users to interact with *SecBox* from their browser. Deploying the sandbox on a different node on the other hand, *i.e.,* the host, allows the solution to scale out to different devices and architectures. More about the advantages of this architecture can be found in Section 4.4.

Among the design choices necessary to implement *SecBox*, the choice of containerization technology was key to the success of the system and will be explored further in the following section.

4.3.1 Containerization Technology

For the choice of containerization technology, four options were evaluated in a formal review regarding monitorability and the isolation they provide. For monitorability, only out-of-the-box solutions available for the respective containerization technology were considered. All four options allow a user to interact with a container by submitting commands, which is why interactivity was disregarded for this comparison. The options evaluated are pure LXC [79], Kata Containers [122], Docker [81], and gVisor [13]. Each of them provides a unique way how to implement lightweight virtualization. The results of this evaluation can be seen in 4.2.

Kata Containers, Docker, and gVisor all implement the OCI runtime specification [95], thus allowing a user to configure Docker to use the respective virtualization technology as a runtime. This is why the system resource usage can be accessed using the `docker stats` [91] command, as well as exposing a `veth0` device which can be monitored through `tcpdump` [123] to capture network packets and analyze network traffic. LXCs, however, do not implement this runtime specification and allow the measurement of system resource usage through their metrics API [124]. For capturing network traffic they also expose a network device that can be monitored similarly to the other solutions.

The most common approach to monitoring system calls relies on the `ptrace` [125] system call. gVisor on the other hand provides a system for monitoring system calls called trace points [126], which will be elaborated on in more detail in Section 5.1. For Kata Containers, no feasible system call monitoring solution was found. Furthermore, Kata Containers' approach to virtualization is based on allowing its users to interact with VMs in the same way as they would with containers. The underlying virtualization technology of Kata Containers is VM-based as its isolation mechanism is a hypervisor.

The isolation mechanism for Linux Containers as well as Docker is, as highlighted in Subsection 2.5.1, a set of various Linux kernel features. gVisor on the other hand, implements its own application-level kernel, offering most of the Linux system call interface. Each

Table 4.2: Comparison of Lightweight Virtualization Techniques

Technology	System Resource Usage	Packet Capture	System Calls	Container Based	Isolation Mechanism
Linux Containers	Metrics API [124]	`tcpdump` [123]	`ptrace` [125]	Yes	Linux Kernel Features
Kata Containers	`docker stats` [91]	`tcpdump` [123]	–	No	Hypervisor
Docker	`docker stats` [91]	`tcpdump` [123]	`ptrace` [125]	Yes	Linux Kernel Features
gVisor	`docker stats` [91]	`tcpdump` [123]	Trace Points[126]	Yes	Application-Level Kernel

isolation mechanism implies a different approach to ensure a hardened container. While gVsior containers are already in a hardened state, Linux and Docker containers require additional configuration, effort, or external tools to achieve hardening.

This evaluation lead to the decision to use gVisor to implement *SecBox*, due to its ease of use by substituting the Docker runtime, the perceived superior isolation provided by its application kernel, and the above-elaborated monitoring capabilities.

4.4 Design of Architecture

The implementation should combine all the above-described factors into one architecture. To achieve this, the system was split into three distinct components:

(1) **Client** - Frontend that is responsible for user interaction and data visualization.

(2) **Server** - Backend that processes data from host and handles user requests.

(3) **Host** - Controls sandboxes and extracts data.

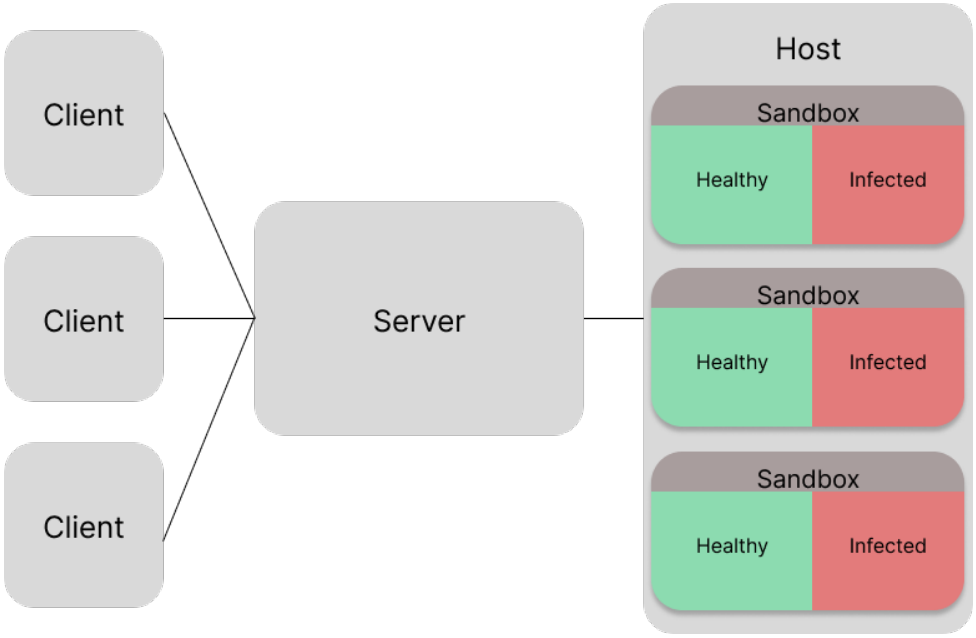


Figure 4.1: High-level Design of Architecture

The key idea behind this architecture is to split up data interaction/visualization (client), data processing (server), and data collection (host) as can be seen in Figure 4.1. This architecture allows multiple clients to connect to the system in parallel, so that collaboration between users is possible. More interesting is the isolation of the host. This was inspired by Cuckoo's [8] architecture and has the advantage that the sandbox containers can be isolated (either virtually or physically) from the backend. This makes it especially interesting as it is possible to deploy a host on a different architecture, e.g., running a host on an IoT device to target IoT malware. Note that the host handles multiple sandboxes. A client can also start more than one sandbox. A single sandbox always has two container instances. While these containers are isolated from each other and thus would each fit the definition of a sandbox, the term sandbox refers to a pair of secure execution environments. The final component needed for the implementation is a database to save past results.

With this high-level design as a basis, a more granular and technical architecture is developed and described in the following chapter.

Chapter 5

Implementation

This chapter explains the technical details of the prototypical implementation of *SecBox*. The implementation adheres to the design vision explained in the previous chapter.

The detailed view of the system and its three high-level components host, backend, and frontend are depicted in Figure 5.1. Each component can be deployed on a different device, as they communicate via WebSockets. The database can also either be deployed locally (on the backend server) or remotely (in the cloud). The following sections will go into specifics regarding the implementation of each of the high-level components.

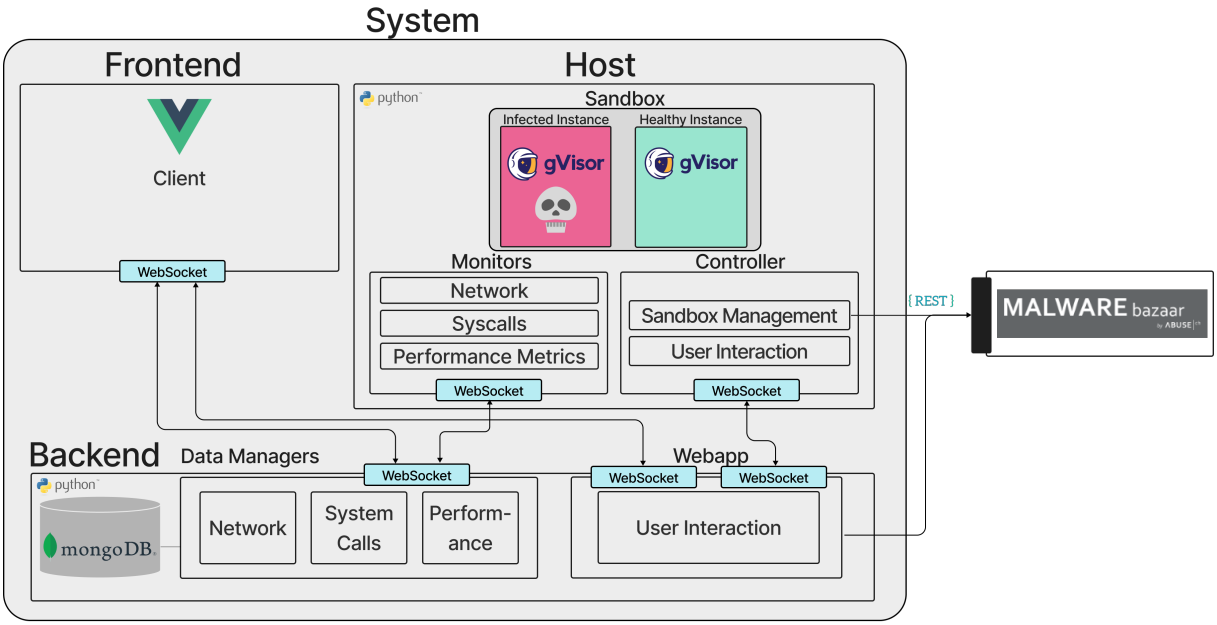


Figure 5.1: The Final *SecBox* Architecture

5.1 Host

The host is the core piece of this solution as it contains the critical components for malware isolation and data collection. It is implemented using Python and communicates with the other components using *Socket.IO* [127], which implements Socket.IO clients and servers which we refer to as WebSockets. The host has three sub-components each of which has a single responsibility. The controller enables user interaction with the sandbox, the sandbox itself executes the malware in isolation and the monitors extract the data from the sandbox.

The controller uses the docker SDK [128] to start, stop and interact with multiple running gVisor containers. This is possible since the gVisor runtime `runsc` implements the OCI runtime specification [95]. gVisor is an application kernel that provides additional separation between the applications running and the host operating system. It implements a large part of the Linux system call interface both for the ARM64 and the x86-64 architecture to do this. gVisor and its runtime are set up using the provided setup instructions [129]. The runtime is reconfigured for both healthy and infected sandbox instances in order to achieve high separation between instances. It also enables the transmission of data to a separate Unix domain socket that enables monitoring of system calls and container events for each sandbox instance. Malware injection happens via its Dockerfile [130] when an infected sandbox instance starts up. Malware samples are directly pulled from Malware Bazaar [131] through their provided API. Malware Bazaar is a curated collection of malware samples operated by `abuse.ch` [132] that provides various IT security-related services free of charge.

The monitors each focus on a set of data to extract from the sandbox and send data to the respective data manager within the backend. Each monitor is responsible for two sandbox instances. It spawns one process for each instance, which continuously and independently performs its function. Data extraction is done using various means. The performance metrics are extracted using the docker stats API at `/containers/(id)/stats` that every docker container exposes. This interface provides detailed information about every single docker container. The data entails networking data such as sent/received bytes and detailed memory usage data such as cache usage, resident set size (RSS), and the number of page faults. Additionally, CPU usage data in nanoseconds CPU time along with other useful CPU-related metrics are provided. This data is used to calculate how much CPU is being used at a given point in time. Here, it is noteworthy that the CPU time is based on the total time used across cores, which leads to the percentage value referring to the percentage of CPU time that was used by a specific container. Thus, a value of 700% refers to seven CPU cores using their entire capacity. Performance data is provided in regular intervals, which results in a predictable and stable load on the WebSocket connection. The running process continuously sends the extracted data through the established WebSocket connection to the backend.

To collect networking metrics and capture packets *SecBox* uses `scapy` [133] to monitor the `veth0` device representing each running container. This `veth0` device represents the bridge network created by docker networking. Scapy is able to sniff packages from various interfaces. Additionally, it provides functionality to capture, dissect and analyze packets,

similar to how `tcpdump` [123] works. Unlike tcpdump however, it provides a Python implementation, allowing for seamless integration into the host. It also provides various export formats, `.pcap` files or string representations, which are used for the export and transmission of captured packets. In order to offset the spikes in network traffic that activities by the malware and the user (such as installing new packages, and launching packet flooding attacks) cause, captured packets are batched. Batches are either sent based on their size if it exceeds 1000 Bytes or on a time basis if the last send occurred more than 1s ago.

System call monitoring uses a monitoring process as described in the gVisor documentation [126] by exposing a Unix domain socket (UDS) to which the sandbox can send container events and system call data. The monitoring process and the sandbox use a UDS of type `SOCK_SEQPACKET` to communicate. *SecBox* implements it own UDS socket server written in Python, despite existing solutions provided in the gVisor code base written in Go and C. This is to remove the dependency and overhead introduced by bazel [134] that is used to build the solutions provided by Google. The desired subset of system calls to be monitored by the monitoring process can be configured through a `session.json` file within the host. Here, the container events, such as container start and stop, and Linux system calls that should be monitored during run time can be specified. Within gVisor, each of these events that are listened to is referred to as `point`. Additional context information for each system call is also available in the form of `context_fields`. An exemplary specification of a session can be found in Listing 5.1.

Listing 5.1: Session.json Example

```
{
  "trace_session": {
    "name": "Default",
    "points": [
      {
        "name": "container/start"
      },
      {
        "name": "syscall/sysno/0/enter",
        "context_fields": [
          "time",
          "container_id",
          "thread_id",
          "credentials",
          "cwd"
        ]
      },
      {
        "name": "syscall/sysno/1/enter",
        "context_fields": [
          "time",
          "container_id",
          "thread_id",
          "credentials",
```

```
            "cwd"
          ]
        }
      ],
      "sinks": [
        {
          "name": "remote",
          "config": {
            "endpoint": "/tmp/infected.sock",
            "retries": 3
          },
          "ignore_setup_error": true
        }
      ]
    }
}
```

Since system calls occur at an extremely high frequency, monitoring them introduces significant performance overhead. This makes continuous transmission during live-analysis using web sockets unstable. In order to improve performance and decrease the load on the WebSocket, the collected system calls are transferred using a `POST` request in batches during and finally at the end of the live-analysis process. This allows for continuous transmission of user-issued commands during live-analysis, however, it introduces small wait times used for system call processing once the user enters post-analysis.

5.2 Backend

The backend serves as the critical piece to enable the data flow between the host and the client. The backend was implemented as a Flask web application. While this is not the most lightweight option, it is easy to implement and has client interfaces with many architectures, including IoT devices. It communicates (mostly) via WebSockets (implemented via flask-socketio [135]) with the frontend and host, which allows for fast real-time data transfer while still being scalable. The biggest advantage of web sockets for this system is their server-initiated transmission, as it enables fast communication between the client, backend, and host without the need for polling.

Once an analysis process has been started, the backend assigns a universally unique identified (UUID v4) to the process. This UUID is used to create a one-to-many mapping between a sandbox and the involved clients. As soon as the host – more precisely, a monitor – sends data to the backend, the corresponding data manager in the backend can process the data. Once the data has been processed, *i.e.,* aggregated and turned into the right format, it is sent to the clients that are in the analysis process with the corresponding UUID. On a conceptual level, each monitor in the host has a corresponding data manager. There is a performance, a system call, and a network manager. This separation allows for modular extensions for more monitors and managers. However, a single manager is

able to handle and process data from multiple hosts and thus monitors. This ensures that multiple hosts can be deployed on various devices with differing architectures (*e.g.,* ARM64 and x86-64) attached to a single backend to make *SecBox* scale horizontally. To support this feature, system calls coming from the host are translated according to the host architecture. These data managers not only handle the transfer and processing of data from the host to the client but also ensure that the data is saved to the database once a process is finished.

The backend also covers user interaction. This includes the start-up and shutdown of sandboxes, as well as the handling of the sending of command prompts from the user to the sandbox and receiving the feedback. This also includes the creation and updating of reports. Effectively, this controls the flow of analysis processes. Starting up a sandbox initializes the live-analysis, and shutting it down begins the post-analysis. This then allows for the creation of a report which can be updated with comments. Further details of this workflow are covered in Section 5.3.

The database enables the saving of previous analysis results, raw data, and information on host capabilities. The host capabilities include a sample set of malware, available Linux images, and the host architecture. Information about malware samples is automatically collected during start-up from Malware Bazaar [131] depending on the chosen host architecture (either 32- or 64-bit architecture) as well as the malware families already present in the database. Thus new strains of Linux Malware are automatically fetched from Malware Bazaar when starting the backend. The current implementation goes through the most recently submitted malware samples and collects one sample per malware type and target architecture. Only malware that is executable in the sandbox is collected, *i.e.,* bash scripts and `.ELF` files for the respective architecture. Note that not the actual malware is downloaded to the backend, but rather their meta information. This way, no malicious code is saved in the backend, but only references to the malware samples in Malware Bazaar. The implementation uses MongoDB [136] which is a document-oriented database. The database is designed to either be hosted on the backend itself or through MongoDB's cloud-native deployment.

5.3 Frontend

The frontend is built with Vue 3 [137] as a framework. It is made up of several different pages, which allows the user to navigate through the analysis process: *home page* to welcome the user, a *live-analysis page* where users can monitor interactive with running sandboxes, a *post-analysis page* where users can pick relevant charts, a *report page* where users can annotate the selected charts, and finally a *report dashboard* where past analysis processes are listed. In the following subsections, the pages and their most important functionalities are described in more detail.

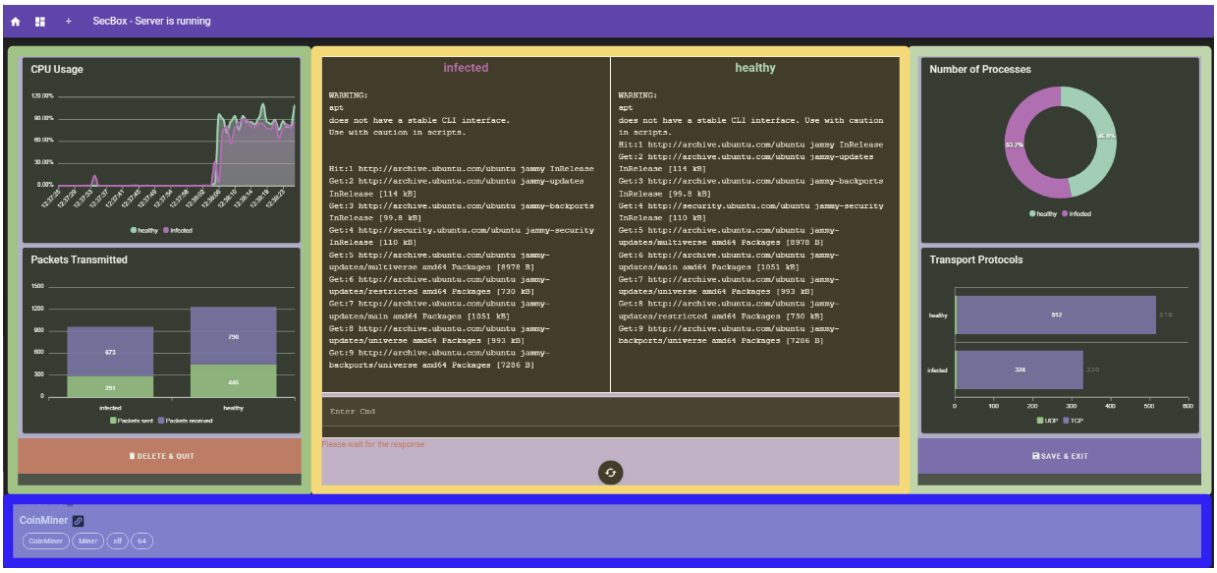


Figure 5.2: Live-Analysis Page

5.3.1 Home Page

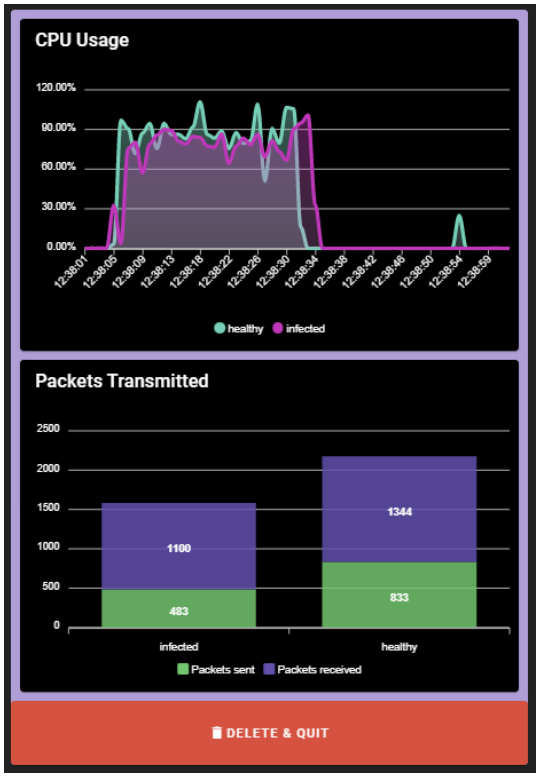
The home page describes the general idea of the project. It also displays some recent reports to the user. Users can start an analysis process by selecting a malware sample and a Linux image from two drop-down lists. In the current implementation, malware samples that are collected from Malware Bazaar and several different Ubuntu images are available.

5.3.2 Live-Analysis Page

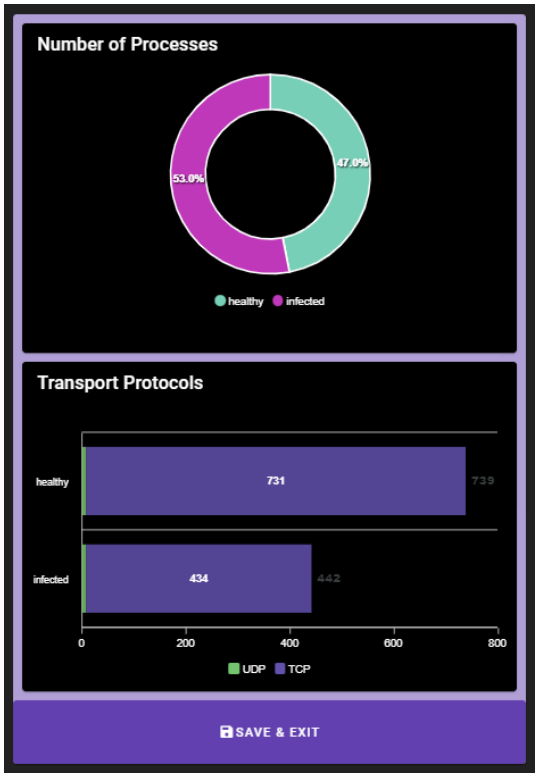
After starting a new analysis process, the user is taken to the live-analysis page. This page has three components as can be seen in Figure 5.2: The CLI allows the user to interact with the started instances (yellow). the performance charts display real-time data from the two instances (green), and the malware information box displays information about the malware sample being analyzed (blue). A more detailed view of the performance charts can be found in Figure 5.3.

The CLI is placed in the middle of the screen due to its importance. Through the CLI, users can send commands to either of the instances or both of them at the same time. It works analogously to a bash shell. It enables users to interact with the sandbox, either to customize the environment (*e.g.,* by installing packages or programs) or elicit malicious behavior. The CLI also displays the command line output from each instance, allowing the user to check whether the execution of the malware sample or installation of a program was successful. To assure that commands are executed in the right order, users can only send commands to an instance after the previously sent command has been executed.

Four charts display live information about the execution behavior. The goal of these charts is to give the user deeper insights into what is happening in their instances. The metrics



(a) CPU and Packets Transmitted



(b) Number of Processes and Transport-layer Segment Count

Figure 5.3: Visualizations of Live-Analysis Page

that are displayed are mostly performance related, as the user most likely is interested to see how the malware is affecting the sandbox's behavior. These charts were implemented through ApexCharts [138] which is a charting library compatible with Vue 3. A key feature is how simple it is to create and update a chart with real-time data, which makes it well-suited for the live-analysis screen.

The first graph, shown in Subfigure 5.3a, displays the **CPU percentage** used by each instance. The frontend only receives data from the last 60 seconds and renders it as two overlapping area charts. It should be noted that real-time rendering of data from two separate instances poses a challenge for synchronicity. This is due to the fact, that data is not retrieved from these at the same intervals. One possibility would be to aggregate the data and average it over timestamps. However, this approach has the downside of introducing a delay. Instead, the data is sent asynchronously, which minimizes the delay, but may lead to inaccurate time stamps, as there is only one x-axis. Hovering over a line shows the user the exact percentage of CPU usage at that time. This value is based on a single virtual core, so it can be larger than 100% if multiple cores are used.

The second graph in Subfigure 5.3a displays the **packages transmitted** by each instance since their start up. To display this information, a simple stacked column chart shows the packages that are sent and received by each instance. This view allows users to gain an overview of the aggregated number of packages and shows how much the network has been used by each instance.

Subfigure 5.3b shows the third chart. It renders a donut chart of **the number of processes**. This data represents an aggregated view of the processes which ran in each instance since start up. This number is calculated by counting the number of process IDs in each instance. This gives users some insight into the process level of each instance. The numbers are displayed as percentages of all processes run in both instances. Hovering over the chart gives the exact number of processes that ran so far.

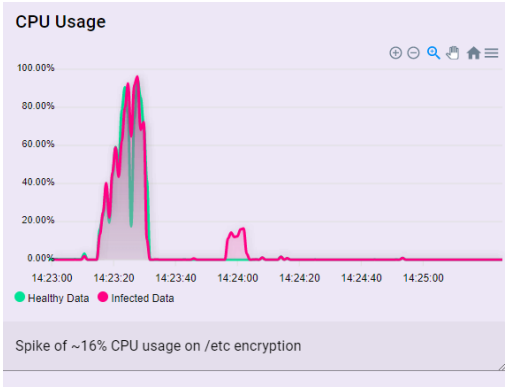
Finally, the last chart is the **transport-layer segment count** displayed in Subfigure 5.3b. This view contains - analogous to the packet transmitted chart - the total number of packages sent and received by each instance. A notable difference is that now they are mapped according to their transport layer. Note that only UDP or TCP are considered.

After the user is done with their analysis, they can terminate the sandbox at will. This allows the analysis of malware that employs timing as an evasion technique, as the analysis can be run until malicious behavior appears.

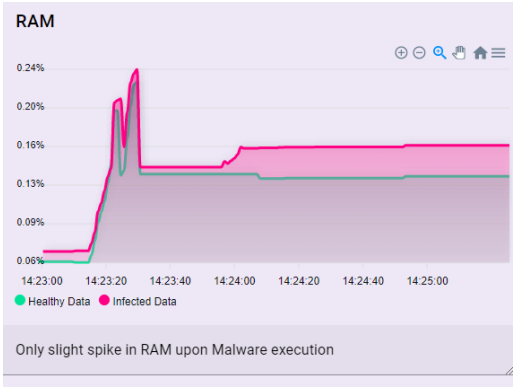
5.3.3 Post-analysis Page

Once the live-analysis process is terminated, *i.e.,* the sandbox is shut down, the user gets taken to the post-analysis screen where they can select which charts will be displayed in the report. This allows for visual analysis of the data that has been collected through six charts. While some charts are similar to the ones on the live-analysis page, here the focus lies on exploratory data analysis rather than real-time information visualization. All data is aggregated and processed before being rendered. In this subsection, the available charts are explained. Most graphic representations were again implemented via ApexCharts [138] with exceptions being explicitly mentioned.

Figure 5.4: Performance Charts in the Report Page.



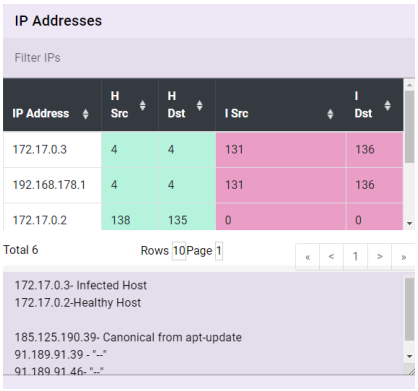
(a) CPU Percentage Chart



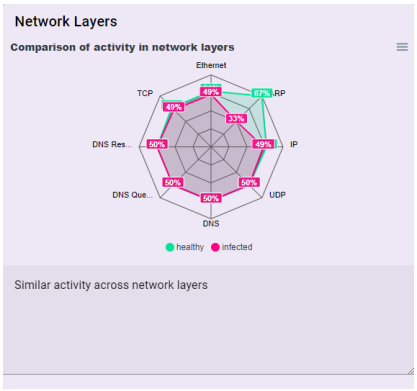
(b) RAM Chart

The **CPU percentage** chart displayed in Subfigure 5.4a is nearly the same chart from the live-analysis screen but over the whole analysis period. Unlike in the previous implementation, synchronicity is no longer an issue, as all data has already been processed. A noteworthy feature is its interactivity. By zooming in and out, users can focus on a relevant time frame. This is especially helpful as some actions require a lot of CPU usage

Figure 5.5: The Two Network Charts in the Report Page



(a) Filterable IP Table



(b) Network Layer Segments

in a short amount of time, which makes it more challenging to see variation during lower usage phases.

The **RAM usage** chart depicted in Subfigure 5.4b is nearly identical to the above-described CPU percentage one. It also displays the RAM usage as an area chart for both instances. The chart also contains data as a time series over the entire analysis period and can also be interacted with. This metric was chosen to also represent memory usage.

To give further analysis capabilities for the network, a further chart is the **network layers** one. It displays network data as a spider chart, as can be seen in Subfigure 5.5b. It compares packages from the infected instance to those of the healthy one. The comparison includes different layers (*e.g.,* IP, UDP, TCP, Ethernet) and is relative. This means that for each axis, both values add up to 100%.

To cover a second dimension of network traffic, an **IP address** table is available. This table displays the number of packages sent and received according to their IP addresses. An example can be seen in Subfigure 5.5a The rows contain the IP addresses, and the columns describe whether it was sent or received by either the infected or healthy instance. The table is also searchable by IP addresses. This table was implemented through the vue3-table-lite package [139].

Low-level information is also collected during the analysis process. The **read write** chart displays read and write system calls as stacked bar plots. This allows the user to compare the total number of read or write operations by each instance. An example can be found in Figure 5.7.

Further information is displayed through the **system calls directory** chart shown in Figure 5.6. It contains a sunburst chart for each instance and displays in which directory the collected system calls were executed. The top-level directory is excluded from this, as it makes the visualization more cluttered. The number of top-level system calls is displayed in the center by hovering over the directory name. The chart consists of multiple rings displaying nested directories. Segments in the same ring are on the hierarchical level. The size of each segment is determined by the number of system calls that were executed in one of its child directories. When hovering over the segments, the exact count is displayed.

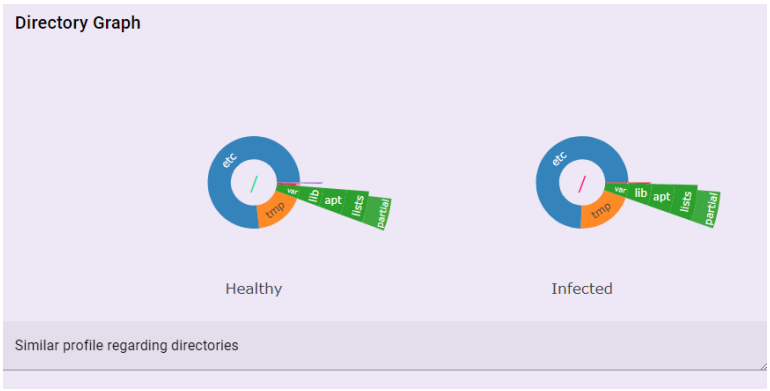


Figure 5.6: Directory chart

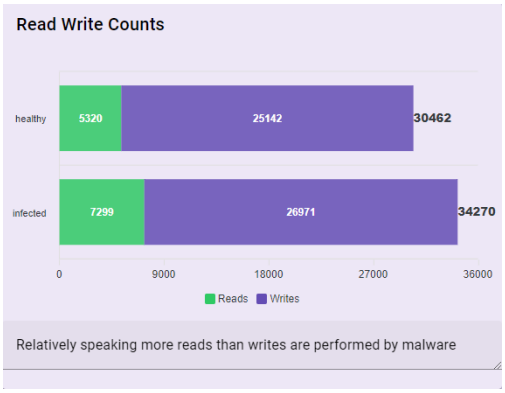


Figure 5.7: Read Write Chart

Clicking on a segment enlarges it and rescales the plot so that only subdirectories are visible. To keep it well-arranged, each instance has a separate view which can be toggled through a button click, a side-to-side comparison of this chart is available when the report has been created. This chart was rendered with Plotly [140].

The charts can be selected by clicking on the respective button in the sidebar. These buttons are grouped according to their domain (*i.e.,* performance, network, and system calls). Once the button is pressed, the chart is rendered. After all charts of interest have been selected, the user can create a report.

5.3.4 Report Page

This page displays the previously selected graphs with a comment field, where users can describe their observations. Examples of each graph with its comment field can be found in Figures 5.4, 5.5, 5.6, and 5.7. Through these comments, knowledge of the analysis process can be shared. In addition, there is the option to download the raw data collected during the analysis. Currently, system calls and a PCAP file from the network activity are available. This allows for downstream processing of the data, *e.g.,* processing the PCAP file with tools like Wireshark [141]. The system calls are saved in a CSV file with the features *timestamp, thread id, sysname, container id, cwd, credentials,* and *args.* The

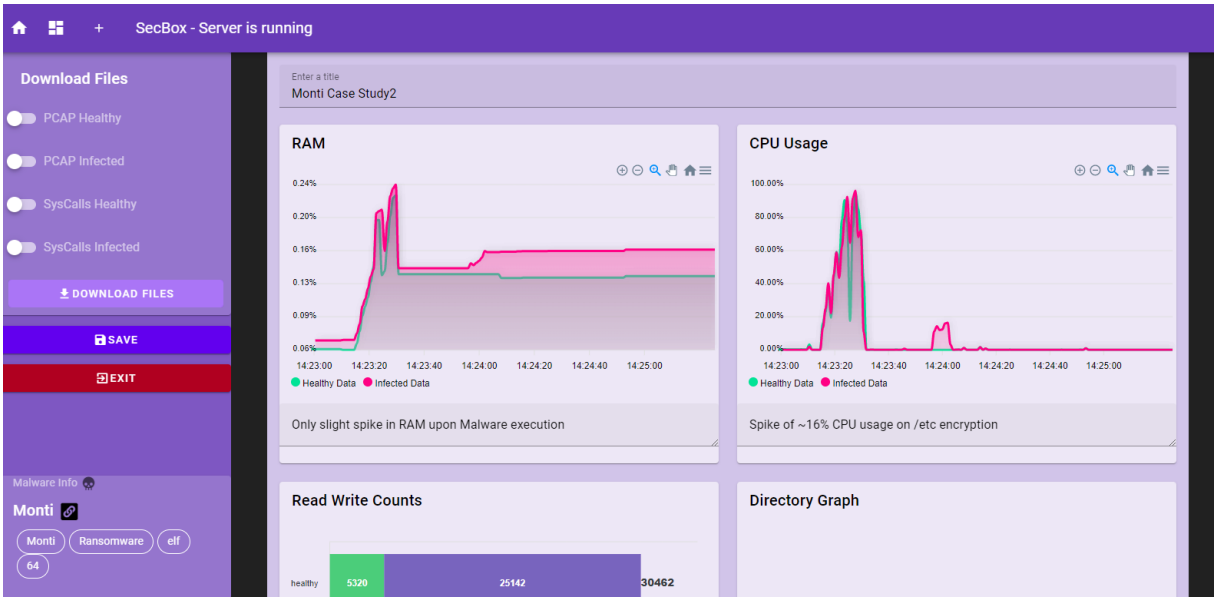


Figure 5.8: Report Page

timestamp explains when a system call occurred and the thread id from which thread it stems. The sysname column contains the name of the system call that was made and cwd in which working directory it was executed. The container and thread IDs refer to the unique identifier a container and a thread have within an OS. Credentials refer to the credentials used for executing the system call, while args is a list of up to six arguments passed to the respective system call. There are four downloadable files, two for each instance. From these files, the user can select which ones they want to download. The malware information and a link to the analyzed sample are also displayed. Users can also give the report a title which simplifies finding it again. By saving the report, the analysis process is finished. All reports are saved to the database and shown with their title and malware sample in the report dashboard.

5.4 Deployment

The deployment can easily be done locally. If additional security is desired, it is recommended to deploy it inside a VM. For the evaluation described in the following chapter, both the backend and host were deployed inside an Ubuntu VM. Due to the fact that the sandboxes use their own Docker runtime, it is not possible to deploy the whole application as a Docker image. One option is to deploy the backend as a reverse proxy for the flask application using an NGINX web server. A detailed tutorial can be found in [142]. The WebSockets also need to be considered when setting up the reverse proxy, instructions can be found in the flask-socketio documentation[143]. The VM used Ubuntu 22.04.1 with 4 CPU cores, 16 GB of memory, and 50 GB of storage. While one might argue that deploying *SecBox* to a VM defeats the purpose of creating a container-based solution, the advantage of reducing overhead still remains. This is due to the fact that only one VM instance is needed to analyze multiple samples which are isolated from each other, and this with very little overhead. VM sandboxes have the overhead of an entire OS for

each instance. Furthermore, stacking layers like this increase the isolation and greatly decreases the chance of a critical breach.

Chapter 6

Evaluation

To evaluate the functionality provided by the developed tool, three case studies were conducted, to analyze specific malware samples in different scenarios and with different goals. Each case study is subsequently evaluated on six quality metrics.

6.1 Case Studies

Three scenarios are proposed in which *SecBox* can be used for malware analysis by security researchers. The first case study focuses on the Mirai malware which was showcased in Subsection 2.2.2. The goal of the user is to analyze a specific sample of Mirai, in order to gain insights into its C&C architecture and block potentially malicious IPs in a productive environment. Additionally, useful raw data for further processing and analysis is collected in order to potentially automate this process in the future. In the second case study, a malware comparison is conducted between a number of crypto-jacking samples. This case study highlights the ability to look at multiple samples of the same type and how these differences are captured by the proposed solution. Lastly, the third case study evaluates the sandbox and its capability to extract malware behavior under load. For this purpose, another sample of Mirai is executed alongside a load-testing tool.

Finally, the three case studies are jointly evaluated along the following dimensions:

- **User Experience**: How well the user is able to use the tool to fulfill their specific task and how well collaboration is supported.
- **Low Visibility**: Whether the malware sample is able to notice it is executed within a sandboxed environment.
- **Monitorability**: How well the behavior of the malware is visible with *SecBox*'s out-of-the-box approach.
- **Degree of Automation**: How many interactions are required from the user to execute a sample.

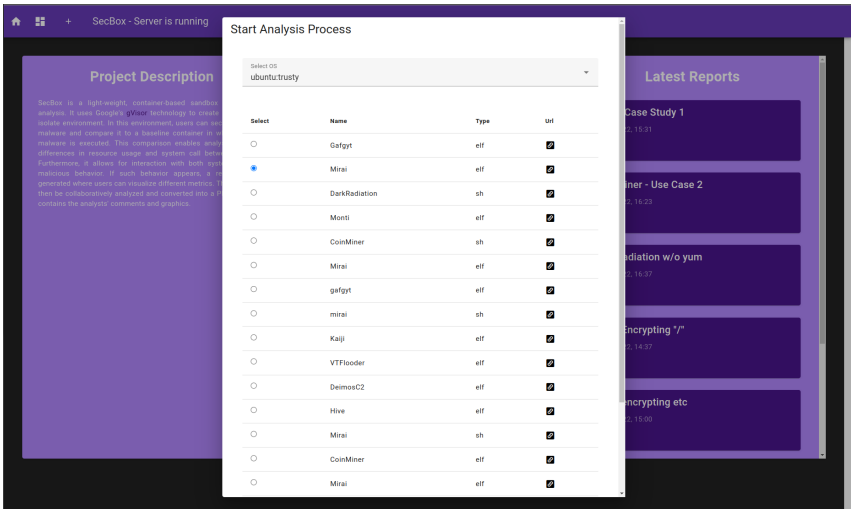


Figure 6.1: Mirai Configuration

- **Performance**: The system resource requirements of the suggested solution.
- **Isolation**: Whether the executed piece of malware is able to escape from the system boundaries.

The data collected in *SecBox* is abstracted into findings by using information gathered through open-source intelligence (OSINT).

6.1.1 Mirai Analysis

For the analysis of the Mirai family, a malware sample from Malware Bazaar was selected based on the file format (Linux executable files like `.ELF` or `.sh`) and the architecture of the target system's CPU. The specific sample is referred to as Mirai A and the Malware Bazaar link can be found in Appendix B.2. Since Mirai prominently targets IoT devices, many available samples are 32-bit, however, the available instance of the *SecBox* host is deployed on a 64-bit system. As an OS, the Trusty Ubuntu [144] container image was selected since it comes with many useful packages already installed, such as `iputils` [145], `cron` or `busybox` [146]. Especially `busybox` is critical for this sample of Mirai, since it employs several of its functions during operation. If executed within an environment without these utility functions, its proper behavior will not be exhibited.

The user connects to *SecBox* and configures their desired analysis process, as can be seen in Figure 6.1. Subsequently, the sandbox is started, and the user enters the live-analysis screen. Using the CLI, the user explores the provided environment and updates the `apt` package lists. In the starting directory, the malware sample is identifiable by its SHA256 hash. In order to make it executable, the `chmod` command is used. Execution of the `.ELF` file is done using bash.

The malware sample executes with no issues, resulting in an increase in the number of processes in the infected sandbox, as well as a small spike in CPU usage. Additionally,

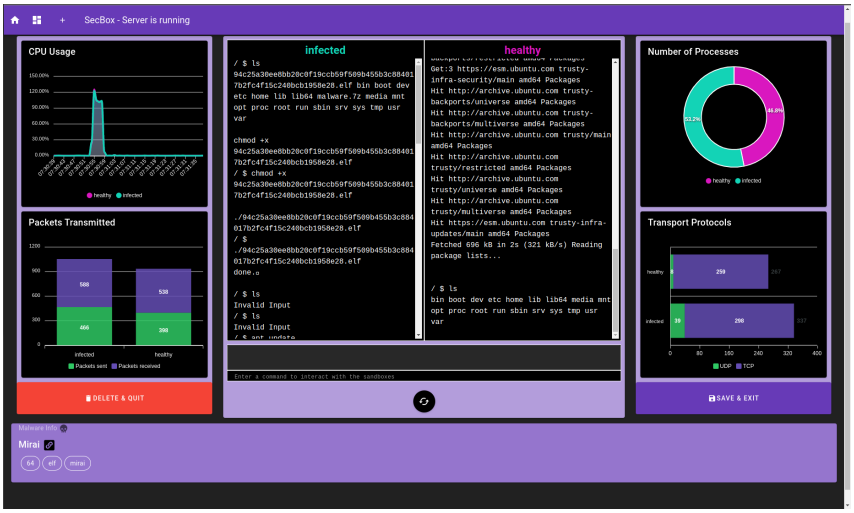


Figure 6.2: Mirai Live-Analysis

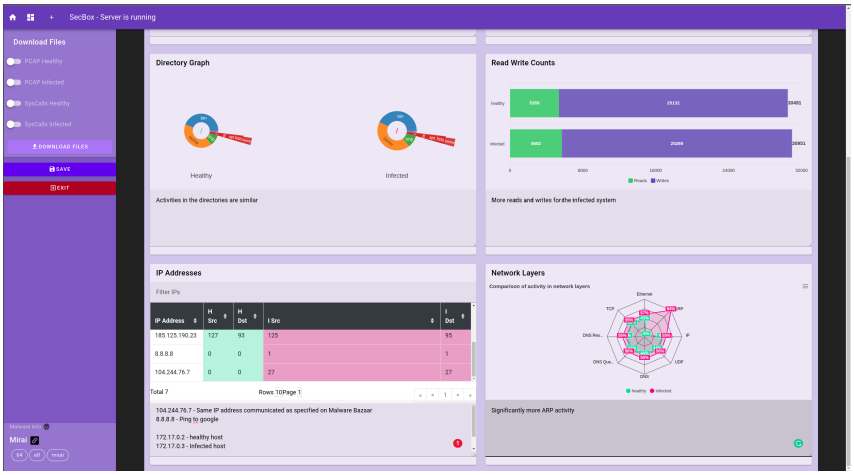


Figure 6.3: Mirai Generated Report

`done.`□ is printed to the command line of the infected instance. After execution, the `ls` command results in `Invalid Input` in the infected sandbox. Using the `dir` command, the user is able to discern that the sample of Mirai obfuscated its own filename to be `bin\220[\216\214\355\177`. Note that this is only the visual representation provided by the command line, which was transformed into a string. It contains unusual characters that make analysis (*e.g.,* using the objdump command [147]) harder [148].

At regular intervals after execution, CPU utilization spikes in the infected sandbox, and an increase in packages sent and received are observed. These packages use the UDP and TCP protocols. An excerpt of this behavior can be seen in Figure 6.2. The characteristic behavior of the Mirai malware to launch a DoS attack on an intended target could not be observed while running it for approximately eight minutes. There are various explanations for this: Either the environment presented was unsuitable for the malware, the intended behavior specifies a longer sleep duration or the sample remains dormant until a coordinated attack is triggered by its C&C server. After eight minutes of execution, the user terminates malware execution and enters the post-analysis screen. All six graphs

are selected for the report. An excerpt from the generated report can be seen in Figure 6.3. The directory activity graphs of both instances look very similar, and within the subdirectories, the system calls observed are virtually identical. However, within the `"/"` directory, where the malware was executed, ∼2400 more system calls happened. The bar chart of read and write operations of the instances is in line with this observation, with the infected instance having ∼300 more reads and ∼130 more writes.

Since Mirai's primary goal is to launch (D)DoS attacks on a target, the networking-related graphs show the biggest discrepancies between the two instances. Across all network layers, the activity is higher for the infected instance, with the biggest difference being the ARP layer. Here, the infected sandbox is responsible for 94% of the generated traffic. Mirai is known to perform ARP scanning (see Subsection 2.2.2), and this behavior is reflected in the graphics generated by *SecBox*. This sample of Mirai communicated heavily with one specific IP address, `104.244.76.7`, which is also mentioned on the Malware Bazaar site of Mirai A to be the distributor of the sample, making it a prime candidate for being the sample's C&C server. Using OSINT tools, such as IPGeolocation [149], a user is able to find out that the IP belongs to a server in Luxembourg hosted by a company named FranTech Solutions. Thus, it is possible to confirm the unvalidated statement made in the threat report using *SecBox*. Additionally, the sample sent a single packet to Google's DNS server with the target IPv4 address 8.8.8.8. The information and insights gathered by the user for each graph are noted down in the text field below each graph, allowing the collaborators to quickly build on the insights discovered by our user. In the case of the IP table, a comprehensive explanation of each IP is provided for future use.

From the report page, the user can also download the captured network data as `.pcap` files and continue a more thorough analysis of the network traffic in a malware-specific tool like SecGrid [150] or a more general-purpose tool like Wireshark [141]. The executed system calls are available for download as well as `.csv` files. Such files may be further processed and used in AI-based malware classification approaches such as highlighted by Kolosnjaji, Zarras, Webster, *et al.* [151].

Therefore, as shown in this case study, *SecBox* enables the user to investigate the behavior of a previously uninvestigated sample of Mirai malware and observe its behavior in a given environment.

6.1.2 CoinMiner Comparison

In this case study, multiple samples of crypto miner malware from the CoinMiner family are analyzed. To gather these samples, a search is conducted on Malware Bazaar [131] where all samples were collected that have the *CoinMiner* signature. From the results, the search is narrowed down to samples that are `.ELF` files. In total, seven samples match these two criteria. The links to these samples can be found in the appendix table B.2. Just as in the previous case study, the host is deployed on a system with a 64-bit architecture. As an OS image, Ubuntu 23.04 is used. Note that, in contrast to the automatic sample collection provided by *SecBox*, binaries that are not tagged with their architecture were also considered in this case study. In fact, only one of those samples is

tagged to be compatible with a 64-bit CPU architecture. Still, simply because samples are not properly tagged in Malware Bazaar, does not mean that they are not suited for analysis. The motivation behind this case study is to see how well-suited *SecBox* is for analyzing and comparing multiple samples. This task is useful, as it allows analysts to gain deeper insights into the behavior of related samples and analyze differences in their behavior.

The seven samples are analyzed in succession. As the first command, a check is performed on whether the file is even executable on the host architecture. Executing the command `od -An -t x1 -j 4 -N 1 [binary name]` prints a 01 if the binary is 32-bit, and a 02 if the binary is 64-bit. In the next step, the binary is moved to the `home` directory with the command `mv`. This simplifies the analysis of system calls in the directory chart of the post-analysis. The binary is then made executable using the `chmod` command and then executed with bash. Other commands were also executed to elicit malicious behavior. If such malicious behavior occurred, the analysis was stopped shortly after. To differentiate the samples, labels have been given to each distinct sample. They are labeled with CoinMiner A to G in no particular order.

The first sample, CoinMiner A, fails during execution. The output of the execution is depicted in Listing 6.1. This output alone shows that there is malicious behavior in this sample. The malware sample tries to open a config file that is required by the Monero miner it tries to use. The miner is Xmrig [152] which works both with CPU and GPU. The miner first tries to access `config.json` in the directory where it is executed and then moves on to the root level searching different locations for the config file. However, since no such file exists, the malware is not successful at installing the miner. One approach to deeper analyze the miner would be to create a config file that can be used by the miner. However, a generic configuration does not deliver much deeper insights. This is due to the fact that a generic configuration does not include some of the most critical information for malware analysis, such as a Monero wallet address to which the mined coins should be sent. It seems like either the malware sample that was uploaded is incomplete (*e.g.,* that it was delivered with a config file but only the binary was uploaded to MalwareBazaar) or that the malware author wrote erroneous code. Still, it allows us to conclude that CoinMiner A tries to install an XMrig [152] miner (which itself is not malicious) on the target's system to hijack the targeted system's resources for crypto mining.

Listing 6.1: Output from CoinMiner A

```
[2023-01-23 16:43:06.752] unable to open "/home/config.json".

[2023-01-23 16:43:06.752] unable to open "/root/.xmrig.json".

[2023-01-23 16:43:06.753] unable to open
"/root/.config/xmrig.json".

[2023-01-23 16:43:06.753] no valid configuration found,
try https://xmrig.com/wizard
```

CoinMiner B is a 32-bit binary, and as such, can not be analyzed on this host.

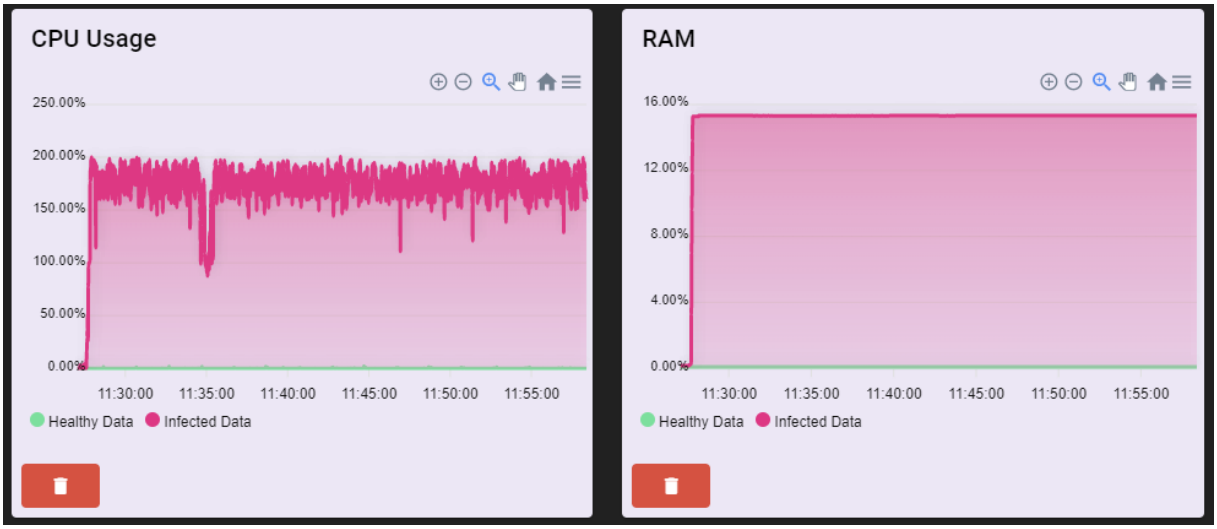


Figure 6.4: CPU and RAM usage of CoinMiner C

CoinMiner C executes successfully in the behavior. It behaves in a manner that is expected from cryptojacking malware. Both CPU and RAM usage are on a consistently high level as can be seen in Figure 6.4. The workload is between 160 and 200% of a core and the used RAM is 15.32% of the total capacity. This is due to the fact, that crypto mining is a CPU-, and RAM-intensive operation. Interestingly, the output after execution prints the statement `sh: 1: [: =: unexpected operator` four times, indicating that the malware sample uses a bash script during execution. Despite this error, the sample ran for 30 minutes and the process was stopped once suspicious network traffic was discovered. Traffic with the IP address `167.114.114.143` is captured. According to IPGeolocation [149] this address is located in Montreal, Canada. 369 packages were sent to this address and 245 were received by it. Only the infected instance communicated with that server, indicating the possibility of it being a part of the C&C structure. The system calls were mostly executed in the `/home` and `/` directories. Interestingly, further investigation of the downloaded system call file shows that three calls were executed in `/root`. One of which is the `exit_group` call, a system call that exits all threads in a process.

The fourth sample, CoinMiner D, also results in an error during execution. The error log is depicted in Listing 6.2. Reports from other sandbox tools, notably on JoeSandbox [153], show the exact same error message, indicating that this - just as with CoinMiner A - is an issue with the malware, rather than with *SecBox*.

Listing 6.2: Output from CoinMiner A

```
panic: runtime error: invalid memory address or nil pointer
dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x18
pc=0x707cf6]

goroutine 1 [running]:
shell/exploit.(*p58f30).loadDropper.func1(0x7dde73, 0x1e, 0x0,
0x0, 0x81bd00, 0xc42006b560, 0x10, 0x773560)
```

```
/Users/k/go/src/shell/exploit/telnet_loader.go:47 +0x46
path/filepath.Walk(0x7dde73, 0x1e, 0xc420067270, 0x0, 0x0)
/usr/local/go/src/path/filepath/path.go:401 +0x6c
shell/exploit.(*p58f30).loadDropper(0xc42008c540)
/Users/k/go/src/shell/exploit/telnet_loader.go:46 +0x14e
shell/exploit.(*p58f30).init(0xc42008c540, 0x79ecc0,
0xc420067201, 0xc42008c540)
/Users/k/go/src/shell/exploit/telnet_loader.go:31 +0xb5
shell/exploit.init.0()
/Users/k/go/src/shell/exploit/telnet_loader.go:327 +0x2d
```

More interesting is the sample CoinMiner E. After execution, it shows evasive behavior, as it does only result in a quick spike of higher CPU or RAM usage. This means that the malware sample does not start mining cryptocurrency after its execution. Still, one thing that is observable is suspicious network traffic. The infected instances contact two IP addresses right after the execution of the malware sample. The first IP address is `103.140.238.41` located in Hong Kong, and the second one is `165.3.86.35` in Cape Town. One SYN message is sent to each IP address and no response message is received. A plausible explanation is that these two addresses are a part of the C&C structure and that the malware waits for a command to be active. After 40 minutes no malicious behavior could be observed, so the analysis process was stopped.

CoinMiner F executes with no issues. After execution, the sample uses between 70% and 100% of a CPU core, while the RAM usage is relatively low at about 0.7%. Both metrics are noticeably lower than that of CoinMiner C. When observing the network metrics, a stark difference appears: While CoinMiner C only sent very few packages, CoinMiner F sent packages to over 23'000 unique IP addresses in just two minutes. The vast majority of packets are TCP SYN messages which can be with Wireshark by downloading the related `.pcap`. It should also be noted that most messages did not receive a response. This could be anti-analysis behavior: While it makes it easy to realize that malicious software has been installed, it makes it difficult to figure out the C&C structure of the malware sample. A further possibility is that this is done to scan which ports are open on the host device: The traffic observed is sent from non-standard ports, which again can be found by analyzing the `.pcap` file. The port numbers of the source (*i.e.,* the infected instance) range from 16'000 to 65'500. A further noteworthy observation is that the malware sample creates two directories in the `/home` directory, where it was executed. The directory is named `.3ZqifV7E`, which seems to be a random string. It is also a hidden directory, as it begins with a dot. The `.3ZqifV7E (deleted)` indicates that some system calls have been executed in a folder that has been marked for deletion. This possibly is further evidence of obfuscating behavior.

Despite being a distinct sample, CoinMiner G shows very similar behavior as CoinMiner F. Both CPU and RAM usage is on a very similar level. The CPU usage is between 50% and 105% of a core, and the RAM usage is at 0.6%. One notable difference is that after ca. 1.5 minutes of execution time, there is a one-minute break where the CPU usage drops to less than one percent. More importantly, the same traffic pattern is observable: The malware initiates SYN messages from a large range of ports. In this case, the malware contacted 28'766 unique IP addresses in 3 minutes. A further similarity to sample F is

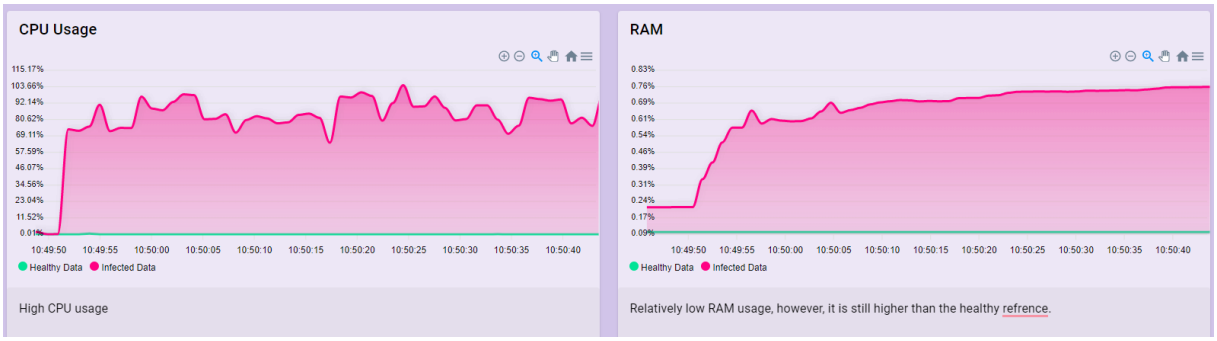


Figure 6.5: CPU and RAM usage of CoinMiner F

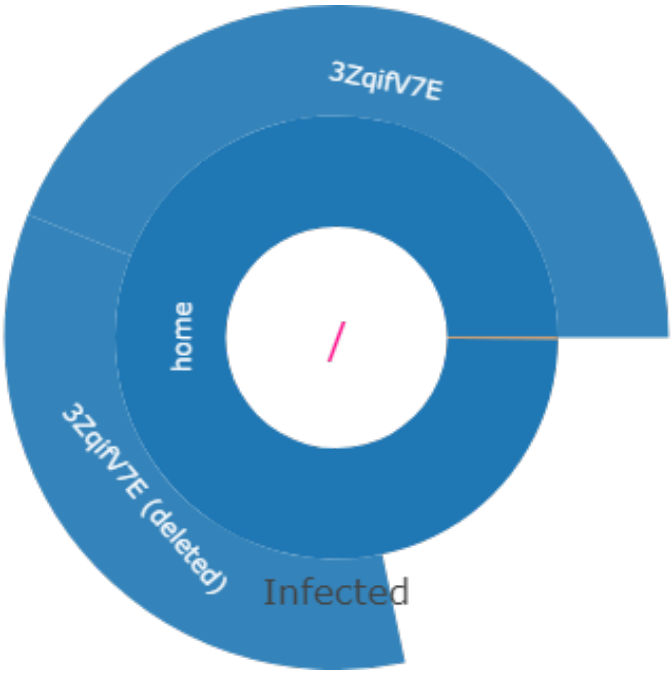
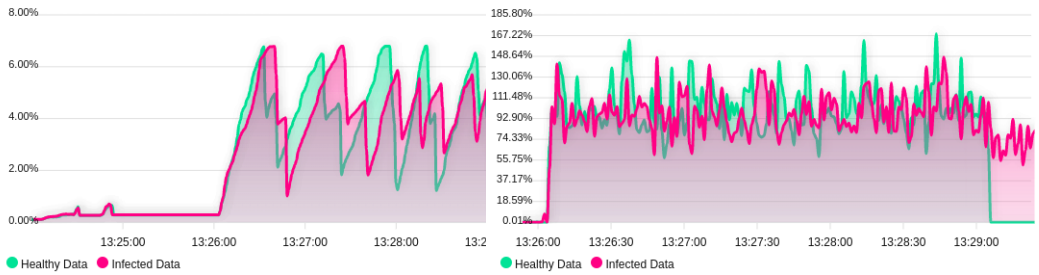


Figure 6.6: Directory Suburn Chart of CoinMiner F

the fact, that directories called `.9VtfueGy` and `.9VtfueGy (deleted)` were created and in which over 300'000 system calls were observed. However, during the live-analysis `ls -a` did show no such directory. Based on their similar behavior, samples F and G are assumed to be closely related.

To conclude, out of seven samples that were analyzed, only three explicitly showed the malicious behavior that is expected from cryptojacking software. Three samples failed during execution, one of which was due to incompatibility with the host architecture. The other two samples most likely failed to execute due to errors in their code. Only one sample was able to obfuscate its presence. However, it is unclear whether this obfuscation was caused by the environment, or because of a lack of instructions from the C&C structure.

Figure 6.7: Charts of Mirai Malware under Load



(a) RAM load with Mirai (b) CPU Load with Mirai

6.1.3 Malware Activity under Stress

In this last case study, malware analysis is performed alongside other running workloads inside the sandboxed environment. This is done in order to investigate the impact of malware in existing production environments running high-performance workloads. Additionally, it will serve as an evaluation of *SecBox*'s ability to analyze malware and extract its core functionalities under aggravating circumstances. To simulate load on the system, the load generating tool `stress` [154] is used and executed through the command highlighted in Listing 6.3.

Listing 6.3: Stress Load Generation Command

```
stress --cpu 8 --io 4 --vm 2 --vm-bytes 500M --timeout 180s
```

This spawns 8 workers generating load on the CPU, 4 workers generating I/O traffic, and 2 workers continuously calling `malloc` and `free` on 500MB of memory for three minutes. A closer look is taken at another sample of Mirai, which is referred to as Mirai B in Appendix B.2. All malware tests are conducted under the previously described load circumstances.

The selected Mirai sample was tagged correctly, thus making sure it is able to be executed in the available 64-bit environment. On execution `xXxSlicexXxxVEGA.` is printed in the output. Since the load generator impacts CPU and RAM the most, as can be seen in Figure 6.7, while the load generator is active the real-time graphs do not yield significant insights regarding the malware's activity. Interestingly enough, however, this sample of Mirai induces significant load to the infected instance after the load generator has ended. However, it does not induce a discernible amount of load while the load generator is active.

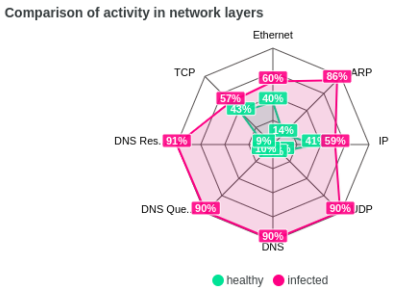
One plausible explanation for this behavior is that the malware caps its activity at a certain system load, so as to not introduce extreme load on the system and continue its operation without issue or discovery.

Thankfully, the network layer and system call data give us much-needed clarity in this scenario. The generated graphs are visible in Figure 6.8. Especially for the network activity layers, the differences are visible very well despite the load.

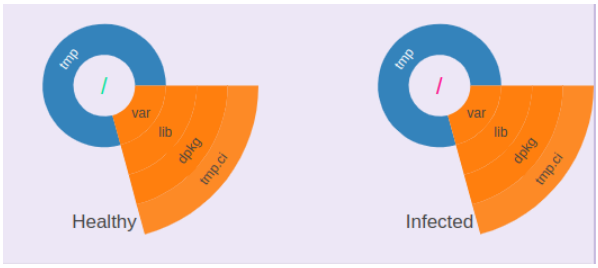
The list of contacted IPs also provides insights, despite the load on the system. The IP `37.44.238.144` was contacted multiple times and through OSINT its IP (a known un-

trusted IP from France) was identified. Additionally, similar to the previously analyzed sample in Subsection 6.1.1 Google's public DNS `8.8.8.8` was contacted. Additionally,

Figure 6.8: Network Layer and Directory Activity - Mirai



(a) Network Layer Activity - Mirai



(b) Directory Activity - Mirai

while the sunburst charts qualitatively look similar, the number of system calls executed in the `/` directory differs vastly for the two instances, namely 72'640 in the healthy case and 451'280 for the infected instance. For further insights, the two system call and packet collections could be contrasted against each other to gain insights in a more fine-grained manner. The read-write chart only shows a slightly higher number in the infected sandbox, not yielding many insights. To summarize, *SecBox* enables a user to detect malware activity with the sandbox rather well despite the generated load. This would however be much harder without the ability to compare the infected system to the healthy baseline. Thus, the design decision to include a separate baseline instance enables a user to discern malware activity despite a noisy environment. One further factor to consider for this evaluation is the fact that the analyzed malware and its behavior were known to the user. Discerning the malware's relevant activities and behavior would be a much more arduous undertaking otherwise. However, not every sample a security researcher analyzes is already known to be a specific type of malware and to exhibit certain characteristics. This problem gets worse if the malware in question introduces empty operations to obfuscate its intended behavior and generate load on the system.

6.2 Discussion

In this section, the conducted case studies are discussed based on the previously introduced evaluative dimensions. This is done to find out whether the proposed solution supports the conducted case studies. More importantly, this discussion should help to find which aspects work as intended, and which areas have room for improvement.

6.2.1 User Experience

As a disclaimer, it needs to be said that the developers also evaluated the user experience. As such, this section has some limitations when compared to a user study conducted

with new users. That being said, the flow of the analysis process is quite straightforward. Selecting the malware sample and Ubuntu image can be done with very few clicks. Once the process has been started, users can go to the next screen by clicking a button in the bottom right corner. Buttons are highlighted in eye-catching colors, which serve as signifiers, *i.e.,* a perceivable cue that highlights the fact that they can be clicked.

The real-time rendering of graphs allows the users to gain a deeper feeling of the relation between how they interact with the system and how this interaction affects the displayed metrics. This is especially important for malware execution, as it allows users to realize whether malicious behavior has been observed. This is useful for use cases as presented in the CoinMiner comparison (Subsection 6.1.2), where a user is able to determine why malware fails to execute.

Users potentially are also able to gain further insights into what is needed for successful execution. The post-analysis charts visualize the results during execution, which is further enhanced through the ability to download `.pcap` and the system calls files, allowing for further downstream processing. This proves very valuable in cases, where the charts do not give granular enough insights. Many of the graphs provide rather high-level insights, which may be of limited use. One example of such a graph is the read-write graph, which only shows highly aggregated data. However, many of the charts also allow interaction from the user, such as the CPU, RAM, or directory sunburst chart, highlighting the potential of the proposed solution. Collaboration is made possible through the ability to comment on the report. This feature is basic, as no user accounts or authorship exists. Still, it serves the purpose of providing a way to share findings and save them for later use.

There is some room for improvement regarding the user experience, such as more expansive visual analysis capabilities through more graphs and or applying filters to the data. Furthermore, letting multiple sample run takes a lot of manual steps. More on that issue is discussed in Subsection 6.2.4. However, all in all, *SecBox* allows users to analyze malware with relative ease, giving them insights into the behavior of a set of malware. It should be noted that this solution still requires some basic understanding of Linux, as users need to know how to execute an `.ELF` file. While there many potential features that could be added, the analysis flow of *SecBox* is simple to understand while providing both aggregated and detailed information about what happened during malware execution.

6.2.2 Low Visibility

To evaluate how visible the sandbox is, one can count how many of the nine malware samples studied in Section 6.1 displayed malicious activity. However, since some of the samples did not execute due to incompatibility with the host architecture or bugs in their code, a more representative comparison is to consider only such samples which executed successfully, but for which no malicious behavior was observed. There is only one such sample that successfully evaded analysis, namely the sample CoinMiner E from Subsection 6.1.2. Still, even in this case, potential evidence of its C&C structure could be found.

Still, evaluating visibility is a challenging task, as the lack of malicious behavior does not necessarily stem from the malware's ability to detect that it is inside a sandboxed environment. It might also be caused by the fact that the malware uses timing as an evasion technique or waits for activation by its C&C server. One limitation introduced by *SecBox* is the fact that gVisor does not implement the entire Linux system call interface. This means that malware that uses system calls that are not implemented in gVsior, can determine that it is inside a sandboxed environment. This type of issue poses a challenge for all kinds of sandboxes, as the complete elimination of all artifacts introduced by a secure environment is a very challenging task.

It should also be noted that the impact of such visibility greatly depends on how sophisticated the analyzed malware is. Due to the lack of container-based malware analysis sandboxes, it is likely that many samples do not explicitly employ anti-container techniques, or at least fewer samples employ such techniques when compared to anti-VM techniques. The issue of container-related artifacts has already been raised by Kedrowitsch, Yao, Wang, *et al.* [118], who state that containers are easily identifiable as such. Still, it should be noted that gVisor is under active development and may eventually implement the entire Linux system call interface and continuously remove artifacts. This could possibly result in the existence of very few observable artifacts in the future.

6.2.3 Monitorability

SecBox shows quite an extensive ability to monitor containers from the outside. Three main domains were considered in this solution: Networking, system calls, and performance metrics. The performance metrics give insights into the resources used (CPU and RAM), and the number of processes that were started inside the sandbox. The networking data is recorded in greater detail. This allows for the creation of `.pcap` files that record all traffic from the sandbox. The system calls give deeper insight into how malware interacts with gVisor's Sentry kernel [13]. Since the monitoring is quite extensive, it also records information that was not initiated by the malware but rather by normal system behavior. This is the reason why a second, healthy sandbox was provided as a baseline. This design decision, as investigated in 6.1.3 allows an analyst to use the healthy instance as a baseline both during live and post-analysis. The provided baseline may also prove to be useful for analysis processes outside of *SecBox* based on the exported files since it allows filtering out captured packets and system calls present in both datasets. This allows for the removal of noise.

An important aspect of *SecBox* is its ability to monitor the sandboxes in real time, allowing users to see if malicious behavior occurs. Furthermore, the data is directly visualized, which helps users to understand the monitored behavior better. Related to the monitorability is also the fact that the data is retained for later usage and saved in a database. It is also provided to the user in formats that are easy to understand and process further. A final aspect that relates to monitorability is how scalable the solution is. The main bottleneck is system calls, as they quickly reach multiple millions, depending on how active the malware is. However, all three case studies could be conducted successfully.

6.2.4 Degree of Automation

Many steps are already automated, such as the fetching of the malware and unpacking it. It still needs to be executed manually, which was a design choice to give the user more control over the process. The shutdown and clean-up also need to be initiated by the user but are fully automated after that. Further automation steps depend on the deployment, as *SecBox* consists of multiple components (frontend, backend, and host) which all need to be running at the same time. However, with proper deployment, this poses no issue. The area where most manual steps are required is if a user wants to analyze multiple samples, as only one sample can be started at once. The selection of multiple samples which can be run with a single command would make for a nice feature, but would also require some redesigning as interaction is a core component of *SecBox* which is not easily scalable. Furthermore, the analysis itself needs to be done by the user, as the solution only provides the (visualized) data. The findings from that still need to be made by the user. A possibility to automate this would be for example the integration of a machine learning module, *e.g.,* to determine the damage function of the analyzed malware.

Still, much of the process is automated. The users only need to select some parameters and then execute the malware. The rest is handled by the system, after that the user simply needs to write down their findings, which allows them to share the gained insight with other users.

6.2.5 Performance

In order to assess the performance of *SecBox* the effects of a running instance on the deployed system, as it was described in Section 5.4 was evaluated in three different scenarios. The metrics were collected using the `ps` command [155]. First, the default state of the system was looked at, without any live-analysis in progress. In this case, the performance impact is very low and should allow the system to scale well. The second scenario is a single sandbox, *i.e.,* two container instances running, however, with no malware or user activity occurring inside the sandbox. The performance in this scenario remains unimpacted, with the frontend being reactive, with no noticeable delay. For the third scenario, a previously presented piece of malware, CoinMiner F, is executed inside the sandbox. CoinMiner F is chosen due to the high load it generates continuously on the system, as highlighted in Section 6.1. In this scenario, the performance is impacted significantly, with the user in the frontend experiencing noticeable delays in their interaction with the system. The performance degradation is mostly due to the high load on the CPU, as can be seen in Figure 6.1.

Thus, system performance is impacted highly by the malware executing inside the sandbox. However, since CoinMiner F is one of the most load-intensive pieces of malware found, and the system remained operational throughout the analysis, allowing the user to successfully analyze the respective piece of malware, performance was deemed adequate for use. The deployed system of *SecBox* runs the host alongside the backend on a high-powered machine. Thus, the load generated on lower-end devices, such as common among 32-bit devices, if running resource-intensive malware, could very well impact the systems'

Table 6.1: Performance Impacts on the Deployed System

Scenario	Host CPU	Host RAM	BE CPU	BE RAM	FE RAM	FE CPU
Default State	0.5%	0.032GB	0.6%	0.096GB	0.21%	0.272GB
Empty Sandbox Running	0.6%	0.256GB	3%	0.096GB	0.3%	0.272GB
CoinMiner H Running	26.0%	0.48GB	1.0%	0.192GB	1.5%	0.32GB

ability to perform dynamic malware analysis significantly. Nevertheless, this issue could be remedied by limiting the resources available to a container.

6.2.6 Isolation

To evaluate the isolation our system provides adequately, an extensive study of *SecBox* would be required. Such a study would need to involve malware designed to escape sandboxes and secure systems, such as Trojans or rootkits, as well as an evaluation of potential exploits and artifacts within the gVisor technology. This would, however, be out of the scope of this project. For this reason, a cursory evaluation of the isolation the system provides is performed. For the deployed system the malware detector `maldet` [156] was used to monitor the system during the above-conducted case studies in Section 6.1, as well as during any preliminary and experimental analysis processes. Maldet uses the ClamAV [157] malware signature collection to detect malicious activity and was set up to send notifications both via E-Mail and through inotify events [158]. During the course of this project, no escapes were detected using this approach.

However, since the malware sample size was severely limited and the malware samples in question were, as hinted above, not specialized to force container escapes and damage the surrounding system, the value this evaluation approach provides is limited.

6.3 Limitations

As can be seen in Section 6.1, the proposed solution has various capabilities and is a lightweight, collaborative container-based tool for dynamic malware analysis. As this tool was nearly built from scratch, it also encounters some limitations. A first limitation is that in its current version, users can only choose from a set of provided, although automatically collected, malware samples. While this was a design choice, it limits the ability to evaluate how this tool performs on unseen or unknown malware samples - or even software of which it is unknown whether it is malicious or not.

Furthermore, the deployment in the conducted case studies, as described in Section 5.4, also poses a limitation. Namely, that the containers are deployed inside a VM. This is a trade-off between overhead and isolation. This also means that one big advantage of containers, namely their near-native performance, is lessened. This is especially relevant since some malware employs the difference between native and virtualized performance as an anti-analysis technique [159]. One possible solution for this limitation is to deploy the sandbox in a non-virtualized environment. This leads directly to the next limitation: The

fact that no malware escape has been observed during the case studies does not mean that it is impossible for malware to escape, especially if such malware was explicitly developed to escape gVisor's architecture. While the isolation of the sandbox is strong in theory, a more systematic approach is necessary to evaluate the isolation mechanism.

Chapter 7

Summary and Conclusions

In absence of any notable container-based sandboxes, *SecBox* represents a unique solution for dynamic malware analysis. The defining design decisions are based on a thorough review of both existing dynamic analysis solutions and attack vectors implemented through malware. The platform provides an easy-to-use, scalable solution for analyzing malware. It leverages the small overhead of containerization technology when compared to VMs or emulators, to create an environment in which the behavior of malware can be analyzed interactively and in real-time. The sandboxes are implemented with gVisor [13], which implements its own kernel for strong isolation while remaining lightweight. A number of malware samples are available which are periodically obtained from MalwareBazaar [131]. *SecBox* offers integrated data collection and visualization support, and a multi-step analysis process.

In the first step, the desired sample and ubuntu image is selected. The second step is the live-analysis phase in which two instances are started: one infected sandbox, that contains the malware sample, and a healthy one that can be used as a baseline reference. During this phase, the user can interact with both instances through a CLI. This page also visualizes data in real time, allowing users to directly see the effects of the malware - or of other processes that are running - through four different metrics. After the live-analysis has been ended by the user, the user can select from six charts in the post-analysis. These charts provide deeper insights than the live-analysis screen and many of them have interactive elements for data exploration. The data can be abstracted into three different categories: Data about the performance, network traffic, and system calls. From the selected charts, a report is created. In the report, users can write comments on each graph which allows them to share insights with different users. Additionally, raw data about the system calls and network traffic can be downloaded for downstream processing.

The proposed solution offers various insights into what is happening inside the sandbox when malware is executed. In its current form, *SecBox* can be used to analyze malware to further explore their damage function or C&C structure, as shown in Section 6.1. It is also suited to compare different samples from the same malware family, to gain knowledge about the behavioral differences between samples. Observations about malware behavior can even be made when there is a simulated workload on both the healthy and infected instances, justifying the design choice of running two containers per malware sample.

In conclusion, *SecBox* offers a unique, user-friendly, and scalable solution for dynamic malware analysis, utilizing the benefits of containerization technology to analyze malware in real time. The platform provides interactive data collection and visualization support, as well as a multi-step analysis process, resulting in deeper insights into malware behavior.

7.1 Future Work

Due to the modular architecture of *SecBox* further development has much potential. One important future work is the addition of a machine-learning module to the current solution. Especially extending *SecBox* with automated dataset generation functionality is highly interesting, as two instances can be started at the same time. Thus the infected instance can be compared to the healthy one. This makes it easy to create a labeled data set that contains information about how the malware that is run affects the system's fingerprint. Furthermore, due to the small overhead of containers, this data set creation could be done at a large scale in an efficient manner. Such a data set can then be used to enhance the automatic analysis capabilities of *SecBox*, by training a machine-learning model to recognize behavioral patterns.

With the goal of expanding the functionality of a single analysis process, an epic on MITRE rule definition was removed from the scope due to the overhead it introduces during the design process in Chapter 4. This would allow users to define the behavior they consider critical through pattern matching of system calls. These rules could be specified, *e.g.,* as a `.yaml` file. Most importantly, a user would be able to define a certain behavior they want to observe and have the live-analysis continue until it is observed by *SecBox*.

Further improvements to a single analysis process could introduce a simulation component that leverages docker networking. This would allow a user to specify a certain network structure to set up infected and healthy systems and their connections. During live-analysis, lateral movement of the malware sample through the specified network structure could be observed, in addition to the currently gathered system data. This would be valuable to companies for the simulation of a malware infection incident and to assess the impact such an infection might have on their current network structure. In order to unlock the full potential of this use case, a set of diverse hosts with diverse architectures would be required, as it could allow modeling systems of IoT devices connected through a network. Running multiple sandboxes on a single device would lower the required hardware for such a simulation, exploiting the minimal overhead of containerization.

As can be seen, there are various potential use cases for a sandbox platform that leverages containerization technology. This work successfully demonstrated how container-based sandboxes can be used in a secure and efficient manner.

Bibliography

[1] D. Freeze, *Cybercrime to cost the world 8 trillion annually in 2023*, Dec. 2022. [Online]. Available: https://cybersecurityventures.com/cybercrime-to-cost-the-world-8-trillion-annually-in-2023/.

[2] N. Idika and A. P. Mathur, "A survey of malware detection techniques", *Purdue University*, vol. 48, no. 2, pp. 32–46, 2007.

[3] *Inside a new ryuk ransomware attack*, https://news.sophos.com/en-us/2020/10/14/inside-a-new-ryuk-ransomware-attack/, Accessed: 2022-08-08.

[4] M. Willett, "Lessons of the solarwinds hack", *Survival*, vol. 63, no. 2, pp. 7–26, 2021. DOI: 10.1080/00396338.2021.1906001. eprint: https://doi.org/10.1080/00396338.2021.1906001. [Online]. Available: https://doi.org/10.1080/00396338.2021.1906001.

[5] *Apache log4j security vulnerabilities*, https://logging.apache.org/log4j/2.x/security.html, Accessed: 2022-07-21.

[6] *Av test malware statistics*, https://www.av-test.org/de/statistiken/malware/, Accessed: 2022-08-10.

[7] *Swiss government computer emergency response team: Malware statistics*, https://www.govcert.admin.ch/statistics/malware/, Accessed: 2022-12-15.

[8] *Cuckoo-docs*, https://cuckoo.sh/docs/, Accessed: 2022-08-15.

[9] *Any.run*, https://app.any.run/plans/, Accessed: 2022-08-15.

[10] *Joesandbox*, https://www.joesecurity.org/joe-security-products, Accessed: 2022-08-15.

[11] D. Lucia and J. C. Michael, "A survey on security isolation of virtualization, containers, and unikernels", 2017.

[12] A. Khalimov, S. Benahmed, R. Hussain, *et al.*, "Container-based sandboxes for malware analysis: A compromise worth considering", in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, ser. UCC'19, Auckland, New Zealand: Association for Computing Machinery, 2019, pp. 219–227, ISBN: 9781450368940. DOI: 10.1145/3344341.3368810. [Online]. Available: https://doi.org/10.1145/3344341.3368810.

[13] *Gvisor - the container security platform*, https://gvisor.dev/docs/, Accessed: 2022-08-17.

[14] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Proactive detection of computer worms using model checking", *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 424–438, 2010. DOI: 10.1109/TDSC.2008.74. [Online]. Available: http://infoscience.epfl.ch/record/167545.

[15] A. P. Namanya, A. Cullen, I. U. Awan, and J. P. Disso, "The world of malware: An overview", in *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2018, pp. 420–427. DOI: 10.1109/FiCloud.2018.00067.

[16] J. Koret and E. Bachaalany, *The antivirus hacker's handbook*. John Wiley & Sons, 2015.

[17] J. Aycock, *Spyware and adware*. Springer Science & Business Media, 2010, vol. 50.

[18] S. Pastrana and G. Suarez-Tangil, "A first look at the crypto-mining malware ecosystem: A decade of unrestricted wealth", in *Proceedings of the Internet Measurement Conference*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 73–86, ISBN: 9781450369480. [Online]. Available: https://doi.org/10.1145/3355369.3355576.

[19] M. Lemoudden, N. Bouazza, B. E. Ouahidi, and D. Bourget, "A survey of cloud computing security overview of attack vectors and defense mechanisms", 2013.

[20] S. K. Pandey and B. Mehtre, "A lifecycle based approach for malware analysis", in *2014 Fourth International Conference on Communication Systems and Network Technologies*, 2014, pp. 767–771. DOI: 10.1109/CSNT.2014.161.

[21] S. Kok, A. Abdullah, N. Jhanjhi, and M. Supramaniam, "Ransomware, threat and detection techniques: A review", *Int. J. Comput. Sci. Netw. Secur*, vol. 19, no. 2, p. 136, 2019.

[22] M. Polychronakis, P. Mavrommatis, and N. Provos, "Ghost turns zombie: Exploring the life cycle of web-based malware", 2008.

[23] O. Alrawi, C. Lever, K. Valakuzhy, K. Snow, F. Monrose, M. Antonakakis, *et al.*, "The circle of life: A {large-scale} study of the {iot} malware lifecycle", in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3505–3522.

[24] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks", in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds., Cham: Springer International Publishing, 2020, pp. 23–43, ISBN: 978-3-030-52683-2.

[25] K. L. Chiew, K. S. C. Yong, and C. L. Tan, "A survey of phishing attacks: Their types, vectors and technical approaches", *Expert Systems with Applications*, vol. 106, pp. 1–20, 2018, ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2018.03.050. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417418302070.

[26] *Mitre attck*, https://attack.mitre.org/, Accessed: 2022-07-20.

[27] A. P. Aliga, A. M. John-Otumu, R. E. Imhanhahimi, and A. C. Akpe, "Cross site scripting attacks in web-based applications", *Journal of Advances in Science and Engineering*, vol. 1, no. 2, pp. 25–35, Sep. 2018. DOI: 10.37121/jase.v1i2.19. [Online]. Available: http://sciengtexopen.org/index.php/jase/article/view/19.

[28] M. Antonakakis, T. April, M. Bailey, *et al.*, "Understanding the mirai botnet", in *26th USENIX security symposium (USENIX Security 17)*, 2017, pp. 1093–1110.

[29] J. Vykopal, "A flow-level taxonomy and prevalence of brute force attacks", in *Advances in Computing and Communications*, A. Abraham, J. Lloret Mauri, J. F. Buford, J. Suzuki, and S. M. Thampi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 666–675, ISBN: 978-3-642-22714-1.

[30] A. Mantovani, S. Aonzo, X. Ugarte-Pedrero, A. Merlo, and D. Balzarotti, "Prevalence and impact of low-entropy packing schemes in the malware ecosystem.", in *NDSS*, 2020.

[31] A. Matrosov, E. Rodionov, and S. Bratus, *Rootkits and bootkits: reversing modern malware and next generation threats*. No Starch Press, 2019.

[32] C. Brierley, J. Pont, B. Arief, D. J. Barnes, and J. Hernandez-Castro, "Persistence in linux-based iot malware", in *Secure IT Systems*, M. Asplund and S. Nadjm-Tehrani, Eds., Cham: Springer International Publishing, 2021, pp. 3–19, ISBN: 978-3-030-70852-8.

[33] *Browser extensions*, https://attack.mitre.org/versions/v12/techniques/T1176/, Accessed: 2023-02-06.

[34] S. T. Zargar, J. Joshi, D. Tipper, and S. Member, "A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks", [Online]. Available: www.google.com.

[35] S. Herwig, K. Harvey, G. Hughey, R. Roberts, and D. Levin, "Measurement and analysis of hajime, a peer-to-peer iot botnet", in *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[36] V. Radunović and M. Veinović, "Malware command and control over social media: Towards the server-less infrastructure", *SJEE*, vol. 17, no. 3, pp. 357–375, 2020.

[37] *Spreading the ddos disease and selling the cure*, https://krebsonsecurity.com/2016/10/spreading-the-ddos-disease-and-selling-the-cure/, Accessed: 2022-08-03.

[38] *Octave klaba twitter*, https://twitter.com/olesovhcom/status/778830571677978624, Accessed: 2022-08-03.

[39] *Anna-senpai, the world's largest mirai botnet, client, echo loader, cnc source code release*, https://hackforums.net/showthread.php?tid=5420472, Accessed: 2022-08-04.

[40] *Bashlite github*, https://github.com/anthonygtellez/BASHLITE, Accessed: 2022-08-04.

[41] *New mirai worm knocks 900k germans offline*, https://krebsonsecurity.com/2016/11/new-mirai-worm-knocks-900k-germans-offline/, Accessed: 2022-08-04.

[42] *Arp-scan(1) - linux manual page*, https://linux.die.net/man/1/arp-scan, Accessed: 2023-02-02.

[43] L. Nagy, "Exploring emotet, an elaborate everyday enigma", *Virus Bulletin*, 2019.

[44] *Alert (ta18-201a), emotet malware*, https://www.cisa.gov/uscert/ncas/alerts/TA18-201A, Accessed: 2022-08-08.

[45] *Emotet's return is the canary in the coal mine*, https://news.sophos.com/en-us/2020/07/28/emotets-return-is-the-canary-in-the-coal-mine/, Accessed: 2022-08-08.

[46] C. Patsakis and A. Chrysanthou, "Analysing the fall 2020 emotet campaign", *arXiv preprint arXiv:2011.06479*, 2020.

[47] *Emotet, software s0367*, https://attack.mitre.org/software/S0367/, Accessed: 2022-12-08.

[48] *A one-two punch of emotet, trickbot, ryuk stealing ransoming data*, https://www.cybereason.com/blog/research/one-two-punch-emotet-trickbot-and-ryuk-steal-then-ransom-data, Accessed: 2022-08-08.

[49] *Emotet's central position in the malware ecosystem*, https://news.sophos.com/en-us/2019/12/02/emotets-central-position-in-the-malware-ecosystem/, Accessed: 2022-08-08.

[50] *World's most dangerous malware emotet disrupted through global action*, https://www.europol.europa.eu/media-press/newsroom/news/world-most-dangerous-malware-emotet-disrupted-through-global-action, Accessed: 2022-08-08.

[51] *Ryuk ransomware: A targeted campaign break-down*, https://research.checkpoint.com/2018/ryuk-ransomware-targeted-campaign-break/, Accessed: 2022-08-08.

[52] *Ryuk*, https://attack.mitre.org/versions/v12/software/S0446/, Accessed: 2022-08-08.

[53] *Cobaltstrike*, https://www.cobaltstrike.com/, Accessed: 2022-08-08.

[54] *Flubot: The evolution of a notorious android banking malware*, https://blog.fox-it.com/2022/06/29/flubot-the-evolution-of-a-notorious-android-banking-malware/, Accessed: 2022-08-10.

[55] *Apkprotector*, https://play.google.com/store/apps/details?id=com.mcal.apkprotector&hl=en&gl=US, Accessed: 2022-08-10.

[56] *Play protect*, https://developers.google.com/android/play-protect, Accessed: 2022-08-10.

[57] *Flubot takedown*, https://www.europol.europa.eu/media-press/newsroom/news/takedown-of-sms-based-flubot-spyware-infecting-android-phones, Accessed: 2022-08-10.

[58] *New massive mobile malware ring targeting europe*, https://www.prodaft.com/resource/detail/flubot-new-masssive-mobile-malware-ring-targeting-europe, Accessed: 2022-08-10.

[59] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools", *ACM Computing Surveys*, vol. 44, 2 Feb. 2012, ISSN: 03600300. DOI: 10.1145/2089125.2089126/FORMAT/PDF. [Online]. Available: http://doi.acm.org/10.1145/2089125.2089126.

[60] S. S. Hansen, T. Mark, T. Larsen, M. Stevanovic, and J. M. Pedersen, *An approach for detection and family classification of malware based on behavioral analysis; An approach for detection and family classification of malware based on behavioral analysis*. 2016, ISBN: 9781467385794. DOI: 10.1109/ICCNC.2016.7440587.

[61] S. S. Chakkaravarthy, D. Sangeetha, and V. Vaidehi, "A survey on malware analysis and mitigation techniques", *Computer Science Review*, vol. 32, pp. 1–23, May 2019, ISSN: 1574-0137. DOI: 10.1016/J.COSREV.2019.01.002.

[62] H. Sun, K. Sun, Y. Wang, and J. Jing, "Reliable and trustworthy memory acquisition on smartphones", *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 12, pp. 2547–2561, 2015. DOI: 10.1109/TIFS.2015.2467356.

[63] R. Mosli, R. Li, B. Yuan, and Y. Pan, "Automated malware detection using artifacts in forensic memory images", May 2016, pp. 1–6. DOI: 10.1109/THS.2016.7568881.

[64] C. Rathnayaka and A. Jamdagni, "An efficient approach for advanced malware analysis using memory forensic technique", in *2017 IEEE Trustcom/BigDataSE/ICESS*, 2017, pp. 1145–1150. DOI: 10.1109/Trustcom/BigDataSE/ICESS.2017.365.

[65] K. Thompson, "Reflections on trusting trust", *Commun. ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984, ISSN: 0001-0782. DOI: 10.1145/358198.358210. [Online]. Available: https://doi.org/10.1145/358198.358210.

[66] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications confining the wily hacker", in *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, ser. SSYM'96, San Jose, California: USENIX Association, 1996, p. 1.

[67] D. Uhřiek, "Lisa – multiplatform linux sandbox for analyzing iot malware", 2019.

[68] C. Kruegel, "Full system emulation: Achieving successful automated dynamic analysis of evasive malware", in *Proc. BlackHat USA Security Conference*, 2014, pp. 1–7.

[69] P. Chen, C. Huygens, L. Desmet, and W. Joosen, "Advanced or not? a comparative study of the use of anti-debugging and anti-vm techniques in generic and targeted malware", in *ICT Systems Security and Privacy Protection*, J.-H. Hoepman and S. Katzenbeisser, Eds., Cham: Springer International Publishing, 2016, pp. 323–336, ISBN: 978-3-319-33630-5.

[70] J. S. Reuben, "A survey on virtual machine security", *Helsinki University of Technology*, vol. 2, no. 36, 2007.

[71] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies", *Black Hat*, vol. 1, no. 2012, pp. 1–27, 2012.

[72] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware", in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008, pp. 177–186. DOI: 10.1109/DSN.2008.4630086.

[73] F. Guibernau, "Catch me if you Can!—Detecting sandbox evasion techniques", San Francisco, CA: USENIX Association, Jan. 2020.

[74] P. Feng, J. Sun, S. Liu, and K. Sun, "Uber: Combating sandbox evasion via user behavior emulators", in *International Conference on Information and Communications Security*, Springer, 2019, pp. 34–50.

[75] *Mitre-anti-vm*, https://attack.mitre.org/versions/v11/techniques/T1497/, Accessed: 2022-08-17.

[76] S. N. T.-c. Chiueh and S. Brook, "A survey on virtualization technologies", *Rpe Report*, vol. 142, 2005.

[77] M. D. Schroeder and J. H. Saltzer, "A hardware architecture for implementing protection rings", *Communications of the ACM*, vol. 15, no. 3, pp. 157–170, 1972.

[78] *Qemu*, https://www.qemu.org/, Accessed: 2022-09-09.

[79] *Linux containers github*, https://github.com/lxc/lxc, Accessed: 2022-08-15.

[80] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments", in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013, pp. 233–240. DOI: 10.1109/PDP.2013.41.

[81] *Docker*, https://www.docker.com/, Accessed: 2022-08-17.

[82] ***cgroups(7) — Linux manual page***, https://man7.org/linux/man-pages/man7/cgroups.7.html, Accessed: 2022-08-15.

[83] *Namespaces(7) — linux manual page*, https://man7.org/linux/man-pages/man7/namespaces.7.html, Accessed: 2022-08-15.

[84] *Capabilities(7) — linux manual page*, https://man7.org/linux/man-pages/man2/seccomp.2.html, Accessed: 2022-08-08.

[85] *Seccomp(2) — linux manual page*, https://man7.org/linux/man-pages/man7/capabilities.7.html, Accessed: 2022-08-08.

[86] *Kubernetes hardening guide*, https://media.defense.gov/2021/Aug/03/2002820425/-1/-1/0/CTR_Kubernetes_Hardening_Guidance_1.1_20220315.PDF, Accessed: 2022-08-08.

[87] *Docker hub*, https://hub.docker.com/, Accessed: 2022-08-15.

[88] *Cve-2022-0185 in linux kernel can allow container escape in kubernetes*, https://blog.aquasec.com/cve-2022-0185-linux-kernel-container-escape-in-kubernetes, Accessed: 2022-08-17.

[89] *Cisa known exploited vulnerabilities*, https://www.cisa.gov/known-exploited-vulnerabilities-catalog, Accessed: 2023-01-30.

[90] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security namespace: Making linux security frameworks available to containers", in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 1423–1439, ISBN: 978-1-939133-04-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/sun.

[91] *Docker documentation*, https://docs.docker.com/get-started/overview/, Accessed: 2022-08-17.

[92] *Flaticon*, https://www.flaticon.com/, Accessed: 2022-08-17.

[93] *Nginx logo download*, https://www.logo.wine/logo/Nginx, Accessed: 2022-08-17.

[94] *Redis logo*, https://dwglogo.com/redis/, Accessed: 2022-08-17.

[95] *Open container initiative*, https://opencontainers.org/, Accessed: 2022-08-17.

[96] V. Vouvoutsis, F. Casino, and C. Patsakis, "On the effectiveness of binary emulation in malware classification", *Journal of Information Security and Applications*, vol. 68, p. 103 258, 2022, ISSN: 2214-2126. DOI: https://doi.org/10.1016/j.jisa.2022.103258. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214212622001223.

[97] G. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox", *IEEE Security and Privacy*, vol. 5, pp. 32–39, 2 Mar. 2007, ISSN: 15407993. DOI: 10.1109/MSP.2007.45.

[98] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions", ser. CCS '08, Alexandria, Virginia, USA: Association for Computing Machinery, 2008, pp. 51–62, ISBN: 9781595938107. DOI: 10.1145/1455770.1455779. [Online]. Available: https://doi.org/10.1145/1455770.1455779.

[99] M. M. Hasan and M. M. Rahman, "Ranshunt: A support vector machines based ransomware analysis framework with integrated feature set", in *2017 20th International Conference of Computer and Information Technology (ICCIT)*, 2017, pp. 1–7. DOI: 10.1109/ICCITECHN.2017.8281835.

[100] S. Kilgallon, L. De La Rosa, and J. Cavazos, "Improving the effectiveness and efficiency of dynamic malware analysis with machine learning", in *2017 Resilience Week (RWS)*, 2017, pp. 30–36. DOI: 10.1109/RWEEK.2017.8088644.

[101] Z. Liu, L. Zhang, Q. Ni, *et al.*, "An integrated architecture for iot malware analysis and detection", in Jan. 2019, pp. 127–137, ISBN: 978-3-658-25690-6. DOI: 10.1007/978-3-030-14657-3_14.

[102] *Hatching.io - cuckoo 3*, https://hatching.io/cuckoo/, Accessed: 2022-08-15.

[103] *Malwr*, https://www.malwr.ee/, Accessed: 2022-08-15.

[104] *Capev2 github*, https://github.com/kevoreilly/CAPEv2, Accessed: 2022-08-15.

[105] *Cape sandbox*, https://capesandbox.com/analysis/, Accessed: 2022-08-15.

[106] A. Melvin and G. J. Kathrine, "A quest for best: A detailed comparison between drakvuf-vmi-based and cuckoo sandbox-based technique for dynamic malware analysis", in Jan. 2021, pp. 275–290, ISBN: 978-981-15-5284-7. DOI: 10.1007/978-981-15-5285-4_27.

[107] O. Ferrand, "How to detect the cuckoo sandbox and to strengthen it?", *J Comput Virol Hack Tech*, vol. 11, pp. 51–58, 2015. DOI: 10.1007/s11416-014-0224-9. [Online]. Available: http://www.cuckoosandbox.org,.

[108] *Drakvuf*, https://github.com/CERT-Polska/drakvuf-sandbox, Accessed: 2022-08-15.

[109] M. Nunes, P. Burnap, O. Rana, P. Reinecke, and K. Lloyd, "Getting to the root of the problem: A detailed comparison of kernel and user level data for dynamic malware analysis", *Journal of Information Security and Applications*, vol. 48, p. 102 365, 2019, ISSN: 2214-2126. DOI: https://doi.org/10.1016/j.jisa.2019.102365. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214212619300109.

[110] *Falcon sandbox*, https://www.crowdstrike.com/products/threat-intelligence/falcon-sandbox-malware-analysis/, Accessed: 2022-08-15.

[111] *Hybrid-analysis.com*, https://www.hybrid-analysis.com/, Accessed: 2022-08-15.

[112] P. Ferrie, "Attacks on more virtual machine emulators", 2007.

[113] K. Monnappa, "Automating linux malware analysis using limon sandbox", Black Hat Europe 2015, 2015.

[114] Y. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "Iotpot: A novel honeypot for revealing current iot threats", *Journal of Information Processing*, vol. 24, pp. 522–533, May 2016. DOI: 10.2197/ipsjjip.24.522.

[115] H.-V. Le and Q.-D. Ngo, "V-sandbox for dynamic analysis iot botnet", *IEEE Access*, vol. 8, pp. 145 768–145 786, 2020. DOI: 10.1109/ACCESS.2020.3014891.

[116] S. Yonamine, Y. Taenaka, and Y. Kadobayashi, "Tamer: A sandbox for facilitating and automating iot malware analysis with techniques to elicit malicious behavior", in *Proceedings of the 8th International Conference on Information Systems Security and Privacy - Volume 1: ForSE,*, INSTICC, SciTePress, 2022, pp. 677–687, ISBN: 978-989-758-553-1. DOI: 10.5220/0010968300003120.

[117] T. Raffetseder, C. Krügel, and E. Kirda, "Detecting system emulators", vol. 4779, Oct. 2007, pp. 1–18, ISBN: 978-3-540-75495-4. DOI: 10.1007/978-3-540-75496-1_1.

[118] A. Kedrowitsch, D. (Yao, G. Wang, and K. Cameron, "A first look: Using linux containers for deceptive honeypots", in *Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense*, ser. SafeConfig '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 15–22, ISBN: 9781450352031. DOI: 10.1145/3140368.3140371. [Online]. Available: https://doi.org/10.1145/3140368.3140371.

[119] I. Vurdelja, I. Blažić, D. Bojić, and D. Draskovic, "An automated framework for runtime analysis of malicious executables on linux", *Telfor Journal*, vol. 13, pp. 87–91, Jan. 2021. DOI: 10.5937/telfor2102087V.

[120] D. Clegg and R. Barker, *Case Method Fast-Track: A Rad Approach*. USA: Addison-Wesley Longman Publishing Co., Inc., 1994, ISBN: 020162432X.

[121] *Mitre-host-recon*, https://attack.mitre.org/versions/v12/techniques/T1592/, Accessed: 2022-12-17.

[122] *Kata containers*, https://katacontainers.io/, Accessed: 2023-02-01.

[123] *Tcpdump*, https://www.tcpdump.org/, Accessed: 2023-01-22.

[124] *Linux containers metrics api*, https://linuxcontainers.org/lxd/docs/master/metrics/, Accessed: 2023-02-01.

[125] *Ptrace(2) - linux manual page*, https://man7.org/linux/man-pages/man2/ptrace.2.html, Accessed: 2023-02-01.

[126] *Gvisor monitoring process*, https://github.com/google/gvisor/blob/master/pkg/sentry/seccheck/sinks/remote/README.md, Accessed: 2022-12-21.

[127] *Socketio*, https://python-socketio.readthedocs.io/en/latest/, Accessed: 2023-01-22.

[128] *Docker python sdk*, https://docker-py.readthedocs.io/en/stable/, Accessed: 2023-01-22.

[129] *Secbox github repository*, https://github.com/SaltySpatula/SecBox, Accessed: 2023-01-21.

[130] *Dockerfile reference*, https://docs.docker.com/engine/reference/builder/#arg, Accessed: 2023-01-22.

[131] *Malware bazaar*, https://bazaar.abuse.ch/, Accessed: 2023-01-21.

[132] *Abuse.ch*, https://abuse.ch/, Accessed: 2023-01-21.

[133] *Scapy*, https://scapy.net/, Accessed: 2023-01-22.

[134] *Bazel*, https://bazel.build/, Accessed: 2023-01-22.

[135] *Flask socketio*, https://flask-socketio.readthedocs.io/en/latest/, Accessed: 2023-01-22.

[136] *Mongo db*, https://www.mongodb.com/, Accessed: 2023-01-22.

[137] *Vue js*, https://vuejs.org/, Accessed: 2023-01-30.

[138] *Apex charts*, https://apexcharts.com/, Accessed: 2023-01-30.

[139] *Vue3 table lite*, https://vue3-lite-table.vercel.app/quick-start, Accessed: 2023-02-05.

[140] *Plotly*, https://plotly.com/javascript/sunburst-charts/, Accessed: 2023-02-05.

[141] *Wireshark*, https://www.wireshark.org/, Accessed: 2022-12-21.

[142] *How to deploy flask application with nginx and gunicorn on ubuntu*, https://www.rosehosting.com/blog/how-to-deploy-flask-application-with-nginx-and-gunicorn-on-ubuntu-20-04/, Accessed: 2023-01-30.

[143] *Socketio flask deployment*, https://flask-socketio.readthedocs.io/en/latest/deployment.html, Accessed: 2023-01-30.

[144] *Trusty ubuntu container image*, https://hub.docker.com/layers/library/ubuntu/trusty/images/sha256-881afbae521c910f764f7187dbfbca3cc10c26f8bafa458c76dda009a901c29d?context=explore, Accessed: 2022-12-19.

[145] *Iputils package*, https://packages.ubuntu.com/source/bionic/iputils, Accessed: 2022-12-19.

[146] *Busybox*, https://busybox.net/, Accessed: 2022-12-21.

[147] *Binutils*, https://www.gnu.org/software/binutils/, Accessed: 2022-12-21.

[148] *Obfuscated files or information*, https://attack.mitre.org/techniques/T1027/, Accessed: 2022-12-21.

[149] *Ip geolocation*, https://ipgeolocation.io/, Accessed: 2022-12-21.

[150] M. Franco, J. Von der Assen, L. Boillat, *et al.*, "Secgrid: A visual system for the analysis and ml-based classification of cyberattack traffic", in *2021 IEEE 46th Conference on Local Computer Networks (LCN)*, 2021, pp. 140–147. DOI: 10.1109/LCN52139.2021.9524932.

[151] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences", in *AI 2016: Advances in Artificial Intelligence*, B. H. Kang and Q. Bai, Eds., Cham: Springer International Publishing, 2016, pp. 137–149, ISBN: 978-3-319-50127-7.

[152] *Xmrig*, https://xmrig.com/, Accessed: 2023-02-02.

[153] *Linux analysis report - uvzmfnywzr.elf*, https://www.joesandbox.com/analysis/768327/0/html, Accessed: 2023-02-02.

[154] *Stress(1) - linux manual page*, https://linux.die.net/man/1/stress, Accessed: 2023-01-26.

[155] *Ps(1) — linux manual page*, https://man7.org/linux/man-pages/man1/ps.1.html, Accessed: 2023-02-02.

[156] *Linux malware detect*, https://www.rfxn.com/projects/linux-malware-detect/, Accessed: 2023-01-27.

[157] *Clam antivirus*, https://www.clamav.net/, Accessed: 2023-01-27.

[158] *Inotify man page*, https://man7.org/linux/man-pages/man7/inotify.7.html, Accessed: 2023-01-27.

[159] *Virtualization/sandbox evasion: Time based evasion*, https://attack.mitre.org/versions/v12/techniques/T1497/003/, Accessed: 2023-02-02.

[160] *Install nodejs 16 on ubuntu 20.04*, https://www.stewright.me/2022/01/tutorial-install-nodejs-16-on-ubuntu-20-04/, Accessed: 2023-02-05.

[161] *Yarn installation*, https://classic.yarnpkg.com/lang/en/docs/install/#debian-stable, Accessed: 2023-02-05.

[162] *Python 3.9.0*, https://www.python.org/downloads/release/python-390/, Accessed: 2023-02-05.

[163] *Pip installation*, `https://pip.pypa.io/en/stable/installation/`, Accessed: 2023-02-05.

Listings

Abbreviations

AI	Artificial Intelligence
AV	Antivirus
ARP	Address Resolution Protocol
API	Application Programming Interface
BTC	Bitcoin
C&C	Command and Control
CLI	Command Line Interface
CPU	Central Processing Unit
DoS	Denial of Service
DDoS	Distributed Denial of Service
ELF	Executable and Linkable Format
GPU	Graphics Processing Unit
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IP	Internet Protocol
IT	Information Technology
IoT	Internet of Things
MAAS	Malware-as-a-Service
ML	Machine Learning
OS	Operating System
OSINT	Open-Source Intelligence
PoW	Proof of Work
RAM	Random-Access Memory
RSS	Resident Set Size
SAAS	Software as a Service
SDK	Software Development Kit
SME	Small and Medium Enterprises
SYN	Synchronize
TA	Threat Actor
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UDS	Unix Domain Socket
USD	United States Dollar
USA	United States of America
UUID	Universally Unique Identifier
VBA	Visual Basic for Applications

VM	Virtual Machine
VMI	Virtual Machine Introspection
VMM	Virtual Machine Monitor
XSS	Cross-Site-Scripting

List of Figures

List of Tables

Appendix A

Installation Guidelines

The following instructions allow a user to set up an instance of *SecBox*. They can also be found on the *SecBox* Github [129] in the respective `readme.txt`. Each system component should be distributed to the desired environment for deployment. The frontend is located in the `SecBox/app` directory, the backend in the `SecBox/api` and `SecBox/backend` directories and the host can be found in the `SecBox/host` directory.

A.0.1 Frontend Setup

The frontend requires Node 16.X [160] and Yarn [161]. To install the dependencies, navigate to `SecBox/app` and run `npm install`.

Create a `.env` file with the value VUE_APP_ROOT and the desired backend IP, e.g. `VUE_APP_ROOT="localhost:5000"` to run locally. To deploy the frontend locally, run `npm run serve`.

A.0.2 Backend Setup

The backend requires Python 3.9 [162] and pip [163]. All dependencies required for the backend can be installed in `SecBox/api` with `pip install -r requirements.txt`.

To add a MongoDB [136], a `.env` file must be configured in this directory:

```
DB_PORT= "27017"
HOST_BITNESS= 64
HOST="mongodb+srv://"
DB= "DB?"
```

The backend can be started with the command `python3 webapp_api.py`.

A.0.3 Host Setup

Again navigate to the folder `SecBox/host` and install the python dependencies with `pip install -r requirements.txt`.

To set up the host on a machine, run:

```
sudo ./setup.sh
sudo ./setup_bazel_gvisor.sh
```

The host can be run from the host directory with the command `sudo python3 ./host.py`. Note that this script needs to be executed as root.

To configure the connection to the respective backend, specify a `.env` file after the following example:

```
#HOST
BE_IP_PORT = 'http://localhost:5000'
```

Appendix B

Figures

Table B.1: In-depth specification of user stories

US	Priority	User Story	Definition of Done
1.1	1	As a user, I want to start the analysis process by configuring a new environment. Each new analysis process gets a unique Job ID from which I can identify the process.	There is a button for creating a new analysis process.
1.2	1	As a user, I want to continue the analysis process by choosing a malware to be analyzed. I can do this by either selecting an existing malware from a list or adding a new malware to the list.	There exists an interface for selecting a piece of malware to be analyzed. The chosen malware is then added to the environment.
1.3	2	As a user I want to continue the analysis process by specifying an OS environment for the malware to be run in. I can do this by selecting a Linux-based OS from the provided list.	There exists an interface for selecting the OS environment the malware is to be run in. The malware is then executed in the desired OS environment.
1.4	3	As a user I want to continue the analysis process by selecting or providing Docker images to be included in the environment of the malware.	There exists an interface for the user to provide the desired docker images to be deployed alongside the malware. The desired packages are deployed to the execution environment.
1.5	1	As a user I want to be able to start the execution of the malware in the specified environment.	There is a button for the user to declare the setup as finished and the malware execution is started.
2.1	2	As a user I want to be able to interact with a sandboxed OS through a CLI.	During malware execution, the user is able to interact with the sandboxed system through a command line interface.
2.2	1	As a user, I want to be provided with a comparison of 2 systems during execution, in order to assess the impact of a specific malware on system operation: "Malware System" and "Healthy System".	During malware execution, the user is provided with comparative visualizations of the two environments. Any user interaction is performed in both environments.
2.3	1	As a user, I want to view live statistics of the sandboxed environment during malware execution.	During malware execution, there exist various visualizations of desired metrics. The data shown in the visualizations represents the real state of the sandboxed environment.
2.3.1	1	As a user, I want to view live hardware metrics of the sandboxed environment during malware execution. Hardware metrics include: CPU Utilization, Memory Utilization, and Storage Utilization.	During malware execution, there exist various visualizations of the specified hardware metrics. The data shown in the visualizations respresents the real state of the sandboxed environment.
2.3.2	2	As a user, I want to view live network traffic data of the sandboxed environment during malware execution.	During malware execution, there exist various visualizations of the network traffic. The data shown in the visualizations respresents the real state of the sandboxed environment.
2.3.3	2	As a user, I want to view live file system interaction data of the malware within the sandboxed environment.	During malware execution, there exists (a) visualization(s) of file system interaction. The data shown in the visualizations respresents the real state of the sandboxed environment.
2.4	1	As a user, I want to be able to terminate malware execution.	There exists a button to terminate the execution of the desired malware in the desired environment. The execution is stopped upon user interaction with this button
3.1	1	As a user, I want to create a dashboard view of the data collected from the previously executed malware.	There exists an interactive dashboard creator that allows the user to display the desired data.
3.2	1	As a user, I want to be able to save a created dashboard.	There exists a button to allow a user to save their created dashboard.
3.3	3	As a user I want to be able to annotate the created dashboard view during creation.	There exists a dashboard annotation feature which allows users to mark and comment on specific elements of visualizations.
3.4	2	As a user, I want to download the raw data from a specific malware execution.	There exists a button that when interacted with triggers the download of the raw execution data.
3.5	3	As a user, I want to download the raw data from a specific malware execution.	There exists a button that when interacted with triggers the download of the raw execution data.
4.1	1	As a user I want to be able to view previously saved reports.	There exists a history screen of previously created reports.
4.2	2	As a user I want to be able to edit previously saved reports.	There exists a button that allows a user to re-enter the report creation screen of a previously created report that allows them to edit it.
4.3	3	As a user I want to be able to share previously saved reports via permalinks.	There exists a button that allows a user to copy a permalink to a previously created report.
4.4	3	As a user, I want to be able to delete previously saved reports.	There exists a button that allows a user to delete a previously created report.
4.5	3	As a user I want to be able to re-run a malware execution based on a previously saved report.	There exists a button that allows a user to enter an automatically configured environment setup.
5.1	3	As a user, I want to be able to create, update and delete (CRUD) rules following the MITRE classification matrix to be matched with syscalls.	There exists a page where a user can see, edit and delete MITRE system call rules.

Table B.2: Malware Samples Used

Malware Sample	**Malware Bazaar Entry**
Mirai A	https://bazaar.abuse.ch/sample/94c25a30ee8bb20c0f19ccb59f509b455b3c884017b2fc4f15c240bcb1958e28/
Mirai B	https://bazaar.abuse.ch/sample/46aec81320084c804f9227a52c63d4b0f49aef84465baba2e175c2ed60ef3d25/
CoinMiner A	https://bazaar.abuse.ch/sample/0ad68d5804804c25a6f6f3d87cc3a3886583f69b7115ba01ab7c6dd96a186404/
CoinMiner B	https://bazaar.abuse.ch/sample/327d653419897c34808489ea3739e53a0b375a5a13f290253db6f7f62b57d21e/
CoinMiner C	https://bazaar.abuse.ch/sample/e2c3e81aa24b20ac71147340adc1eaedf077ad00e4a2359e3db47b166cf5411a/
CoinMiner D	https://bazaar.abuse.ch/sample/89bb089b93f771a0127e2b764d8741c95ead4f43f61043c51bb67ff3b20a8361/
CoinMiner E	https://bazaar.abuse.ch/sample/39164cfdd35b9e2ff89106d1b66ba9dfc186f45370616b6aa2a73afbdab633bd/
CoinMiner F	https://bazaar.abuse.ch/sample/f425fa120bc1b3926ed92ed5cec74898f1d40c3059ae443f816a9e10a2699f80/
CoinMiner G	https://bazaar.abuse.ch/sample/81e5beaf5683b92b1a238a909805bd02298830dc90d383facdebdb170230ec34/