
Malware Information Grabber

Master's Thesis
Emil Villefrance

Aalborg University
Electronic Systems



Electronic Systems
Aalborg University
<https://www.aau.dk>

AALBORG UNIVERSITY

STUDENT REPORT

Title:

Malware Information Grabber

Theme:

Master's Thesis

Project Period:

Spring Semester 2023

Project Group:

-

Participant(s):

Emil Villefrance

Supervisor(s):

Kim Christensen

Jens Myrup Pedersen

Copies: 1**Page Numbers:** 70**Date of Completion:**

June 2, 2023

Abstract:

This project investigates how the lack of recent behavioral malware data sets can be addressed, as well as whether the widespread use of Cuckoo sandbox, despite its lack of maintenance, is a problem. Analysis reports were scraped from online instances of Cuckoo and Cape sandbox to form the basis of a published data set with behavioral features. Additionally, the source code for the scrapers were published to enable other researchers to keep the data set up to date by scraping new reports and adding them to the data set at a later point in time. The data set was used to investigate whether Cuckoo sandbox is evaded more often than Cape by comparing reports from the two sandboxes for the same samples. This project shows several indications that this may be the case, however, more research is needed to confirm this on a general level.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Contents

Preface	iv
1 Introduction	1
1.1 Initial Problem Statement	2
1.2 Main Contributions	2
1.3 Limitations	3
1.4 Outline	3
2 Problem Analysis	4
2.1 Malware Definition and Classification	4
2.1.1 Malware Labelling	7
2.2 Dynamic Malware Analysis	8
2.3 Malware Evasion	11
2.4 Online Public Malware Sandboxes	14
2.4.1 Existing Online Public Malware Sandboxes	15
2.4.2 Online Public Malware Sandbox Data Privacy	16
2.5 Malware Data Sets	17
2.5.1 Malware Data Set Quality	18
2.6 Problem Statement	18
3 Methodology and Design	20
3.1 Lack of data sets	20
3.2 Cuckoo Sandbox Evasion	22
3.3 Data Collection	24
3.4 Publishing Data Set and Source Code	25
3.5 Summary	25
4 Implementation	26
4.1 Scraping Cuckoo	27
4.1.1 Cuckoo Analysis Report Structure	27
4.1.2 Cuckoo Start Spider	28
4.1.3 Cuckoo Spider	32

4.1.4	Unzip Cuckoo reports and convert to single .json file	33
4.2	Scraping Cape	34
4.3	Ethical Considerations	34
4.4	Analyzing Results	35
4.4.1	remove_duplicates.py	35
4.4.2	check_same_hash.py	36
4.4.3	calculate_results_matching_hashes.py & calculate_results_single_sandbox.py .	36
4.4.4	check_evasion_signature_from_repos.py	36
4.4.5	find_signature_only_in_one_sandbox.py	36
4.5	Summary	36
5	Results	37
5.1	Comparing Reports	39
5.1.1	antivm_generic_cpu	39
5.1.2	antisandbox_sleep	40
5.1.3	ransomware_extensions	40
5.2	Tests	40
5.2.1	Pafish	40
5.2.2	Al-Khaser	41
5.2.3	anticuckoo.exe	41
6	Discussion	42
6.1	RQ1: How can a behavioral malware data set be created, that can be used for future research?	42
6.2	RQ2: How can a behavioral malware data set be continuously updated in an easy way? .	43
6.3	RQ3: How can it be investigated whether Cuckoo Sandbox is evaded more often by malware compared to an actively maintained sandbox, such as Cape?	43
6.4	Main findings	45
7	Conclusion	46
	Bibliography	48
A	Appendix: Implementation	55
A.1	Scraping	55
A.2	Analyzing Results	61
B	Appendix: Results	68
B.1	Comparing Reports	68
B.2	Tests	70

Preface

Aalborg University, June 2, 2023

Emil Villefrance
<eville21@student.aau.dk>

Chapter 1

Introduction

The threat of malware attacks is increasing year by year. The number of cyber attacks went up by 38% in 2022 compared to the year before [1]. The amount of malware has been increasing by between 80 million and 150 million samples per year since since 2015 [54]. Costs associated with cyber crime is expected to hit over \$10 trillion annually by 2025, up from \$6 trillion in 2021 and \$3 trillion in 2015 [36]. The cost associated with ransomware alone is expected to reach \$265 billion by 2031 [39]. The most direct consequence when a company gets hit by ransomware is the financial loss associated with being unable to operate their business normally for the period of the ransomware attack, as well as the costs associated with restoring the systems and getting back up and running normally again. Data may also be lost, and in the worst cases, a ransomware attack may even cost lives by hitting medical equipment [52]. To combat this trend, security researchers need ways to analyze new malware families in an automated manner, instead of relying solely on manually creating signatures to detect malware [38].

Malware can be analyzed statically, where the binary code is investigated without actually executing the code. This usually involves reverse engineering the file to find out what the malware is actually doing. As opposed to static analysis, dynamic malware analysis work by executing the sample and observing its behavior. Dynamic analysis can be performed in analysis environments, such as virtual machines, sandboxes, emulators, hypervisor type-1 and bare-metal systems. Online sandboxes and dynamic analysis services exist, where users can upload a file that will then be analyzed and an analysis report will be generated based on what the file did while under analysis. Examples of online dynamic malware analysis sandboxes include commercial options such as any.run [14] and JoeSandbox [47], as well as online instances of open-source sandboxes such as Cuckoo [35] and Cape [23]. Submitting file samples to these online sandboxes under the free tier, will make the analysis report for the file publicly available for anyone to see. Therefore, thousands (and for some sites millions) of public reports exist, where a lot of time and resources have been spent on analyzing malware (and benign) samples. The public analysis reports contain valuable information about malware behavior, but the data may not be utilized to its full extend, since it can be lost in the huge amounts of data (ie. not a lot of people reach the analysis reports on page 898).

This project aims to investigate whether public analysis reports from multiple online sandboxes can be collected in an automated manner, and be used to make a data set that can be used by other researchers for future studies.

Once the malware analysis reports have been gathered, the goal is to see if it is possible to find any interesting patterns in the data.

1.1 Initial Problem Statement

To help scope the project, an initial problem statement is formulated as follows:

How can public malware analysis reports be collected from multiple online sandboxes in an automated way, and how can this data be used to find interesting patterns?

To further help guide the problem analysis, the following subquestions are formulated:

- How is malware defined, and how can it be classified?
- How can malware be analyzed?
- How can malware evade analysis?
- Which online public malware sandboxes exist, and how can their public data be utilized?
- What is the state of existing malware data sets?

A problem analysis based on the questions above, will be carried out in chapter 2.

1.2 Main Contributions

The first contribution of this work is to turn the public reports from two open-source public dynamic malware analysis sandboxes into a data set that can be used for further research. This is accomplished by crawling Cuckoo [35] and Cape Sandbox [23], and parsing the data into an easy to use format, like JSON. The data set can be updated by running the crawler again at a later point in time, to ensure the data set can be updated as needed.

The other contribution is to use the gathered data set to analyze the differences in malware evasion behavior across Cuckoo and Cape for the same malware samples. The purpose of this is to investigate whether Cuckoo is more often evaded by malware, which may lead to incomplete results due to the malware not exhibiting its true malicious intent.

1.3 Limitations

A limitation of this study is that it only compares data from two sandboxes (Cuckoo and Cape), and that it does not have ground-truth labels [84] for the gathered data. This means that we can not be sure that a given malware sample did not manage to evade both Cuckoo and Cape sandbox. Also, the data was gathered over a limited period of time, which may impact the results. This limitation can be mitigated by running the study again at a later point in time and possibly over a longer time period, to gather more data leading to more robust results.

1.4 Outline

The rest of the report is structured as follows. First, a problem analysis will be carried out in chapter 2 to analyze the existing literature and come up with a problem statement based on this. chapter 3 will then discuss the methodology and design of the project, as to which methods exist to solve the problem identified in chapter 2, including which choices were made and why. Chapter 4 will go over the actual Implementation of the solution based on chapter 3. The results will be presented in chapter 5 and discussed in chapter 6. Finally, the project will be concluded in chapter 7.

Chapter 2

Problem Analysis

In this chapter a problem analysis will be performed based on the initial problem statement and questions in section 1.1. First, in section 2.1 a definition of malware will be presented, along with different ways to classify malware. Then in section 2.2 dynamic malware analysis and the different analysis setups will be discussed. Section 2.3 will discuss the issue of malware evasion and how anti evasion techniques are required to analyze evading malware in a dynamic analysis environment like a sandbox. In section 2.4 online public malware sandboxes are discussed, both in terms of previous studies using data from them as well as which specific sandboxes exist. Data privacy in online malware sandboxes is also discussed in section 2.4. Existing malware data sets are reviewed in section 2.5 along with the common pitfalls regarding data set quality. Based on the problem analysis, the chapter ends with a problem statement in section 2.6.

2.1 Malware Definition and Classification

In this section malware will be defined, and common ways of classifying malware will be presented. The challenges with malware labelling will also be discussed.

To work with malware and malware analysis, we first need a common understanding of what malware is. In this project the definition from Or-Meir et al. [60] will be used. Other ways of defining malware exist, but the definition by Or-Meir et al. was chosen due to it being from a recent state of the art survey about dynamic malware analysis. Or-Meir et al. reviewed prior definitions and found that they had too much focus on the malware developers' intentions, which may not be possible to know. Therefore, they come up with the following definition, which will be used in this project:

“Malware is code running on a computerized system whose presence or behavior the system administrators are unaware of; were the system administrators aware of the code and its behavior, they would not permit it to run.

Malware compromises the confidentiality, integrity or the availability of the system by exploiting existing vulnerabilities in a system or by creating new ones.”[60]

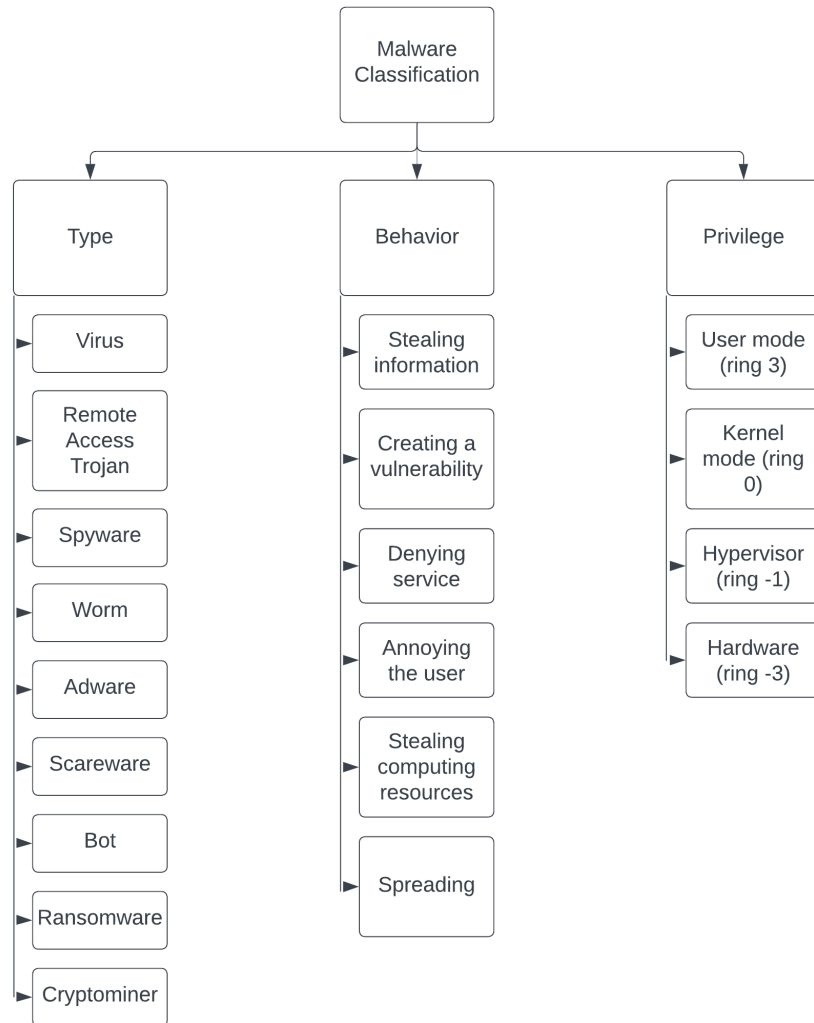


Figure 2.1: Common ways to classify malware (as suggested by Or-Meir et al.[60])

Figure 2.1 shows common ways to classify malware as presented by Or-Meir et al., which is by type, behavior or privilege. A type of malware will have some associated behaviors. A worm for instance works by Spreading and Denying service, whereas a cryptominer works by Stealing computing resources [60]. Malware is usually only categorized as a single type (e.g. Virus, Ransomware, Cryptominer), but it can be associated with multiple behaviors. Therefore, categorizing by behavior is more flexible, since any combination of behavior is possible. Defining the behaviors and type a malware can have is a work in progress, since new behaviors and types may arise as the threat landscape is evolving (e.g. the types Ransomware and Cryptominer as well as the behavior Stealing computing resources are fairly recent additions to the taxonomy). Conversely, some behaviors (such as Annoying the user) may become less common over time, and may end up disappearing entirely

[60].

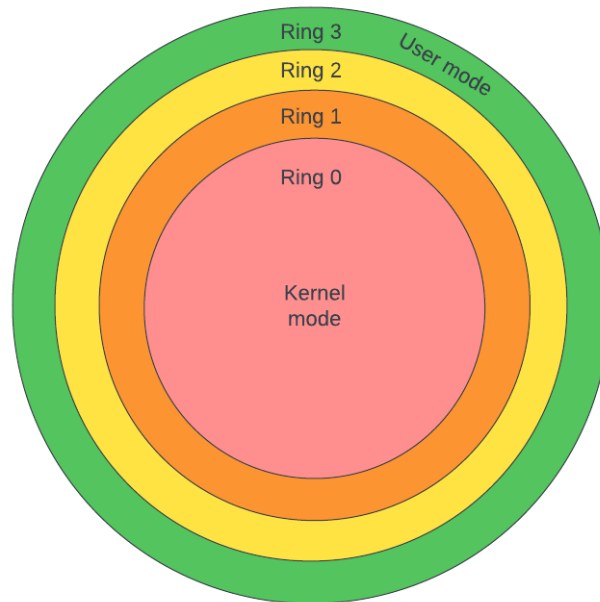


Figure 2.2: Protection rings illustrating the different layers of privilege in an operating system (inspired by Reid & Caelli [69]).

Classifying a malware by privilege lets us know what protection ring the malware operates under. Figure 2.2 (inspired by Reid & Caelli [69]) shows the different protection rings in an operating system according to the level of privilege. Ring 3 (user mode) is the least privileged mode and is where most applications run. Some operating systems, including Windows, do not utilize Ring 1 and Ring 2. Ring 0 (kernel mode) is responsible for handling system resources and have direct access to the CPU and physical devices attached to the computer. This mode is reserved for the operating system kernel, as well as device drivers (since they need access to the hardware). A malware running in kernel mode is called a rootkit, and can be very difficult to get rid of and detect, since it has direct privileged control of everything happening in the system, and can therefore hide and manifest itself very well. Even more privileges can be obtained if the malware gain access to a Type 1 Hypervisor (even though this is not a real protection ring, it is sometimes referred to as Ring -1 due to having more privileges than Ring 0). This will allow it to control the guest Virtual Machines (VMs) running on the host, and take away root access from the installed operating systems. Malware operating at this level is called a virtual machine-based rootkit (VMBR). Finally, infecting the hardware directly, usually in the form of a malicious firmware update, will gain the attacker the most privileges. Malware running at the hardware level can be referred to as a Ring -3 rootkit. Similar to Ring -1, Ring -3 is not a real protection ring, but it can be a useful abstraction

when thinking about the different levels of privileges malware can have [60].

Any behavior can be paired with any protection ring. A malware may for instance work by Stealing information in either user mode, kernel mode (rootkit) or at the hypervisor or hardware level [60]. Most malware run in user mode, since developing and deploying rootkits is complex and often unnecessary, unless the attacker is an advanced persistent threat (APT) or highly skilled cybercriminals targeting corporations [77].

2.1.1 Malware Labelling

Antivirus vendors label malware automatically based on static and dynamic features and signatures. However, they don't use the same taxonomy, which make it hard to compare labels between different vendors. A malware may for instance be labelled by its type/class (virus, worm, etc.) or the malware family it belongs to (WannaCry, Petya, etc.). It may also be labelled by file properties (e.g. whether its packed or not, or the specific packer used like Themida). Additionally, it may be labelled according to its behavior (Stealing information, Denying service, etc.). It can also simply be labelled as malicious or benign [75].

Looking at a random sample on VirusTotal [85] shows an example of the many different ways antivirus vendors label a sample. Some for instance label this sample as "Win32:Malware-gen", "Unsafe" or "Malicious". One vendor labels it according to file properties, e.g. "AI:Packer.2C6AB60816". Others label it according to behavior, e.g. "Trojan.DownLoader19.10684", "Trojan.Injector.vl!c", while others label it according to the malware family they think it belongs to, e.g. "Gen:Variant.-Zusy.353316", "Trojan.Win32.Bandok", "Backdoor.Bandock.A". This shows that for the same malware sample, a lot of different labels are generated, based on different (often closed) taxonomies, making it hard to use for research purposes. Ways to standardize the taxonomy have been proposed previously, for instance with the Malware Attribute Enumeration and Characterization (MAEC) by Mitre [53]. This has had a low adoption rate though, and requires frequent updating due to having a fixed set of tags. Another way to extract labels or tags is to use a tool like AVClass2 [75]. The tool work by taking reports, for example from VirusTotal as input. Based on these reports, it will extract relevant tags according to behavior, class, file properties and family. It works by taking the input labels (e.g. "Trojan.DownLoader19.10684", "Backdoor.Bandock.A", "Win32:Malware-gen") and extracting each individual token from them (eg. "trojan", "downloader", "backdoor", "bandock", "win"). It will then run these tokens against a set of tagging rules to obtain a set of standardized tags (e.g. "win" becomes "windows", generic tokens like "trojan" are removed). If a token has no tagging rule it is kept as is. Finally, the tags are expanded based on the taxonomy (e.g. "bandock" becomes "FAM:bandock", "windows" becomes "FILE:os:windows"). Based on the expanded tags, AVClass2 is able to output a list of labels with confidence scores based on how many vendors labelled the sample as such. AVClass2 work by having a default taxonomy (that can be customized), but supports unknown tags as well. This make it useful even for new malware families that it have not encountered yet [75]. At the time of writing, the AVClass2 tool has 396 stars on Github [16], 436 citations on the paper from 2016 about the original version of the tool (AVClass) [74] and 55

citations on the paper from 2020 about the current version of the tool (AVClass2) [75].

Takeaways

The purpose of this section was to gain an understanding of what malware is, and how the different kinds of malware are usually classified according to type, behavior or privilege as shown in figure 2.1. We then moved on to discuss how these different ways of classifying malware can be a challenge when trying to label malware samples, since different antivirus vendors have different ways of classifying and naming the malware. A tool like AVClass2 [75] can be used to help labelling malware, by taking for example a VirusTotal report for a malware sample with labels from multiple antivirus vendors, and then providing a standardized output of labels. The benefit of this is that the labels are based on a confidence vote according to how many antivirus vendors labelled the malware with this tag, and due to the open taxonomy the labels can be easily compared across samples. Now that we understand what malware is, we can move on to investigate how it can be analyzed, which will be discussed in section 2.2.

2.2 Dynamic Malware Analysis

In this section dynamic malware analysis will be investigated, along with the different ways to perform dynamic malware analysis.

Dynamic malware analysis means that the malware is executed in an analysis environment (such as a sandbox or an emulator), and its behavior is then observed for instance by monitoring which system calls it performs. This is opposed to static analysis, which relies on investigating the compiled binary code, without executing it [38].

A downside of relying only on static analysis for malware analysis is that the source code of the malware is often not publicly available, which means that the binary has to be reverse engineered to understand what it is doing. This is a complex task and becomes even harder if the malware is packed (which is elaborated on in section 2.3). The malware may also rely on non-static values such as the current system date and time, complicating the static analysis further. In addition to this, malware authors can mislead the static analysis on purpose by using various obfuscation techniques [38]. Dynamic malware analysis is immune to these effects, due to actually executing the malware sample (causing it to unpack itself as discussed in section 2.3) and observing its behavior, instead of relying solely on analyzing its binary code [60].

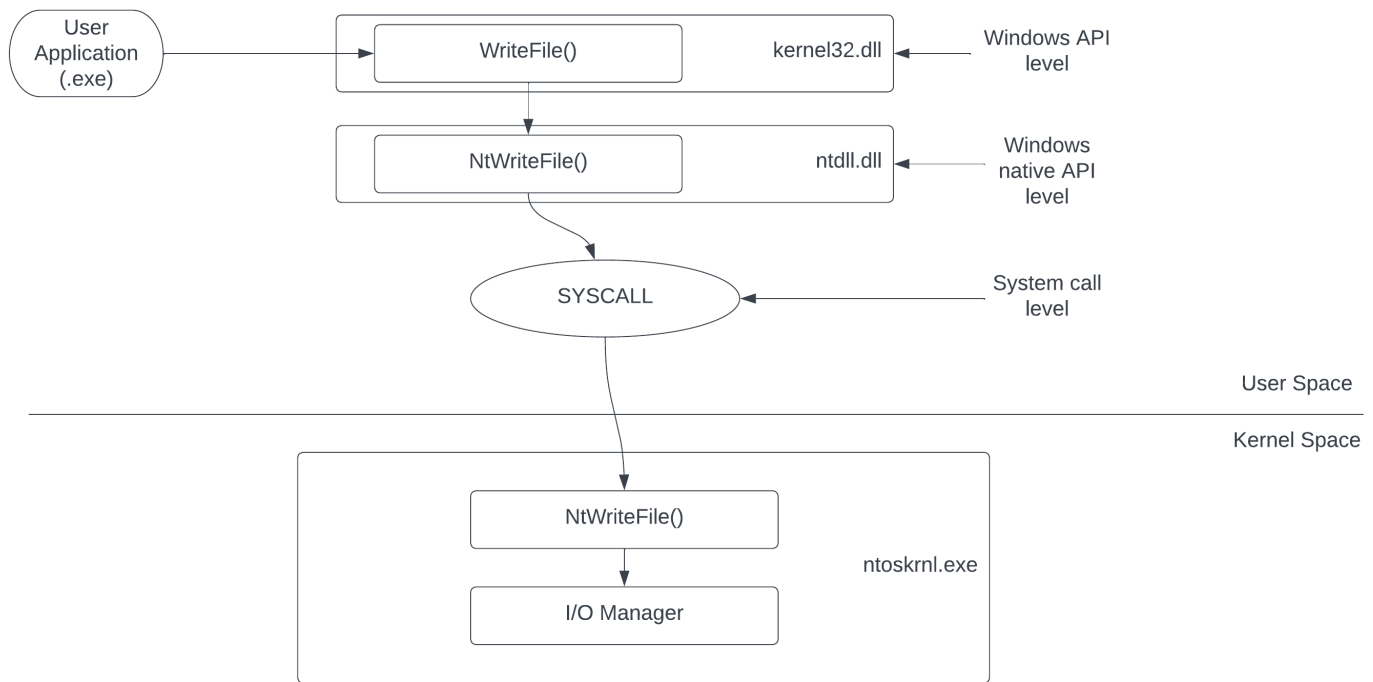


Figure 2.3: Simplified example of a Windows API call sequence (inspired by Monnappa [57]).

Malware executed in user mode can not interact directly with the hardware in a system, but instead needs to go through system calls (functions exported by other system processes) by either calling them directly or through an API. In Windows, different levels of system APIs exist. The Windows API is at the highest level, remains stable between releases and is well documented [57]. The Windows API usually calls the Windows Native API, which is not stable between releases and is not documented (at least not officially) since it is meant to be called internally by the Windows operating system itself. The Windows Native API performs a system call, which acts as the actual gateway between user and kernel mode [38, 57]. A simplified example of the Windows API call sequence is shown in figure 2.3 (inspired by Monnappa [57]). The figure shows an application executed in user mode, that wants to write a file to disk. It first calls the `WriteFile()` function in the `kernel32.dll`, which is part of the Windows API level. The `WriteFile()` function in `kernel32.dll` then calls the `NtWriteFile()` function in the `ntdll.dll`, that is part of the Windows Native API level. The `NtWriteFile()` function in the `ntdll.dll` then performs a syscall in the System call level, which then calls the actual `NtWriteFile()` function in the `ntoskrnl.exe` (kernel executable). A malicious actor may not follow the usual call sequence (show in figure 2.3), but may instead call the Windows Native Api or perform a system call directly [57].

Since malware executed in user mode needs to either call the Windows API or Windows Native API, or invoke a system call to interact with and perform actions in the infected system, most dynamic analysis tools work by monitoring and recording the the calls performed by the malware while it

is executed. By analyzing the parameters passed to these functions, it is possible to group similar calls and help the analyst in understanding what actions the malware has performed [38].

A dynamic malware analysis framework consists of the following components: The malware sample, the hardware and operating system, and the analysis tool used [60]. There are different implementation strategies of a dynamic malware analysis system. This involves the choice of whether to run the analysis component in user or kernel mode, as well as whether to perform the analysis in an emulator, VM, on bare metal or in a type 1 hypervisor. Additionally, it should be decided what level of internet access the malware should have (if any) [38].

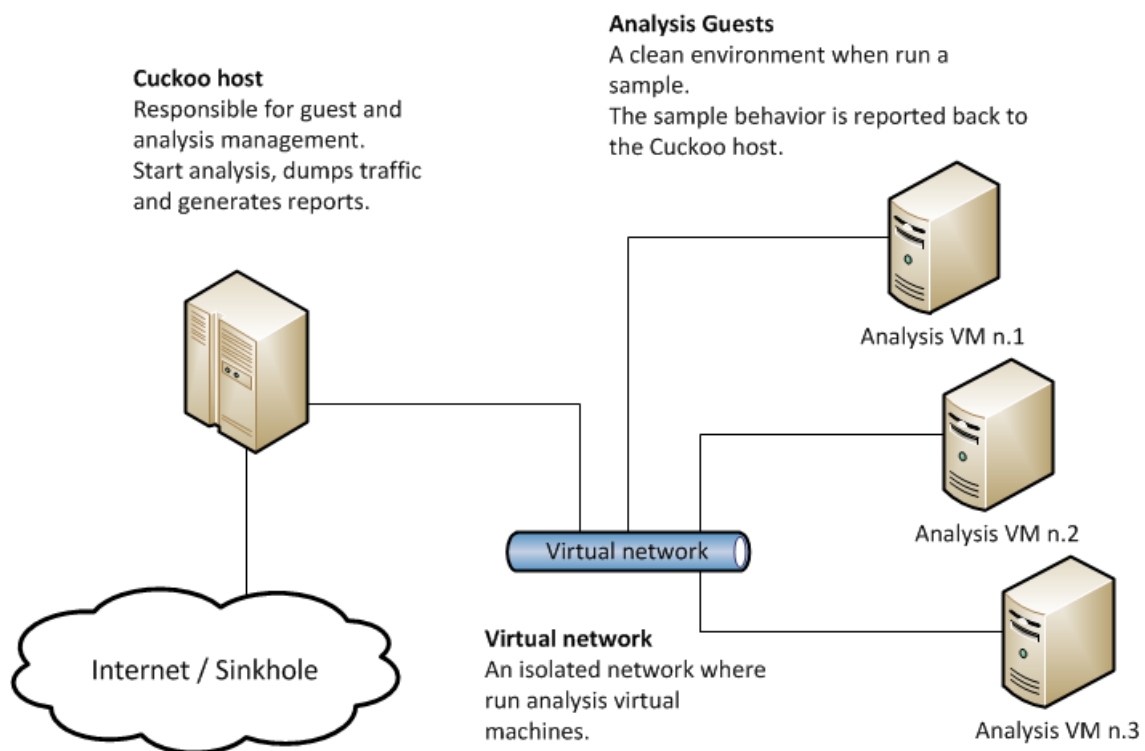


Figure 2.4: Cape sandbox architecture [21].

A common way to analyze malware dynamically is to use a sandbox. A sandbox may consist of a VM in which the malware is executed in an isolated environment, as well as a host that gathers the results and generates a report after the analysis is done. For the host to be able to see what the malware is doing inside the guest VM, a component can be deployed inside the guest VM that then reports back to the host via a virtual network. This is how both Cuckoo Sandbox and Cape Sandbox work [33, 25], as illustrated in figure 2.4

Another setup for dynamic malware analysis sandboxes is to use Virtual Machine Introspection (VMI). In this case there is no analysis component inside the guest VM itself, but rather a kernel

driver is usually deployed to the guest kernel. This driver can then be queried from outside the guest VM to get information on how the malware is affecting the system [60]. An example of a sandbox using the VMI setup is Drakvuf [51].

Takeaways

In this section the concept of dynamic malware analysis was presented, along with the benefits it has over static analysis. Different dynamic malware analysis setups were discussed, including the setup sandboxes like Cuckoo and Cape use.

2.3 Malware Evasion

In this section, malware evasion techniques deployed by malware authors to avoid analysis will be investigated. Anti malware evasion techniques used by sandboxes to counteract dynamic malware evasion will also be discussed.

Malware evasion techniques and anti evasion techniques is an ongoing battle between malware authors and malware analysts. Malware creators do not like to have their malware analyzed, since the malware rely on not being detected to carry out its job. If the malware is detected, it will usually be removed from the infected system, before it is able to perform its malicious actions. Therefore, malware authors employ different techniques for evading analysis [38]. An example of malware evasion is using a packer to obfuscate or encrypt the malware binary, making it harder to detect with static signatures, since the binary will look different when encrypted, even though the code is actually the same when unpacked. This is mostly a problem when analyzing the binary statically, since it should unpack it itself before running as usual when being executed in a dynamic analysis environment [38].

A commonly used technique to detect when a malware sample has finished its unpacking routine, is to enforce a “write xor execute” policy. This is done in three steps. First, the memory pages of the analyzed binary are marked as executable and read-only. This makes it possible to know when the binary begins to unpack itself, since it will then try to write to a memory page, resulting in a page fault that can then be caught by the analysis component. The page permissions will then be changed to read-/write-only, allowing the binary to continue the unpacking process. Once done unpacking, the malware will try to execute the unpacked binary. Doing so will result in another page fault (since the page permissions were changed to read-/write only). This indicates to the analyzing program that the malware has finished unpacking, and also gives the entrypoint for the unpacked binary. This technique will work even if a binary is packed recursively using different packer algorithms [38].

Malware evasion techniques can be broadly categorized into two main categories as suggested by Afianian et al. [3], manual dynamic analysis evasion (anti-debugging) and automated dynamic

analysis evasion (anti-sandbox). Since this project is about automated dynamic malware analysis using sandboxes, this category was explored further. The automated category consists of the following subcategories:

- Detection-Dependent
 - Fingerprinting
 - Reverse Turing Test
 - Targeted
- Detection-Independent
 - Stalling
 - Trigger-based
 - Fileless (AVT) attacks

[3]

The detection-dependent techniques mean that the malware tries to identify the environment it is running in and changes its behavior based on this detection. The three types of detection-dependent techniques, are explained below.

Fingerprinting is a common evasion technique used by malware to detect that it is running in an analysis environment, and then change its behavior accordingly to avoid revealing its malicious behavior. Fingerprinting works by the malware looking for specific artifacts (such as files, drivers, registry keys, hardware, installed applications) or other signs of being in a virtualized environment such as detecting hooks used by the analysis system [3]. The malware can also check the CPU performance counters, to check if extra CPU cycles have been spent, indicating that the malware might be under analysis [38]. Anti malware evasion techniques should be used by sandboxes to counteract fingerprinting. One way of doing this is by maintaining shadow copies of the state of for example the memory page protection settings or the EFLAGS register (that is commonly used to set the trap flag to hook into the malware's behavior while analyzing it). By maintaining shadow versions of important settings and registers, the malware analysis environment can return the values the malware expects, and thereby conceal the true values of said settings or registers, so that the malware can not use these to determine that it is currently under analysis [38]. A similar technique can be used for the Windows APIs, so if the malware for instance makes a call to the Windows API to return information about the CPU, username of the system or similar values that can be used to fingerprint the environment, the analysis system should return a random value the first time a specific value is queried, and then store the value so the same value can be returned again on subsequent calls [19].

The **Reverse Turing Test** technique is used by malware to detect human user input, such as mouse movement, clicks and keyboard strokes. If no input is detected at all, the malware assumes that it is not in a real system, since on a real system a human user would eventually interact with

the mouse or keyboard. Even if the analysis environment simulate human input, such as mouse movement, it is often still possible for malware to detect that it is not caused by a real user, for example by looking at the speed at which the mouse pointer moves [3].

The **Targeted** evasion technique is similar to fingerprinting in the sense that it tries to detect a specific environment. However, as opposed to fingerprinting where the intent is to detect and evade an analysis environment, the intent of targeted attacks is to detect a specific target and only execute the malicious behavior if that target is present. The first known malware to use this technique is Stuxnet, that spread like a worm but stayed dormant until a specific industrial control system was reached [3].

The detection-independent techniques are not dependent on the malware detecting the environment it is running in, since these type of evasion techniques will be employed by the malware no matter what environment it is running in. The 3 types of detection-independent techniques are explained below.

The **Stalling** technique tries to exploit the fact that automated analysis is usually limited by time (eg. the analysis stops after 5 minutes). To exploit this fact, some malware authors made their malware sleep for a long period of time to avoid analysis. Sandbox developers mitigated this by skipping these long periods of sleep, however, this resulted in malware developers finding more advanced stalling techniques, such as checking if execution has been accelerated by sleep skipping. Another form of stalling is to perform operations that take a long period of time, such as writing huge amounts of random data to memory, or encrypt a payload with a weak encryption key and then brute-force it while executing. These active forms of stalling has the benefit that the malware never actually sleeps, making it harder to detect that it is doing something irrelevant to stall the time. Furthermore, the analysis system may run out of resources (such as memory) while performing these stalling operations, which may make the analysis system unresponsive [3].

Trigger-based techniques, also commonly referred to as "logic bombs" is a way for malware to stay unnoticed until a certain event triggers them. Logic bombs can be both time based (ie. the malware will only execute once the system reaches a specific time/date), or the trigger can be network-based, such as waiting for a command from a Command and Control (C&C) server. The command from the C&C server can even contain the encryption key to unpack the malware. Other events, such as the user pressing a given number of keys, can also be used to trigger the malware. For this reason, malware sandboxes should also emulate user input (such as keyboard presses and mouse movement/clicks) [38, 3].

Fileless Malware is a technique where an exploit in the operating system or browser is used to inject the malware payload directly into the memory of the victim system, without the use of a file. The analysis evasion in this technique lay in the fact that fileless malware is often not exposed to dynamic analysis, since most sandboxes involve submitting a file to analyze [3].

Programs have been developed to show how malware can succeed in evading an analysis environment, such as anticuckoo [9] that is able to utilize the fingerprinting technique to detect the hooks used by Cuckoo. Pafish [61] is another application that can detect whether it is under analysis by using both fingerprinting and reverse turing tests (such as mouse movement). Al-Khaser [4] is similar to Pafish but check for even more evasion techniques.

Takeaways

Different malware evasion techniques as well as anti malware evasion techniques were discussed in this section. A framework for classifying malware evasion behavior in automated dynamic analysis environments by Afianian et al. [3] were presented. It is crucial for a malware analysis sandbox to implement anti malware evasion strategies, to trick the evading malware to run and exhibit its true malicious behavior even while under analysis. If the malware manages to detect and evade the analysis environment, it will not show its true malicious intents, which would lead to misleading results in the analysis report generated by the sandbox. Programs like anticuckoo [9], Pafish [61] and Al-Khaser [4] can be useful to test whether an analysis environment (such as a sandbox) is detectable by some of the techniques commonly used by malware to evade analysis.

2.4 Online Public Malware Sandboxes

In this section, studies that has used data from online public malware sandboxes will be investigated. Furthermore, open source and commercial public online sandboxes will be discussed. Finally, the privacy of the data present in online malware sandboxes will be looked into.

Only a few studies have investigated the data collected by public malware dynamic analysis sandboxes. The work by Graziano et al. [41], shows how the data generated by public malware sandboxes can be used for analyzing the malware development process done by malware authors. In the experiment they used samples collected from the Anubis sandbox [13] from 2006 to 2013. Based on the premises that the sample must have been submitted to the sandbox before being observed in a real system, and that it had to be manually submitted by an individual user, the data set was reduced to only include relevant samples.

The initial data set of around 32 million samples, was filtered down to around 520,000 by only looking at the samples submitted by individual users, and removing samples that were part of a previous batch submission. The samples were then filtered further by first checking whether they were known by Symantec, and then Virustotal. This process reduced the data set to around 214,000. Finally, non-executable files and packed binaries were removed from the data set. The authors state that even though most of the features would work on packed binaries as well, they decided to remove the packed binaries to make it easier to verify and double check their results. This resulted in a final data set of 121,856 samples submitted by 68,250 distinct IP addresses.

The samples were then clustered based on similarities in the identified features. Based on the clus-

tering of similar malware samples, and by looking at the compile and submission time of similar samples, the authors were able to identify timelines of development of a particular malware sample.

Interestingly, the study [41] showed that multiple targeted malware attacks (including Stuxnet) were submitted to the sandbox months (some even years) before being used for an attack and observed in the wild. This indicates that the authors of the different malware samples used the public sandbox to develop and test their samples (for example while implementing evasion techniques), a long time before releasing the malware into the wild. Had researchers been aware of this, they would have had better time to analyze and come up with defensive strategies for these particular malware families.

2.4.1 Existing Online Public Malware Sandboxes

Cuckoo Sandbox was investigated, due to it being open-source and well known within the academic community. The newest version of Cuckoo Sandbox (2.07) was released in June 2019, and it has not been actively maintained since [31]. In spite of this, Cuckoo still seems to be widely used by the research community. This is likely a problem due to the malware evasion techniques constantly evolving (as discussed in section 2.3), which make the results generated by Cuckoo at risk of being partially invalid if some malware samples manages to evade the sandbox.

Cape sandbox [25] is another open source sandbox that is originally forked from Cuckoo and adds (among other features) anti-evasion techniques to the sandbox. Additionally, it is still actively maintained (at the time of writing) [25].

A Google Scholar search for "cuckoo sandbox malware analysis" for papers published from 2020 onwards, returns 1180 results (as of 28/03-2023). A similar search for "cape sandbox malware analysis" returns 64 results. This is an indication that Cuckoo is still widely preferred by researchers. Looking at how many samples have been submitted to the public online version of Cuckoo [35] over a 24 hour period (22/03-2023 14:40 - 23/03-2023 14:40), the number is around 3660, whereas for Cape [23] that same number is 360, meaning that the online instance of Cuckoo had 10x as many samples submitted for analysis in the time period than Cape. This further points to Cuckoo still being the preferred sandbox of choice for many researchers. A possible explanation for this is that the online version of Cape requires a user account to be activated by the website admin, before it is possible to upload samples to the sandbox.

Public online sandboxes generate a lot of publicly available malware behavioral data in the form of reports, but the data is not in a format that is easily available to perform further analysis on, since the analyst has to look through or download reports for each sample manually. Therefore, a lot of potentially valuable data and insight from these reports may not be utilized to its full extend.

Commercial Online Public Malware Sandboxes

Commercial public sandboxes such as JoeSandbox [47], any.run [14], tria.ge [81], Intezer Analyze [46] and Hybrid Analysis [44] were also considered, however, they all had terms of service (ToS) stating that all data published on their websites are copyrighted and may not be published by others. A study by Nappa et al. used data from commercial sandboxes, but had to anonymize the data and the sandbox to avoid violating any ToS [59]. Anonymizing the data from commercial sandboxes were also considered for this study, to be able to compare reports from more sandboxes. Especially reports from JoeSandbox could have been interesting to include in this project, since they offer bare-metal dynamic analysis of malware, which make the analysis environment close to impossible to detect and thereby evade by malware. However, including data from commercial sandboxes would compromise the reproducibility of the study (since the data and sandbox would have to be anonymized), and it was therefore chosen to stick with open source sandboxes (e.g. Cuckoo and Cape).

2.4.2 Online Public Malware Sandbox Data Privacy

Weathersby did a recent study [86] on an online public malware analysis service to check whether benign PDF files uploaded to the service contained Personally Identifiable Information (PII). For example, one could imagine that a user receives a PDF file that supposedly comes from their doctor (e.g. blood sample results), or a letter from a lawyer stating that they need to go to court. The user wants to make sure that the PDF file is benign and does not contain any malware. The file is therefore uploaded to an online public malware analysis service to analyze it. The result of the analysis may show that the file is benign. However, since the analysis report (including the PDF file) is now publicly available at the malware analysis service, the user may unknowingly have shared personal information about themselves such as their name, Social Security Number, address, phone number, or other information like that they have to go court at a specific date. The study by Weathersby [86] found that 81% of the analyzed files contained some sort of PII, however, the majority of this information was in the form of the author name in the PDF metadata. It was therefore concluded that the impact of the presence of PII in online malware sandboxes is limited, since only a small percentage of the analyzed samples contained sensitive information beyond the authors' names [86].

Takeaways

There is not a lot of studies that have used the behavioral reports generated by public malware sandboxes in their analysis. This may be due to the cumbersome process involved with downloading a large amount of reports, and then transforming those reports into a data set that can be used for malware analysis. Graziano et al. [41] showed one way of using data from a public malware sandbox to detect patterns in the malware development process of malware authors. Regarding malware sandboxes, it was found that Cuckoo Sandbox still seem to be widely used by researchers, even though it has not been actively maintained since 2019, which may be a problem in terms of

Name	Year Published	Size	Behavioral Features
EMBER2018 [8]	2018	1,000,000	No
MOTIF [48]	2022	3,095	No
Malrec [76]	2018	66,301	Yes
Mal-API-2019 [27]	2019	7,107	Yes
RanSAP [42]	2022	12	Yes
Behavioural Reports of Multi-Stage Malware [26]	2023	8,087	Yes
Avast-CTU Public CAPE Dataset [18]	2022	48,976	Yes
A Quest for Best [56]	2019	273	Yes

Table 2.1: Recently published malware data sets.

malware evasion. An actively maintained fork of Cuckoo Sandbox that introduces anti evasion techniques is Cape Sandbox. Investigating whether Cuckoo Sandbox is evaded more often than Cape Sandbox could be interesting, to see whether researchers should move on to use Cape instead of Cuckoo to minimize the amount of malware samples getting evaded in their analysis. Finally, a study by Weathersby [86] investigated whether PII is present in benign files uploaded to online malware sandboxes and found that the presence of sensitive personal information is limited, and mainly consists of the file authors' names.

2.5 Malware Data Sets

In this section the state of existing malware data sets will be investigated.

Data sets are essential for malware analysts to develop mitigations for new malware threats. Especially in terms of training and evaluating machine learning algorithms against malware. There is a lack of public malware data sets containing behavioral features [84, 7, 52, 80, 55]. Malware data sets should be kept up to date with new samples to stay relevant [87, 55, 82, 49]. Recent/updated data sets are essential for the research community to be able to keep up with the quick pace of malware development [52].

Table 2.1 shows recently published malware data sets, that were found as part of reviewing the existing literature. Out of the data sets, Ember is by far the biggest with 1 million samples in total. The Ember data set is based on extracting static features from Portable Executable (PE) files. The authors of Ember also published source code that makes it easy for others to add new samples to the data set, thereby making sure it can be updated to stay relevant [8]. MOTIF is made by manually labelling around 3000 malware samples with ground truth labels based on reports from large security vendors, and would therefore be very time consuming to update and add new samples to (especially at a large scale). Malrec, Mal-API-2019, Behavioral Reports of Multi-Stage Malware, Avast-CTU Public CAPE Dataset and A Quest for Best [76, 27, 26, 18, 56] requires setting up a sandbox locally to execute samples and add new data based on the analysis performed which can also be very time consuming. RanSAP [42] is focused on Ransomware specifically and involves

setting up a type 1 hypervisor with an unpublished function, which makes it unfeasible for other researchers to do. To summarize, most of the recently published malware data sets containing behavioral features are either small or time and resource demanding to keep up to date.

2.5.1 Malware Data Set Quality

In the survey on dynamic malware analysis by Omir et al. [60], the data sets used in the surveyed studies are compared. The comparison shows that most of the studies used small data sets for evaluating their findings. 14 out of 29 studies used data sets with under 100 samples in total (malware and benign). The two biggest data sets used had around 115.000 and 25.000 samples respectively (all malicious), whereas the data sets in the rest of the studies had less than 7000 samples. The authors state that ideally benign files should be included in the malware data sets in a ratio close to reality, to make it possible to evaluate the false-positive rate in projects using machine learning algorithms. They also state that most of the surveyed papers did not make their source code open source, which they should do to enable the research community to benefit from their findings [60]. Another survey on malware classification by Abusitta et al. [2] shows that 15 of the surveyed studies were not scalable due to using very small data sets. Smith et al. reviewed existing open-source malware data sets used in machine learning, and found that there is a lack of behavioral information (e.g. behavioral signatures) in the malware data sets used in machine learning [80]. There is also a lack of recent behavioral data sets for ransomware analysis, even though recent data sets are essential, since the quick development of the technology make old data sets irrelevant (due to new threats continuously evolving) [52]. Missing up to date malware data sets is also an issue for training machine learning classifiers used in Intrusion Detection Systems (IDSs). Most of the IDSs are built on machine learning classifiers trained using data sets from 1999, since no newer acceptable alternatives are publicly available [49].

Takeaways

In this section recent and previous malware data sets were investigated, and it was found that most have the issues of being either small and/or very time consuming to keep up to date. Several studies suggest that there is a lack of recent/updated public malware data sets containing behavioral features for use in machine learning and malware analysis.

2.6 Problem Statement

How can the lack of recent malware data sets containing behavioral features be addressed, and how can it be investigated whether the lack of updates to Cuckoo Sandbox is a problem?

Based on the problem analysis performed in this chapter, it was found that there is a lack of updated malware data sets containing behavioral features. Many studies do not publish their data sets, and the data sets that do get published are hard to add new malware samples to, to keep them up to date. Therefore, the first aim of this project is to publish a malware data set with behavioral

features, that can easily be updated to add data for new samples at a later point in time.

Another problem identified during the problem analysis, was that Cuckoo Sandbox still seem to be the preferred sandbox within the academic community. This may be a problem, since it has not been actively maintained since July 2019, and may therefore likely be evaded by malware more often than a sandbox with anti evasion techniques in place that are actively updated. The second aim of this project is therefore to investigate whether Cuckoo Sandbox is being evaded more often than Cape Sandbox by comparing analysis reports for the same samples. The objective of this is to investigate whether researchers should consider using another sandbox than Cuckoo when analyzing malware dynamically, to avoid their results being partially wrong due to malware evading analysis.

To help guide the research, the project aims to answer the following research questions:

- *RQ1: How can a behavioral malware data set be created, that can be used for future research?*
- *RQ2: How can a behavioral malware data set be continuously updated in an easy way?*
- *RQ3: How can it be investigated whether Cuckoo Sandbox is evaded more often by malware compared to an actively maintained sandbox, such as Cape?*

Chapter 3

Methodology and Design

This chapter discusses the methodology and design choices made to carry out this project. It will start off more broadly, exploring the different possible ways to approach the work and answer the research questions formulated in section 2.6. It will then get more specific into which approaches was chosen, what design choices were made and why.

First, a problem analysis was conducted in chapter 2 to investigate the literature within the area of dynamic malware analysis, and to find out what challenges could benefit from further analysis. The problem analysis was done by reviewing existing literature based on the areas defined in the initial problem statement in section 1.1. Google Scholar was linked with the Aalborg University Library (AUB) and used to find literature [40]. Keywords such as "dynamic malware analysis", "dynamic malware evasion" and "online malware sandbox" were used, and the literature was deemed relevant based on its title, abstract and conclusion. The age of the publication was also considered, as in some cases newer, updated work was available. The number of citations a paper had was used as an indicator of the paper's impact in the field (ie. papers that were cited more times, were deemed more impactful and therefore prioritized higher). Several iterations of finding literature was done, as an attempt to find the most relevant literature for this project. In some cases, papers were also found by being mentioned in other papers (such as in a survey).

3.1 Lack of data sets

As mentioned in the problem statement (see section 2.6), there is a need for updated data sets containing behavioral features within the malware research community. Because of this, many studies produce their own data sets, but they are often quite small and do not always get published. When a large data set do get published, like Malrec [76], it soon becomes obsolete due to the fast pace of malware development. Therefore, one of the research aims of this study is to provide a large data set, that can easily be updated when new data is needed. This is similar to what Ember [8] did for static malware analysis. The authors provided a large data set (based on 1 million samples), as well as the source code used to extract features from malware samples, enabling other researchers to add

new data as needed.

To create a data set containing behavioral information of malware (and benign) samples, different approaches exist. One approach would be to generate the data yourself by executing and analyzing the malware samples either in a local (private) sandbox (like Ijaz et al. [45]) or in a public online sandbox, and then gather the results into a data set. This approach has the benefit that you have full control over what data goes in to the data set, e.g. what malware samples are analyzed, and if benign samples are analyzed as well (and if so, what the balance between malware/benign samples should be). In the case of a local sandbox, we control what gets analyzed in the sandbox, and can therefore be sure which samples are benign and which are not (provided that the samples are labelled beforehand).

The downsides of this approach is that it requires a lot of time and resources. First of, the samples to analyze would have to be selected and gathered from somewhere. Then the sandbox would have to be set up and configured (if running it locally), which is not a trivial task and can be quite difficult to get right. Making sure the sandbox is configured correctly with proper anti evasion techniques can also be hard. Finally each sample would have to be ran and analyzed in the sandbox, before processing the results. This would also mean that for other researchers to add new samples to the data set, they would have to go through the same time consuming process. Using an online public malware sandbox would get rid of some of this work, since the sandbox is already set up and usually configured with (at least some) anti evasion techniques. While using a public online sandbox, we do not have full control over what gets uploaded and analyzed (since others can upload samples to the sandbox as well), but we can still control what goes into the data set by only including the samples we uploaded to the sandbox ourselves.

While it is possible to generate new malware behavioral data for a data set ourselves as mentioned above, another possibility is to use existing data. There are multiple ways of doing this. One way is to download previously published data sets by other researchers and combine this into a larger data set. The advantage of this approach would be that it is fairly easy to do, and does not require much time or resources, since the hard work of gathering and processing the data has already been done by others. On the other hand, this approach would not really bring anything new to the table, since the published data set would consist of a combination of other publicly available data sets. Also, it would not be easy to keep up to date, since it would depend on other authors publishing new data to include in the data set. Additionally, going with this approach would mean less control regarding what the data set contains and the balance of different malware families as well as benign samples.

Another approach of using existing malware behavioral data would be to find a way to utilize the analysis reports that are generated and made publicly available by online malware sandboxes like Cuckoo [35] and Cape Sandbox [25]. This approach would enable us to utilize the large amount of public reports, where resources and time have already been spent on analyzing a lot of samples, and turn those into a published data set to make it easier for the research community to use in their

studies. Using a website's API to download data would often be the preferred approach, since it is often easier to get the data you want in a useable format, and at the same time you can be fairly certain that you interact with the web server in the way intended by the developers. Many websites also make use of API keys to monitor and limit the usage of their API, to make sure the server is able to handle all the requests without getting overwhelmed. However, not all websites have APIs, or want to expose them publicly, or they may not contain the information you need.

Another approach to obtain data for our own data set from a web site is by scraping it [88]. This involves creating a small program, called a spider, that will then crawl the web site and extract the needed information from it. The downside of this approach is that it can be slow, compared to using an API, since additional data will often need to be downloaded, as the spider would interact with the web server as if it was a real user browsing the web site. On the other hand APIs may be rate limited, in which case using the API might actually be slower. Also, to extract the correct data, a spider has to be written specifically for one particular website, since the structure of the data for most web sites is unique. For instance to scrape elements from a website we might look for specific HTML ids or classes, which may be something like "cuckoo.result.malware.1" for Cuckoo while for the same data on Cape it could be something like "cape.result.malware23.1". The structure of the data may also change at any point in time, which will result in breaking the spider until it has been updated to handle the new structure [50]. The good thing is that scraping enables us to gather exactly the data we need, as long as it is publicly available on the website. This means that if we are able to find the data via a web browser, it should technically be possible to create a spider to extract that data for us. There are a few caveats to this though, as protection mechanisms against web crawlers exist, such as rate limiting and user agent checks. This will be elaborated on in chapter 4.

For this study I chose to use web scraping to gather data, since it enabled me to answer two of my research questions (RQ1 and RQ2) defined in section 2.6 regarding creating and publishing a large data set for further research that can easily be updated at a later point in time to include new samples. As discussed above, none of the other solutions seemed viable for this, as they are hard to keep up to date with new samples. Also, since a lot of resources have already been spent on analyzing a lot of different malware samples in these sandboxes, it made sense to see if this existing data could be utilized in a bigger context, and made more easily available for researchers needing big data sets for their research experiments. Finally, I found it interesting to explore this way of gathering data and the challenges and considerations that comes with it, which is elaborated upon in chapter 4.

3.2 Cuckoo Sandbox Evasion

Another problem that was identified during the problem analysis is that it appears that Cuckoo Sandbox is still being used a lot in behavioral analysis of malware samples even though it has not been actively maintained since June 2019. This is very likely a problem since a lot of malware has implemented evasion techniques, to make them able to evade analysis by sandboxes like Cuckoo. Since the developers of malware come up with new evasion techniques in a continuous manner, the

sandboxes used to analyze said malware need to implement and update anti evasion techniques continuously to trick the malware into running even though it is being analyzed. If malware samples manage to evade the analysis environment successfully, the results obtained from the analysis will only be partially valid, which put the entire research project at risk of leading to false conclusions. Due to the risk of incomplete results, researchers should be aware of how they use the data set. Incomplete results (due to malware evasion) could lead to even more incomplete research if other researchers blindly use and rely on the data set. Another aim of this research project is therefore to investigate whether using an actively maintained open-source sandbox like Cape, leads to more complete/correct results than using Cuckoo, due to better/more up to date handling of malware evasion. In other words, the hypotheses is that malware may succeed in escaping Cuckoo Sandbox more often than Cape Sandbox.

It is hard to find a measure that shows with certainty whether a malware sample managed to escape analysis or not, but behavioral signatures can be used as an indicator. Behavioral signatures are applied to the analysis results by both Cuckoo and Cape sandbox. These signatures help simplify the analysis by identifying known behavioral patterns, such as network activity, file activity, registry key activity or evasion behaviors. More specific signatures also exist, such as whether the malware exhibits known behavior of ransomware, keyloggers, cryptominers, etc. In principle, more signatures identified, should at least mean that more behavior/activity was identified from the sample [17]. So if one sandbox for example only reports 2 signatures, and the other sandbox reports 35 signatures for the exact same sample, it would indicate that the malware managed to escape the first sandbox, at least partially. There is of course also the possibility that it manages to escape either both sandboxes or neither, in which case it may be hard to conclude anything. Instead of simply trying to evade analysis, there is also the possibility that some malware may trigger fake signatures to confuse the malware analysis. So if the malware detects that it is in an analysis environment, instead of sleeping, doing nothing or crashing the process (which would normally happen in malware evasion), it can change its behavior and perform other actions than its true malicious intent to trick the analysis into thinking it is benign [17]. Though signatures are not a complete measure of whether the malware managed to evade one or both sandboxes as discussed above, it should give an indication as to whether there's a pattern in one sandbox identifying more behaviors than the other and thereby probably succeeding in analyzing evading malware more often. Therefore, I chose to compare signatures between the two sandboxes, both on a general level and with specific examples.

Another way to check how good a sandbox is at avoiding being detected (and thereby evaded) by malware, is to use programs like Pafish [61], Al-Khaser [4] and anticuckoo [9] as discussed in section 2.3. This will be done to give another indication of the sandboxes' anti-evasion techniques.

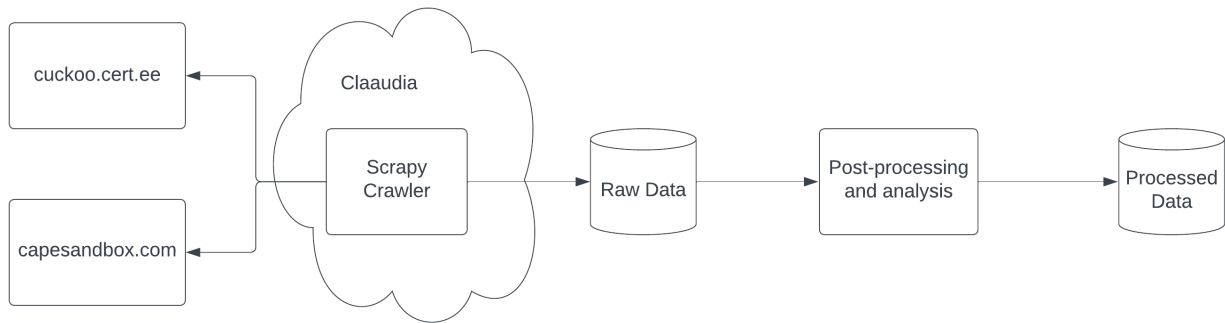


Figure 3.1: Overview of architecture.

3.3 Data Collection

The overall architecture of the data collection method is shown in figure 3.1. The open source python scraping framework, Scrapy [72], was used to gather data from Cuckoo and Cape sandbox. Scrapy was chosen as a tool due to it being an open source Python framework with a large community and great documentation [71]. It is fairly easy to get started with, but at the same time very customizable. Scrapy can be used both for making the requests to the website to get the unstructured data, as well as (optionally) handling the parsing to structured data. Scrapy supports both built-in and custom middleware, which makes it easy to handle things like scheduling, cookies, headers, redirects, retries etc..

Another common approach to scraping is to use the requests module for Python to handle the actual http requests to get the unstructured data from the web server and then use a parsing library like BeautifulSoup to parse it. This approach would have meant additional work in the form of handling scheduling of requests, download delays, retries, etc. manually, with no real benefit of doing so. Scrapy can also be run with a state folder, which makes it possible for it to keep track of which requests have already been made. This is handy if the crawling session gets interrupted for whatever reason (server shuts down unexpectedly, an exception is raised etc.), since the session can then easily be resumed without having to crawl the same pages twice. Since performance is already very fast with Scrapy, and since I imposed a limit on the crawling rate myself to avoid overloading the servers, there were no issues regarding performance either. Another programming language could have been used as well, but since I was familiar with Python already, and provided it has a very large active community with lots of documentation and examples, it made sense to stick with that. Overall, while choosing another tool for scraping was certainly a possibility, I do not believe it would have made a difference to the end results, since the gathered data would be the same.

3.4 Publishing Data Set and Source Code

The data set gathered during this study will be published along with the source code used to gather and process it. Both the "raw data" (e.g. the full Cuckoo reports in .json, and the HTML pages from Cape), as well as the processed data containing the sample sha256 hash, the url the data was obtained from (including the analysis task id), the date and time of when the analysis was completed as well as the behavioral signatures identified for the sample. This processed data set is a lot smaller than the raw data set and therefore may be easier to work with. However, the downside is of course that not all the data is present. This is why the raw data is published as well. In terms of the data from Cuckoo this can be used directly, since it is already in .json format, whereas for Cape the HTML will have to be processed first (as discussed in chapter 4). If researchers want to use the reports for machine learning, an approach like the one presented by Darshan et al. [37] can be used, where the Cuckoo json reports are converted to a machine learning friendly format. Utility scripts are published as part of this project as well to help with processing the raw data. This way, if features that are not in the processed data set is needed, it can be added by modifying the source code of the scripts to include it, and processing the data again. This can be done without having to crawl the websites again, as long as the needed data is in the published raw data.

To add new data in the form of new analysis reports to the data set, the spiders can be run again at a later point in time. A list of already crawled pages will be published. This list can be used with the Scrapy crawler to only crawl analysis reports for new malware samples that it has not scraped data for already. This is elaborated upon in chapter 4.

3.5 Summary

In this chapter the methodology and design of this project was discussed. This was done by first discussing the problems identified in the problem analysis and which approaches exist to solve them. For answering RQ1 and RQ2 in this project I chose to create a data set based on existing malware behavior reports from online public malware sandboxes obtained by web scraping, since this allows the data set to be easily updated in the future by running the scraper again when needed. To answer RQ3, the amount of behavioral signatures identified by Cuckoo and Cape sandbox respectively will be compared, to see if Cape consistently identify more signatures, which would indicate that Cuckoo is evaded more often.

Chapter 4

Implementation

In this chapter the implementation of the solution proposed in chapter 3 will be presented. First, the code made for collecting data for the data set will be discussed, then I will talk about the ethical considerations to keep in mind when implementing a scraper, and finally the scripts used to analyze the results will be presented.

The full source code made during the implementation can be downloaded at: <https://github.com/Villefrance/malware-web-scraper>

The data was collected by scraping publicly available behavioral malware analysis reports from two open-source online malware analysis sandboxes, Cuckoo and Cape sandbox.

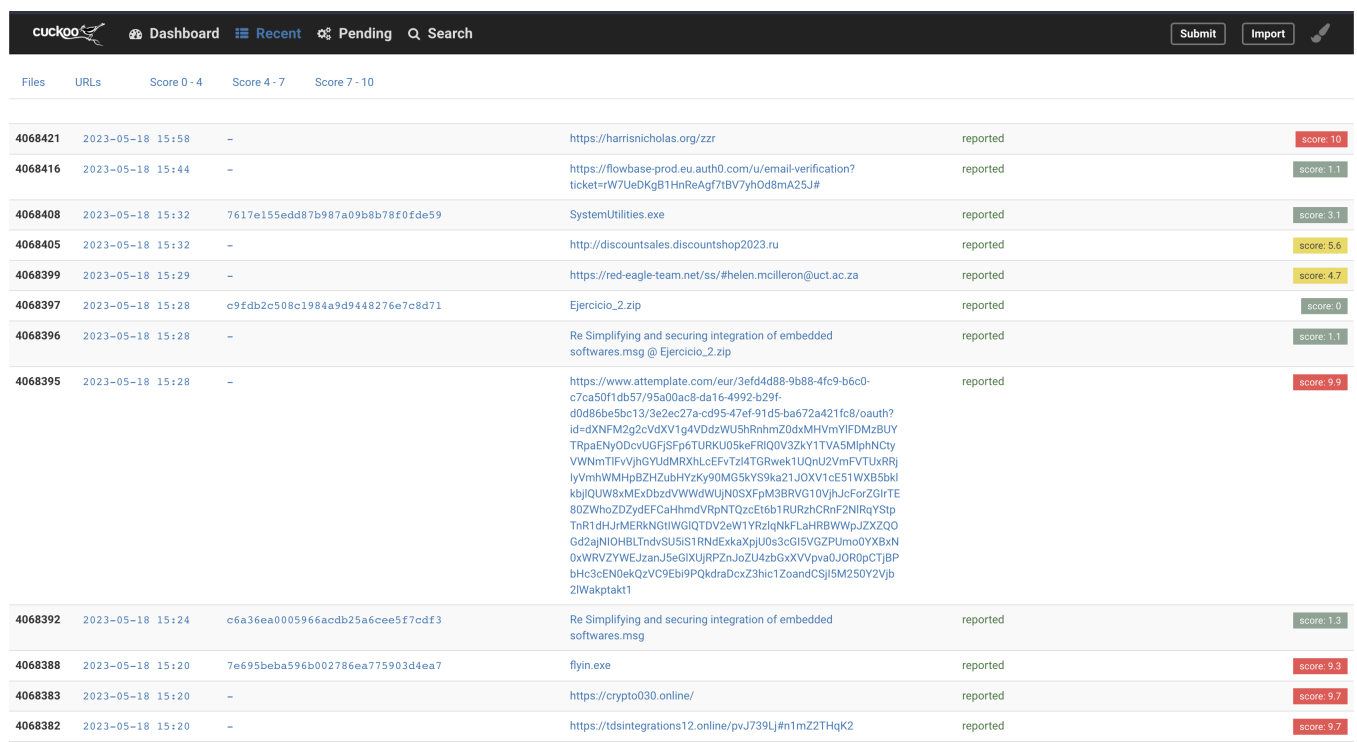
The scraping was done by writing spiders for each website, to handle the differences in which data is structured on the two sites. For Cuckoo it was possible to download the full analysis report as .json, whereas for Cape this required an active user account, which I did not succeed in obtaining for this study. I tried to create a user to get access to this feature as well as their API, but was met with the message that the admin needs to activate new user accounts manually. Looking at their Github repo reveals that the way to ask for user account activation is to ping them on Twitter [20]. I did this, but did not receive a response, nor get my account activated (as of this writing). Looking at previous comments from other users, it seems that they used to respond fairly quickly and activate peoples accounts within a couple of days, but have since stopped responding to such requests (at least for the time being). This left me with the option of scraping the publicly available HTML data instead, and parse the relevant information to .json. Therefore, the data from Cuckoo contains more information, since it was possible to download the full analysis reports directly as .json.

The crawler was run on a Virtual Machine (VM) in the research cloud environment at Aalborg University, Claudia [29]. This had the benefit of being able to run the crawler from a research network. Additionally, the VM had 32 vCPU cores and 256GB RAM, which was especially useful during the post-processing and analysis of the gathered data.

4.1 Scraping Cuckoo

To be able to scrape Cuckoo [35], a few things had to be kept in mind. First, to even get a response from the server, the user-agent needed to be set to something looking like a real browser. In this project the user agent was set to **"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/111.0.0.0 Safari/537.36"**, which resembles the most recent version of the Google Chrome browser on Windows at the time of writing. The user agent was chosen due to being the most common at the time of writing [58].

4.1.1 Cuckoo Analysis Report Structure



Task ID	Timestamp	Score	URL	Status
4068421	2023-05-18 15:58	-	https://harrisnicholas.org/zzr	reported
4068416	2023-05-18 15:44	-	https://flowbase-prod.eu.auth0.com/u/email-verification?ticket=rW7UeDKgB1HnReAg7f7bV7yH0d8mAZ5J#	reported
4068408	2023-05-18 15:32	7617e155edd87b987a09b8b78f0fde59	SystemUtilities.exe	reported
4068405	2023-05-18 15:32	-	http://discountsales.discountshop2023.ru	reported
4068399	2023-05-18 15:29	-	https://red-eagle-team.net/ss/#helen.mclleron@uct.ac.za	reported
4068397	2023-05-18 15:28	c9fdb2c508c1984a9d9448276e7c8d71	Ejercicio_2.zip	reported
4068396	2023-05-18 15:28	-	Re Simplifying and securing integration of embedded softwares.msg @ Ejercicio_2.zip	reported
4068395	2023-05-18 15:28	-	https://www.attempte.com/eur/3efd4d88-9b88-4fc9-b6c0-c7ca50f1db57/95a00ac8-da16-4992-b29f-d0d86be5bc13/3e2ec27a-cd95-47ef-91d5-ba672a421fc8/oauth?id=dXNFM2g2cVdXV1g4VDdzWU5hRnhmZ0dxMHVmfYFDmzBUYTRpaENyODcvUGFJSFp6TURKU05keFRlQ0V3ZkY1TVAS5MlphNCtyVWNmTlFvVjhGYUdMRXhLcEFvTzI4TGdwek1UQnU2VmFVTUxRRjlyVmhWMHhpbBZHZubHYzKy90MG5kYS9ka21J0XV1cE51WXB5bklkbjQUW8xMExDbzdVWVWdWUjN0SXFpM3BRVG10VjhJcForZGlrTE80ZWhoZDZydEFCaHmdVRpNTQzcEt6b1RURzhCRnf2NlRqYStpTnR1dHJrMERkNGIiWG1QTDV2eW1YRzlkNkFLaHRBWwPjJXZQOGd2ajNlOHBLTndvSU5iS1RNdExkaXpU0s3cGI5VGZPUmo0YXBxN0xWRVZYWEJzanJ5eG1UjRPNzJoZU4zbXVpva0J0R0pCTJBPbHc3cEN0ekQzVC9Ebi9PQkdraDcxZ3hic1ZoaandCSjI5M2Y0Y2Vjbi2lWakptakt1	reported
4068392	2023-05-18 15:24	c6a36ea0005966acdb25a6cee5f7cdf3	Re Simplifying and securing integration of embedded softwares.msg	reported
4068388	2023-05-18 15:20	7e695beba596b002786ea775903d4ea7	flyin.exe	reported
4068383	2023-05-18 15:20	-	https://crypto030.online/	reported
4068382	2023-05-18 15:20	-	https://tdsintegrations12.online/pvJ739Lj#n1m2ZTHqK2	reported

Figure 4.1: Cuckoo recent analysis page.

To scrape the data we need from Cuckoo [35], we first need to figure out how the data is presented and structured on the website. The "Recent" tab in the menu bar leads to **/analysis**, and shows recently completed analysis tasks. Figure 4.1 shows the analysis task ids to the left, and it looks like they are incremental, since the bottom task has the lowest id (4068382) and the most recently completed task has the highest id (4068421). Some ids are skipped, for example it goes from id 4068383 to 4068388, which could be due to the analysis' for the ids in between not being done yet or they may have failed to run at all. Still, the general pattern is that the id increases for each new analysis task. Clicking on a specific analysis task, reveals that the url structure includes the analysis

id: `/analysis/4068408/summary`

The following subsections will discuss how behavioral malware data was scraped from Cuckoo Sandbox [35]. An overview of the data flow can be seen in figure 4.2.

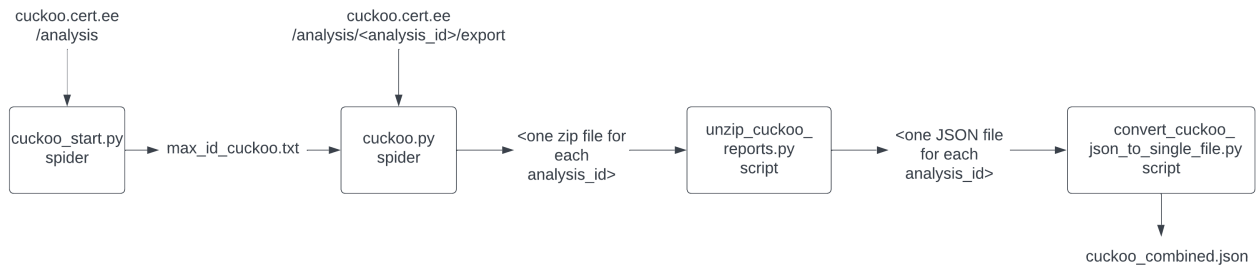


Figure 4.2: Cuckoo data flow.

4.1.2 Cuckoo Start Spider

Since the ids seem to be increasing for each new analysis task and the analysis id is included in the url (as discussed in the previous sub-section), it should be possible to fetch the id from the most recently completed task (which we suppose is the largest id, ie. the max id), and use this as a starting point when fetching the analysis reports from Cuckoo. The max id can then be decreased by 1 at a time to get previous analysis tasks.

```

255 <script>
256
257 $(document).ready(function() {
258
259     // only do this if we are on the recent page
260     if(!$('#table#recent').length) return;
261
262     var recent;
263
264     $('body').on('click', 'div.nav_container>div', function() {
265
266         var btn_id = $(this).find("a").attr("data-filter");
267         var btn_active = $(this).attr('class');
268         var btn_categories = ["cat_files", "cat_urls"];
269
270         if($('#a[data-filter^="score"]', this).length) {
271             var score_choice = $(this).find('a').attr('data-filter');
272
273             $('.nav_container>div#item>a[data-filter^="score"]').each(function(i, obj){
274                 obj = $(obj);
275                 var uid = obj.attr('data-filter');
276
277                 if(uid != score_choice){
278                     obj.parent().removeClass('active');
279                 }
280             });
281         }
282
283         if(!btn_active){
284             // when a category button is clicked, force deselection of adjacent categories
285             if(btn_id.startsWith("cat_")){
286                 btn_categories.forEach(function (category) {
287                     if (category != btn_id) {
288                         category = $('div#item>a[data-filter="' + category + '"').parent();
289
290                         if (category.hasClass("active")) { category.removeClass("active"); }
291                     }
292                 });
293             }
294
295             $(this).addClass('active');
296             var filter = $(this).attr('data-filter');
297             recent.load();
298         } else {
299             $(this).removeClass('active');
300             recent.load();
301         }
302     });
303
304     recent = new Recent();
305     recent.load();
306
307     // Detect end of page scroll
308     $(window).scroll(function() {
309         if($(window).scrollTop() + $(window).height() == $(document).height()) {
310             if(!recent.empty_results) recent.lazy_load();
311         }
312     });
313
314 });
315 </script>

```

Figure 4.3: Cuckoo recent analysis page JavaScript code.

By investigating the HTML source code for the `/analysis` page, we can see that the analysis ids are not directly present in the HTML. Instead the data (including the ids) is loaded via JavaScript (as seen in figure 4.3). By default, Scrapy will just fetch the HTML page at the url provided by us, and parse that. However, in this case we need it to actually render the JavaScript and wait for the ids to load.

```
1 import scrapy
2 from scrapy_playwright.page import PageMethod
3
4 class CuckooStartSpider(scrapy.Spider):
5     name = "cuckoo_start"
6     allowed_domains = ["cuckoo.cert.ee"]
7
8     def start_requests(self):
9         yield scrapy.Request(
10             url="https://cuckoo.cert.ee/analysis/",
11             meta={
12                 "playwright": True,
13                 "playwright_page_methods": [
14                     PageMethod("wait_for_selector", "table#recent tbody tr td strong")
15                 ]
16             })
17
18     def parse(self, response):
19         with open('./max_id_cuckoo.txt', 'w') as max_id_file:
20             max_id_file.write(response.css('table#recent tbody tr td strong::text').get())
21
```

Figure 4.4: cuckoo_start spider to fetch the max analysis id.

To render the JavaScript and wait for the analysis ids to load, the python plugin Playwright [64] is used. Playwright has an integration with Scrapy [73] which makes it possible to integrate directly into Scrapys workflow.

Figure 4.4 shows the source code for the CuckooStartSpider, that was made to fetch the max analysis id from the recent analysis page (`/analysis`), and write it to a text file. On line 14 in the figure, we see that the Playwright PageMethod "wait_for_selector" is used, which means that Playwright should render the website in a browser, until the css selector we specified is present on the page. To find the relevant element to wait for, the browser dev tools were used to inspect the page when fully loaded as shown in figure 4.5.

The screenshot shows the Cuckoo dashboard interface. At the top, there's a navigation bar with 'cuckoo' logo, 'Dashboard', 'Recent', 'Pending', and 'Search' buttons. Below this, there are tabs for 'Files', 'URLs', and score ranges 'Score 0 - 4', 'Score 4 - 7', and 'Score 7 - 10'. A table of recent analyses is displayed. The first row is highlighted, showing ID 4068421, date 2023-05-18 15:58, status '-', and URL https://harrisnicholas.org/zzr. Below the table, a developer tools window is open, showing the 'Elements' tab. It displays the HTML structure of the table, specifically the first row's cells: <table id='recent' class='table table-striped table-responsive' style='table-layout: fixed; '> <tbody> <tr> == \$0 <td> 4068421 </td> <td> ... </td> <td> ... </td>

Files	URLs	Score 0 - 4	Score 4 - 7	Score 7 - 10
4068421	2023-05-18 15:58	-		https://harrisnicholas.org/zzr
4068416	2023-05-18 15:44	-		https://flowbase-prod.eu.auth0.com/u/email-verification?ticket=rW7UeDKgB1HnReAgf7tBV7yhOd8mA25J#
4068408	2023-05-18 15:32	7617e155edd87b987a09b8b78f0fde59		SystemUtilities.exe
4068405	2023-05-18 15:32	-		http://discountsales.discountshop2023.ru
4068399	2023-05-18 15:29	-		https://red-eagle-team.net/ss/#helen.mcilleron@uct.ac.za
4068397	2023-05-18 15:28	c9fdb2c508c1984a9d9448276e7c8d71		Ejercicio_2.zip
4068396	2023-05-18 15:28	-		Re Simplifying and securing integration of embedded softwares.msg @ Ejercicio_2.zip

Figure 4.5: Cuckoo recent analysis page, using dev tools to find selector.

The element we want is the first analysis id in the table, as this represent the most recent analysis (which has the highest id). This can be grabbed with the following selector "table#recent tbody tr td strong". First, we specify that we're looking for a table with the id "recent", then we're just going through the nested structure as shown in figure 4.5, until we reach the "strong" element, which contains the actual id that we want to extract. Once the element appears on the page (ie. when the JavaScript has finished rendering), the page is then passed to the parse function.

The parse function (lines 18-20 in figure 4.4) extracts the most recent analysis id by using the selector identified in figure 4.5, and writes it to a file called "max_id_cuckoo.txt" for use as a starting point by the CuckooSpider, which will be elaborated upon in section 4.1.3.

4.1.3 Cuckoo Spider

The screenshot displays the 'Export analysis' interface in the Cuckoo web application. The top navigation bar includes 'Dashboard', 'Recent', 'Pending', and a search function. The left sidebar contains various icons for navigation. The main content area is titled 'Export analysis' and features a form with two columns of file selection options. Each option is preceded by a checked checkbox. The first column lists categories like 'files (8 files)', 'shots (3 files)', 'buffer (0 files)', 'extracted (0 files)', 'memory (0 files)', 'curtain (1 files)', 'sysmon (1 files)', 'package_files (0 files)', 'logs (3 files)', 'reports (2 files)', 'suricata (3 files)', and 'network (4 files)'. The second column lists specific files: 'task.json', 'binary', 'cuckoo.log', 'dump.pcap', 'analysis.log', 'files.json', 'reboot.json', 'tlsmaster.txt', and 'dump_sorted.pcap'. Below these options, the text 'Chosen analysis nr.4068577 to export' is displayed, followed by a box containing 'LMO_Setup_233291.exe'. At the bottom of the form is a blue button labeled 'Download 34.8 MB'.

Figure 4.6: Cuckoo export analysis page.

Cuckoo allows exporting analysis reports to JSON directly under the url `/analysis/<id>/export`. The export analysis page contains a form (as seen in figure 4.6), where the user can select which files they want to download. In this case we are only interested in the **reports**, so we deselect everything else and press the download button. Using the browser dev tools network tab, we can see that pressing the download button triggers a POST call to the same endpoint (`/analysis/<id>/export`), with the form data containing a variable called **csrfmiddlewaretoken**, as well as one called **dirs**. **csrfmiddlewaretoken** is used by the Django framework that Cuckoo's web interface is built on to protect against Cross Site Request Forgery (CSRF) [30].

To download the JSON-report from Cuckoo, two requests were needed per analysis task (see figures A.4 & A.5 in Appendix A). This is due to the way the site handles CSRF protection, by generating and setting a csrfmiddleware token both in a cookie and as a hidden element in the form that is submitted to get the actual file from the server [30]. Therefore, the first request to the server will be a GET request that is used for two things. First, to check if an analysis task with the given id exist (if not, the server will return 404 and no more requests will be made by the crawler for that id). If the analysis task does exist, we will save the session and csrfmiddleware cookies, and also set the csrfmiddleware token as part of the form data. This data is then used in addition to the specification of which files we want to download from the server, in the form of setting the “dirs” variable to “reports”. This call will return a .zip file containing the reports.json file that contains the full analysis report in JSON.

4.1.4 Unzip Cuckoo reports and convert to single .json file

The script **unzip_cuckoo_reports.py** (see figure A.1 in Appendix A) was made to unzip the **reports.json** files from the .zip files generated by the CuckooSpider. The output from running the script will be one .json file for each analysis task, named after the analysis id (ie. **3999517.json**). The script **convert_cuckoo_json_to_single_file.py** (see figures A.2 & A.3 in Appendix A) takes the analysis reports .json files (outputted by the **unzip_cuckoo_reports.py** script), and extracts the following information from each report, and saves it to a single json file (with one analysis task per line):

sha256 sha256 hash of the analyzed file

(ie. **12013662c71da69de977c04cd7021f13a70cf7bed4ca6c82acbc100464d4b0ef**)

sha1 sha1 hash of the analyzed file

(ie. **292559e94f1c04b7d0c65d4a01bbbc5dc1ff6f21**)

md5 md5 hash of the analyzed file

(ie. **eec5c6c219535fba3a0492ea8118b397**)

analysis_url The url from which the analysis report was scraped, including the analysis id

(ie. **https://cuckoo.cert.ee/analysis/3793297/**)

completed_at Date and time when the analysis was completed (ie. **2022-12-26 09:50:49**)

signatures List of behavioral signatures triggered by the analyzed sample (ie. **port_scan, network_cnc_http**).

convert_cuckoo_json_to_single_file.py will skip analysis reports based on urls from Cuckoo, so only the reports based on analyzing file samples are kept.

4.2 Scraping Cape

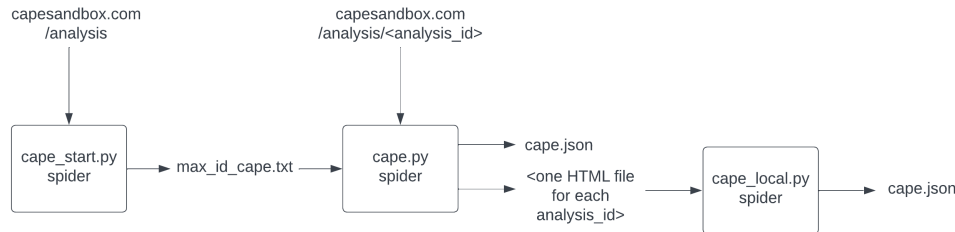


Figure 4.7: Cape data flow.

The following subsection will discuss how behavioral malware data was scraped from Cape Sandbox [23]. An overview of the data flow can be seen in figure 4.7.

Similarly to scraping Cuckoo, the user-agent needed to be set to something that looks like a browser to be able to get a response from the server.

The Cape max id was fetched (by the `cape_start` spider, see figure A.6 in Appendix A) from the recent analysis page (**capesandbox.com/analysis**), and written to a text file called **max_id_cape.txt** (similarly to Cuckoo). The max id file was then used by the `cape` spider (see figures A.7 & A.8 in Appendix A) as a starting point, and the analysis ids were decreased by 1 at a time to get previous analysis tasks. Cape does not offer exporting of analysis reports in .json without being logged in to the server as previously discussed in chapter 3. Therefore, the HTML pages were scraped instead to get data from Cape. The parse method extracted **md5**, **sha1**, **sha256**, **analysis_url**, **completed_at** and **signatures** to a .json with one line per analysis task. Besides the parsed data that was saved as .json, the source in the form of the HTML file for each analysis task was also saved and included in the data set. This enables extracting other data present in the HTML page without having to request the pages of the already gathered analysis tasks from the server again. A spider called **cape_local** (see figure A.9 & A.10 in Appendix A) was made to parse the local HTML files. It works similarly to the `cape` spider, but with the HTML files being locally stored and without the download delays discussed in section 4.3.

4.3 Ethical Considerations

Several precautions were taken to ensure that the websites scraped were impacted as little as possible by the scraping. A download delay was introduced using Scrapy's `AutoThrottle` middleware [15], to make sure the sites were not getting bombarded with requests, which in the worst case might have brought them down (similarly to a DDoS attack). The minimum download delay was set to 2 seconds, and the concurrent threads for each domain was to 1. This means that the crawler would never send more than 1 requests every 2 seconds to the same domain/sandbox. Furthermore, the `AutoThrottle` middleware would increase the download delay automatically based on how long

the server took to respond to each request (latency) [15]. At one point, the Cuckoo website took 11 seconds to respond to a request, and the following requests were therefore delayed by 11 seconds between each, since the AutoThrottle algorithm calculates the delay as:

$$\text{latency_from_previous_response} / \text{concurrent_threads_for_the_domain}$$

[15] The download delay on the following requests are then calculated as an average between the previous download delay and the target download delay, so the delay is gradually decreased when the server starts responding with a status code of 200 in a normal latency again [15].

Additionally, CAPE sandbox has some rate limiting in place (using Cloudflare). This means, that performing too many requests too fast to the server would result in a response with the http status code of 429 Too Many Requests. Neither the AutoThrottle middleware or the Retry middleware built in to Scrapy would honor these 429 status codes by slowing down. Therefore, a custom middleware called **TooManyRequestsRetryMiddleware** was implemented to handle this (see figure A.11 in Appendix A). Essentially, since this middleware replaces the existing Retry middleware, it first checks if the status code of the response was 429. If yes, it tries to fetch the value in the “retry-after” header. This is sent by the server to indicate when the client can start sending requests again without hitting the rate limit [70]. If no “retry-after” header is found, it will instead use a default delay of 60 seconds. This value was chosen since I never encountered a “retry-after” value over 60 seconds, suggesting that the rate limits in place may be implemented as a number of requests allowed per minute.

Regarding the Cuckoo crawler that requires two requests (a GET and a POST) for each analysis, the second (POST) request will only be made if the first (GET) request does not return 404 Not Found. This prevents unnecessarily making additional requests to the server when not needed thereby minimizing the load created.

Saving the CAPE analysis tasks HTML source locally was also done (as mentioned above) to be able to add additionally data to the .json output, without having to bother the web server unnecessarily by downloading the same page twice.

4.4 Analyzing Results

To help analyze the results, several python scripts were made.

4.4.1 remove_duplicates.py

A script was made to find samples with more than one analysis report from the same sandbox (see figures A.12, A.13 & A.14 in Appendix A). This was done by looking for each sample’s **sha256** hash, and count the amounts of reports based on that hash. If there were more than one report, the newest report was kept and the older ones discarded. The resulting .json file (without duplicate reports) were written to a new file and used for further analysis.

4.4.2 `check_same_hash.py`

This script takes two .json files as input, one with Cuckoo reports, and one with Cape reports (see figure A.22 in Appendix A). It then finds the samples that have an analysis report from both sandboxes, by looking for its `sha256` hash.

4.4.3 `calculate_results_matching_hashes.py` & `calculate_results_single_sandbox.py`

These scripts calculate some of the numbers (eg. average amount of signatures, different signatures) used in the results table in chapter 5 (see figures A.15, A.16, A.17 & A.18 in Appendix A).

4.4.4 `check_evasion_signature_from_repos.py`

This script was used to generate the data used in table 5.3 (see figures A.19, A.20 & A.21 in Appendix A). It filters the total amount of signatures down to those starting with given prefixes (ie. "antivm", "antisandbox"), and then counts how many times each of those signatures were present in analysis reports from Cuckoo and Cape respectively.

4.4.5 `find_signature_only_in_one_sandbox.py`

This script was made to find samples where a given signature was only present in either Cuckoo analysis reports or the Cape analysis report (see figure A.23 in Appendix A). This was used to compare reports and investigate some of the differences found in table 5.3.

4.5 Summary

In this chapter the implementation of scraping Cuckoo and Cape sandbox to gather analysis reports for a data set was presented. Ethical considerations related to web scraping were also discussed, and finally the scripts made for analyzing the results were presented.

Chapter 5

Results

In this chapter the results of the project are presented in relation to the three research questions formulated in section 2.6.

To help answer RQ1 (How can a behavioral malware data set be created, that can be used for future research?), a data set consisting of malware analysis reports from the online public sandboxes Cuckoo [35] and Cape [23] have been gathered by scraping the sites as discussed in chapter 4.

The full data set created in this project can be downloaded from: <https://nextcloud.ntp-event.dk:8443/s/LtttWPzk6TzeKtw>

To answer RQ2 (How can a behavioral malware data set be continuously updated in an easy way?) the source code for the spiders has been published, along with the state files, so that new reports can be added to the data set by crawling the sandboxes again.

The source code made as part of this project, can be downloaded at: <https://github.com/Villefrance/malware-web-scraper>

Sandbox	Total reports	File reports	Unique samples	Avg. signatures	Different signatures
Cuckoo [35]	537,951	516,193	273,603	2.81	271
Cape [23]	29,784	29,784	29,426	17.28	319

Table 5.1: Results for all analysis reports gathered by scraping Cuckoo [35] and Cape [23].

Table 5.1 shows the results for the full data set consisting of malware analysis reports from Cuckoo [35] and Cape sandbox [23]. As the column **Total reports** shows, the amount of analysis reports gathered from Cuckoo was over 500,000 more than from Cape. This was due to Cape not having more reports available on their site at the time of writing. Cuckoo allows analysis of urls instead of file samples [83]. The column **File reports** only include the analysis reports that were based on analyzing file samples, which filtered the amount of Cuckoo reports from 537,951 to 516,193. Cape does not offer url analysis, and therefore all the reports were based on analyzing files, and the number of reports were therefore unchanged. Some samples were analyzed multiple times in

the same sandbox, which resulted in multiple analysis reports for the same file hash. The **Unique samples** column only counts each sample once. The column **Avg. signatures** shows a significant difference between the average amount of signatures per analysis report in Cuckoo and Cape. The **Different signatures** column shows the amount of different signatures present throughout the reports for each sandbox. This is based on looking through all unique analysis reports and counting each signature found once (ie. the signature **antivirus_virustotal** may be present in a lot of reports, but is only counted once here to find out how many different signatures is represented in the reports for each sandbox). Cuckoo has 449 different signatures available in total (when considering the 'windows' and 'network' folder) in their Git repo [32], out of which 271 different signatures were found in the unique Cuckoo reports. Cape has 465 signatures available in their Git repo [22], out of which 319 were identified in the unique Cape reports.

To answer RQ3 (How can it be investigated whether Cuckoo Sandbox is evaded more often by malware compared to an actively maintained sandbox, such as Cape?), the full data set was reduced to only include reports for samples that were present in both the Cuckoo and Cape data. This was done to make a more fair and direct comparison. The reduced data set was made by only including the analysis reports concerning file samples (ie. Cuckoo url analysis reports were not included). Furthermore, only one report for each sample (from each sandbox) were included. In case of multiple reports for the same sample hash, only the newest report was kept in the data set. Finally, the sha256 sample hashes were compared between the two sandboxes' data to find the file samples that had an analysis report present in both the Cuckoo and the Cape data.

Sandbox	Matching sample reports	Matching sample avg. signatures
Cuckoo [35]	1,500	12.94
Cape [23]	1,500	17.22

Table 5.2: Subset of the full results, where both sandboxes have an analysis report for the same sample.

Table 5.2 shows the resulting numbers of the reduced data set. The column **Matching sample reports** shows that 1500 file samples had an analysis report from both sandboxes. Based on these 1500 reports, the average number of signatures per report were 12.94 for Cuckoo as shown in column **Matching sample avg. signatures**, which is a significant difference compared to the average of 2.81 seen in table 5.1. For Cape the average signatures for the matching samples reports were 17.22, which is close to the average of 17.28 in table 5.1.

To help answer RQ3, I chose to look for signatures (based on Cuckoo and Cape's signature repos [32, 22]) where the filename of the signature started with **antivm**, which indicate behavior where the malware tries to detect whether it is running in a virtualized environment, and **antisandbox**, which indicate that the malware tries to detect whether it is running inside a sandbox [78]. These signatures were chosen since they relate to malware's evasion behavior when being analyzed in sandboxes.

Signature	Cuckoo count	Cape count
antivm_vbox_provname	1	3
antivm_vbox_window	1	1
antivm_vbox_keys	11	18
antivm_generic_bios	14	28
antivm_vmware_keys	3	3
antivm_vbox_files	4	7
antivm_generic_cpu	10	307
antivm_vbox_devices	1	1
antisandbox_mouse_hook	1	2
antivm_vmware_files	3	3
antisandbox_sleep	105	625
antivm_generic_disk	5	14
antisandbox_joe_anubis_files	1	1
antisandbox_unhook	21	18

Table 5.3: Evasion signatures in matching samples reports.

Tabel 5.3 shows the evasion related signature that were present at least once in analysis reports from both Cuckoo and Cape. Generally, Cape identified the signatures more times, except some cases where it identified the same amount as Cuckoo. Only the signature **antisandbox_unhook** were identified more times in Cuckoo (21 times compared to Cape's 18). There's a few significant differences between the amount of times a given signature was seen in one sandbox compared to the other. For instance, the signature **antivm_generic_cpu** was seen in 10 Cuckoo reports compared to 307 Cape reports (which is over 30 times as much). The signature **antisandbox_sleep** also has quite a big difference with 105 Cuckoo reports against 625 Cape reports (around 6 times as much). These differences are investigated in section 5.1.

5.1 Comparing Reports

In this section I will investigate the differences in the amount of times the signatures **antivm_generic_cpu** and **antisandbox_sleep** were found in Cuckoo and Cape reports as seen in table 5.3.

While analyzing the results, I also stumbled upon another finding where the signature **ransomware_extensions** was found 13 times at Cuckoo compared to 2 times at Cape. This is also discussed in this section.

5.1.1 antivm_generic_cpu

If we look at one of the samples (**sha256: d0cc1ecb03997d41886914f7c78052b52d48c571067e016c8133-afa158885350**) where the signature **antivm_generic_cpu** is only present in the Cape analysis report, we can see that Cuckoo has identified 7 signatures for the sample, whereas Cape has identified 30 signatures for the same sample (see figure B.1 in Appendix B). 5 of the 7 signatures found by

Cuckoo relies on other services to identify the file as malicious (ie. VirusTotal, Irma, Suricata, Snort, Yara). The two remaining Cuckoo signatures are **raises_exception** and **application_raises_exception** which both indicate that the sample under analysis raised an unhandled exception and then crashed the process [66]. For Cape, we can see that different evasion behaviors were detected, but it still managed to identify the sample as using the CVE-2017-11882 Office exploit to exploit the equation editor in Microsoft Office applications to download and execute an .exe file. It also shows that private information from locally installed internet browsers and credentials from FTP clients were harvested.

5.1.2 **antisandbox_sleep**

By comparing the **antisandbox_sleep** signature from Cuckoo [12] with the one from Cape [11], we can see that Cuckoo triggers the signature when the total amount of seconds the sample tried to sleep is 120 seconds or more, whereas for Cape the threshold is set to 250 seconds. We can also see that both sandboxes base the signature on calls to the api `NtDelayExecution`. Looking at how the two sandboxes handle the sleep calls to `NtDelayExecution` in their respective monitors, we can see that Cape is able to handle infinite sleep [43], whereas Cuckoo is not [28, 79].

5.1.3 **ransomware_extensions**

I looked at one of the samples (sha256: **a294620543334a721a2ae8eaaf9680a0786f4b9a216d75b55cfd2-8f39e9430ea**) where the Cuckoo report has the signature **ransomware_extensions** and the Cape report for the same sample does not (see figure B.2 in Appendix B). Looking at the full Cuckoo analysis report for the sample (**3890280.json**) shows that the mark that triggered the signature was: **C:\Users\Administrator\AppData\Roaming\Mozilla\Firefox\Profiles\af61ph6j.default\parent.lock** (see figure B.3 in Appendix B). The file **parent.lock** appears in the **Profile** folder of **Mozilla Firefox**. The file is used by the Mozilla Firefox browser to make sure only one running copy of the application at a time can access a profile [65], and should therefore not be considered malicious or associated with ransomware. The Cuckoo signature was triggered due to the file having the file extension **.lock** [67], whereas the corresponding signature with the same name at Cape does *not* trigger on the **.lock** extension [68].

5.2 Tests

5.2.1 **Pafish**

I uploaded Pafish [61] to Cuckoo [63] as another way to test the sandboxes' anti-evasion behavior. Pafish tries to detect whether it is running in a virtual environment by using some of the same techniques that malware employs to evade analysis. When done analyzing, Pafish generates a log file with the result, that can then be used to see with which methods (if any) it succeeded in detecting the virtual machine. This is useful to get an indication as to which parameters malware

may use to detect the sandbox and evade analysis. It was not possible for me to submit files to Cape, since this required an active account (which I did not succeed in obtaining, as discussed in chapter 4). However, after uploading Pafish to Cuckoo, I used the sha256 hash to search for the sample on Cape, and found that it had been uploaded by another user [62]. Comparing the Pafish log files from Cuckoo (see figure B.4 in Appendix B) and Cape (see figure B.5 in Appendix B), showed that Pafish detected Cuckoo on 11 parameters, compared to 3 parameters at Cape.

5.2.2 Al-Khaser

Al-Khaser [4] is an application that test different malware evasion techniques (similar to Pafish, but using other techniques). I uploaded Al-Khaser to Cuckoo [6] and found a sample on Cape as well [5], but as Al-Khaser did not produce a log file (like Pafish did), it was not possible to see the resulting report as it was just shown in a CMD window (which I did not have access to, since I was not able to interact directly VM during analysis, and no useful screenshots were taken by the sandbox during analysis). However, it was still possible to compare the analysis reports generated by Cuckoo and Cape. Cape detected 15 signatures for the sample, whereas Cuckoo detected 3.

5.2.3 anticuckoo.exe

The application anticuckoo [9] is able to detect Cuckoo's usage of hooks into the certain Windows functions. I uploaded anticuckoo to Cuckoo, and based on the screenshot from the analysis report [10], we can see that the program succeeded in detecting several Cuckoo hooks. I tried to find an upload of anticuckoo (both by file name and hash) on Cape, but did not succeed in doing so.

The results presented in this chapter, will be discussed in chapter 6, where the main findings will also be highlighted.

Chapter 6

Discussion

In this chapter the results presented in chapter 5 will be discussed and interpreted. At the end of the chapter, the main findings will be summarized and highlighted.

6.1 RQ1: How can a behavioral malware data set be created, that can be used for future research?

This research question was answered by making a web scraper to scrape analysis reports from the two open-source sandboxes, Cuckoo and Cape, and turn this into a data set that is published, so it can be used by other researchers. The details of why scraping was chosen as a data gathering approach is discussed in chapter 3, and the actual implementation of the solution is discussed in chapter 4. Finally, the resulting data set is presented in chapter 5.

One could argue whether it is relevant to publish a data set containing a lot of analysis reports from Cuckoo, when I at the same time argue that researchers should consider whether using Cuckoo may be a problem in terms of malware evasion. I considered this, but came to the conclusion that the data would still be useful as long as the potential problem of malware evasion is kept in mind. For instance, it enables comparison with other sandboxes like Cape (or even commercial sandboxes) to test whether the hypotheses still holds when tested on a larger scale. Even for the samples where Cuckoo identify little (or no behavior) compared to Cape, it often classify the sample correctly as malicious anyway, due to using data from external services such as VirusTotal [85]. This mean that even if some (or all) behavior is missed by Cuckoo when analyzing a sample, it is at least still able to show whether the sample is (likely) malicious or not. Therefore, analysis reports from Cuckoo can still be useful, as long as this is kept in mind. Additionally, Cuckoo does have data for a lot of samples that Cape does not (ie. Cuckoo had around 4 million reports at the time of writing, where Cape only had about 30,000). A reason for this may be that Cuckoo is more well known within the research community, but also that Cape requires a user to get activated by the website admin to upload samples to the sandbox, which is not the case with Cuckoo (where everyone can upload

samples anonymously). This is not a problem if Cape is set up and run locally, as the user then has full control over the instance, but as discussed previously, setting up a sandbox like Cuckoo or Cape locally can be a challenging and time consuming task and may not be feasible for everyone. The data set is split up, so researchers can choose to use reports from only one of the sandboxes if desired (either Cuckoo or Cape).

6.2 RQ2: How can a behavioral malware data set be continuously updated in an easy way?

To answer this research question, the source code for the web scraper (discussed in chapter 4) has been published (see chapter 5) along with the associated state folders, which contains files that allows the scraper to know which analysis reports have already been crawled (as discussed in section 3.3). This enables other researchers to re-run the software at a later point in time, to grab new analysis reports from Cuckoo and Cape and add them to the data set in an easy way.

6.3 RQ3: How can it be investigated whether Cuckoo Sandbox is evaded more often by malware compared to an actively maintained sandbox, such as Cape?

It was hard to come up with a concluding answer for RQ3, since there is not a single measure or value that shows that one sandbox is evaded more often than the other. However, to make a fair comparison, I decided only to look at the reports from samples that had been analyzed by both Cuckoo and Cape. This resulted in a subset of the full data set, consisting of 1500 samples (with an analysis report from both Cuckoo and Cape) as shown in table 5.2. The amount of avg. signatures shows that the analysis reports from Cape had 4.28 more signatures than Cuckoo on average. This could indicate that Cape find more signatures than Cuckoo in general (possibly due to malware evading Cuckoo more often), but it may also be due to Cuckoo analyzing more benign samples (which result in a smaller amount of signatures).

The fact that Cape in general identified more evasion signatures as seen in table 5.3, further indicates that Cuckoo is evaded more often.

The results regarding `antivm_generic_cpu`, suggests that the analyzed sample managed to evade Cuckoo Sandbox but not Cape. Cape was able to identify the exploit used by the malware along with a lot of its behavior, whereas the process crashed on Cuckoo, suggesting that it may have detected the sandbox and therefore exited to hide its malicious behavior. Interestingly, the network activity in the Cuckoo report shows no activity, but the Suricata and Snort rules raised detected a (possibly malicious) .exe file download.

The difference in the number of **antisandbox_sleep** signatures triggered by Cuckoo and Cape was investigated by looking at the source code for the signature in each sandbox. This revealed that the total amount of seconds the malware tried to sleep should be over 120 seconds for Cuckoo to trigger the signature and over 250 seconds for Cape to do the same. This would suggest that Cuckoo's signature should be triggered more often, since the threshold is much lower. However, since this was not the case, I had to dig deeper to find a possible explanation for the difference. Both the signature from Cuckoo and Cape looks for the call **NtDelayExecution** to the Windows native API, so to find out how they handle these calls, I looked at their respective monitors, since they handle the hooking and monitoring part of the sandbox. The monitor for Cuckoo is called Cuckoo Monitor [34], and Cape's monitor is called Capemon [24]. Investigating how the monitors work, revealed that both sandboxes implement sleep skipping (to prevent malware from sleeping for long periods of time when under analysis), however, Capemon also implemented functionality for handling infinite sleep. The Cuckoo Monitor does not appear to have this functionality. A possible explanation of why Cape identified 6 times as many samples with sleep behavior compared to Cuckoo, even though Cuckoo's threshold is lower, could be due to the lack of handling of infinite sleep in Cuckoo, which may result in malware successfully managing to evade the sandbox more often.

Looking into the difference in the amount of the signature **ransomware_extensions** revealed that Cuckoo trigger the signature on the file extension, whereas Cape does not. In the Cuckoo analysis report for the analyzed sample, it looks like signature was triggered as a false positive, since the file that triggered it was a benign file used by Mozilla Firefox to make sure only one instance of the application is using a given profile at a time. This is an interesting finding, since it means that a sandbox may not only fail to identify a malware's behavior, it may also in some cases identify benign behavior as malicious due to having false positives on some of its signatures.

Comparing the log files from Pafish between Cuckoo and Cape, revealed that Pafish detected Cuckoo as a virtual environment on 9 parameters, whereas Cape was detected on 3 parameters. This suggest that Cape is more robust against anti-analysis (evasion) techniques. However, it also shows that it is still possible to detect Cape as well, and that there is still work to do by the developers to improve the anti-evasion techniques even further.

As I was not able to get the actual results from Al-Khaser, I compared the number of signatures found by Cuckoo and Cape while analyzing the program instead. This showed that Cape identified a lot more behavior than Cuckoo, suggesting that the program managed to evade (or at least hide itself for) Cuckoo to a bigger extent.

anticuckoo.exe revealed that it is still possible to detect Cuckoo by the use of its hooks into the Windows APIs. This is likely a problem, since it can be used by malware to detect that it is under analysis. As it was not possible to obtain a similar result for Cape, we can not directly use this as a comparison between the two. However, it does show another way for malware to detect Cuckoo,

which mean more ways for malware to evade analysis in the sandbox.

6.4 Main findings

Based on the discussion of the results in this chapter, the main findings were:

- Scraping was used to gather a behavioral malware data set consisting of analysis reports from Cuckoo and Cape sandbox. The resulting data set will be published for other researchers to use.
- The published data set can be updated by running the scrapers again, along with the state file to avoid collecting the same data from the servers twice. The source code for the scrapers, along with the utility scripts used for unzipping the Cuckoo reports will be published as well.
- While it is hard to conclude that Cuckoo sandbox is evaded more often than Cape, there are several indicators that this may be the case.
 - The average amount of signatures is higher in Cape, suggesting that more malicious behavior is found on average, possibly due to Cuckoo being evaded more often by malware.
 - Cape identified more evasion signatures, which further indicates that Cuckoo is evaded more often.
 - The comparison of reports between the sandboxes for the signatures **antivm_generic_cpu** and **antisandbox_sleep** both suggest that Cuckoo was evaded more often than Cape.
 - The tests with **Pafish** and **Al-Khaser** both showed that Cape was better at hiding itself and detecting more anti-evasion behavior than Cuckoo.
 - The **anticuckoo.exe** test showed that it is possible for malware to detect the presence of Cuckoo via the hooks it makes into the Windows APIs, further suggesting that malware may use this technique to evade the sandbox. It was not possible to find or upload this sample to Cape, so a comparison between the two was not possible here.
- A false positive was found in a Cuckoo report based on the signature **ransomware_extensions**. This could be interesting to investigate further to see if there are more cases of false positives, and whether Cape has false positives as well.

Chapter 7

Conclusion

This project aimed to answer the following problem statement (as described in section 2.6):

How can the lack of recent malware data sets containing behavioral features be addressed, and how can it be investigated whether the lack of updates to Cuckoo Sandbox is a problem?

The first part of the question concerning the lack of recent malware data sets containing behavioral features was addressed by creating spiders to crawl two online instances of open-source sandboxes, Cuckoo and Cape. The data set is easily updated by running the crawlers again at a later point in time to add analysis reports for new samples to the data set, without having to crawl the already gathered reports again (thanks to a state file that keeps track of already seen requests). I have made both the data set as well as the source code for the crawlers and utility scripts publicly available for other researchers to use.

The second part of the problem statement regarding whether it is a problem that Cuckoo Sandbox has not been updated since July 2019, was answered by comparing it to Cape sandbox in terms of malware evasion behavior. While it was not possible to conclude for certain that Cuckoo sandbox is evaded more often than Cape, several indications pointing in that direction were found, including comparisons of the amount of signatures identified by each sandbox for the same samples, as well as uploading different programs to the sandboxes that were able to report on the anti-evasion behavior of the sandboxes. More research is needed to test this hypotheses on a larger scale, but this project shows that Cuckoo was evaded in several cases, which in itself indicate a problem that should be taken into account if using the sandbox for future research.

A signature triggered by Cuckoo (which was not triggered by Cape for the same sample), was found to be a false positive. This should be investigated further in the future as to see whether this is a general problem for Cuckoo, and whether it is occurring in a sandbox like Cape as well.

In this project the comparison between Cuckoo and Cape sandbox were done with the focus on malware evasion behavior. In the future, it could also be interesting to compare the two sandboxes

in general. For instance, whether one sandbox is better than the other for specific use cases, or whether there is a pattern in what kind of malware is uploaded to which sandbox.

Labelling of the behavioral malware data set could also be useful to look into as future work, for example by using a tool like AVClass2 with VirusTotal (as mentioned in section 2.1.1). This would further strengthen the robustness of the data set, by providing standardized labels based on multiple antivirus vendors.

My main contributions of this project consist of a large published data set containing behavioral features that can be used for future research and easily updated with the published source code. Additionally, a comparison between Cuckoo and Cape sandbox showed that Cuckoo is likely evaded more often than Cape, although more research is needed to confirm this on a larger scale.

Bibliography

- [1] 2023 SECURITY REPORT: CYBERATTACKS REACH AN ALL-TIME HIGH IN RESPONSE TO GEO-POLITICAL CONFLICT, AND THE RISE OF 'DISRUPTION AND DESTRUCTION' MALWARE. Last accessed on 13-03-2023. URL: <https://research.checkpoint.com/2023/2023-security-report-cyberattacks-reach-an-all-time-high-in-response-to-geo-political-conflict-and-the-rise-of-disruption-and-destruction-malware/>.
- [2] Adel Abusitta, Miles Q. Li, and Benjamin C. M. Fung. "Malware classification and composition analysis: A survey of recent developments". In: *Journal of information security and applications* 59 (2021), p. 102828. DOI: 10.1016/j.jisa.2021.102828. URL: <https://dx.doi.org/10.1016/j.jisa.2021.102828>.
- [3] Amir Afianian et al. "Malware Dynamic Analysis Evasion Techniques: A Survey". In: *ACM computing surveys* 52.6 (2020), pp. 1–28. DOI: 10.1145/3365001. URL: <http://dl.acm.org/citation.cfm?id=3365001>.
- [4] *al-khaser* - Github. Last accessed on 28-05-2023. URL: <https://github.com/LordNoteworthy/al-khaser/>.
- [5] *al-khaser.exe* - Cape Analysis report. Last accessed on 28-05-2023. URL: <https://capesandbox.com/analysis/378400/>.
- [6] *al-khaser.exe* - Cuckoo Analysis report. Last accessed on 28-05-2023. URL: <https://cuckoo.cert.ee/analysis/4078360/summary>.
- [7] Bander Ali Saleh Al-rimy, Mohd Aizaini Maarof, and Syed Zainudeen Mohd Shaid. "Ransomware threat success factors, taxonomy, and countermeasures: A survey and research directions". In: *Computers & security* 74 (2018), pp. 144–166. DOI: 10.1016/j.cose.2018.01.001. URL: <https://dx.doi.org/10.1016/j.cose.2018.01.001>.
- [8] Hyrum S. Anderson and Phil Roth. *EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models*. Tech. rep. Cornell University Library, arXiv.org, 2018. URL: <https://arxiv.org/abs/1804.04637>.
- [9] *anticuckoo* - Github. Last accessed on 28/05-2023. URL: <https://github.com/therealdreg/anticuckoo>.
- [10] *anticuckoo.exe* - Cuckoo Analysis report. Last accessed on 28/05-2023. URL: <https://cuckoo.cert.ee/analysis/4079968/summary/>.

- [11] *antisandbox_sleep signature - Cape Community Github*. Last accessed on 28-05-2023. URL: https://github.com/CAPE Sandbox/community/blob/c1f56a947d0fc31b5299ae9e556e310d054e15e4/modules/signatures/antisandbox_sleep.py.
- [12] *antisandbox_sleep signature - Cuckoo Community Github*. Last accessed on 28-05-2023. URL: https://github.com/cuckoosandbox/community/blob/14864b9fa2ba2576ca11887caf37616eaec6d941/modules/signatures/windows/antisandbox_sleep.py.
- [13] *Anubis - Malware Analysis Tool (archive.org)*. Last accessed on 11-05-2023. URL: <https://web.archive.org/web/20150319043753/http://anubis.iseclab.org/?action=about>.
- [14] *any.run*. Last accessed on 11-05-2023. URL: <https://any.run/>.
- [15] *Autothrottle - Scrapy Documentation*. Last accessed on 30-05-2023. URL: <https://docs.scrapy.org/en/latest/topics/autothrottle.html>.
- [16] *AVClass - Github*. Last accessed on 15-05-2023. URL: <https://github.com/malicialab/avclass>.
- [17] Fabrizio Biondi et al. "Tutorial: an Overview of Malware Detection and Evasion Techniques". In: 2018. URL: https://link.springer.com/chapter/10.1007/978-3-030-03418-4_34.
- [18] Branislav Bosansky et al. "Avast-CTU Public CAPE Dataset". In: (2022). DOI: 10.48550/arxiv.2209.03188. URL: <https://arxiv.org/abs/2209.03188>.
- [19] Alvaro Botas et al. "On Fingerprinting of Public Malware Analysis Services". In: *Logic journal of the IGPL* 28.4 (2020), pp. 473–486. DOI: 10.1093/jigpal/jzz050.
- [20] *Cape - Github - Issue Template*. Last accessed on 21-05-2023. URL: https://github.com/kevoreilly/CAPEv2/issues/new?assignees=&labels=&projects=&template=bug_report.md&title=.
- [21] *CAPE Architecture*. Last accessed on 15-05-2023. URL: <https://capev2.readthedocs.io/en/latest/introduction/what.html#architecture>.
- [22] *CAPE Community Github - Signatures*. Last accessed on 27-05-2023. URL: <https://github.com/CAPE Sandbox/community/tree/master/modules/signatures>.
- [23] *Cape Sandbox*. Last accessed on 11-05-2023. URL: <https://capesandbox.com/>.
- [24] *capemon (Cape Monitor) - Github*. Last accessed on 01-06-2023. URL: <https://github.com/kevoreilly/capemon>.
- [25] *CAPEv2 - Github*. Last accessed on 11-05-2023. URL: <https://github.com/kevoreilly/CAPEv2/>.
- [26] Marcus Carpenter and Chunbo Luo. *Behavioural Reports of Multi-Stage Malware*. Tech. rep. Cornell University Library, arXiv.org, 2023. URL: <https://arxiv.org/abs/2301.12800>.
- [27] Ferhat Catak and Ahmet Yazı. *A Benchmark API Call Dataset for Windows PE Malware Classification*. Tech. rep. Cornell University Library, arXiv.org, 2021. URL: <https://arxiv.org/abs/1905.01999>.

- [28] Alexander Chailtyko and Stanislav Skuratovich. *DEFEATING SANDBOX EVASION: HOW TO INCREASE THE SUCCESSFUL EMULATION RATE IN YOUR VIRTUAL ENVIRONMENT*. URL: https://blog.checkpoint.com/wp-content/uploads/2016/10/DefeatingSandBoxEvasion-VB2016_CheckPoint.pdf.
- [29] *Claudia Compute Cloud Strato (Aalborg University) - Documentation*. Last accessed on 21-05-2023. URL: <https://www.strato-docs.claudia.aau.dk/>.
- [30] *Cross Site Request Forgery protection - Django documentation*. Last accessed on 30-05-2023. URL: <https://docs.djangoproject.com/en/2.1/ref/csrf/#how-it-works>.
- [31] *Cuckoo Blog - v.2.0.7 release*. Last accessed on 11-05-2023. URL: <https://cuckoosandbox.org/blog/207-interim-release>.
- [32] *Cuckoo Community Github - Signatures*. Last accessed on 27-05-2023. URL: <https://github.com/cuckoosandbox/community/tree/master/modules/signatures>.
- [33] *Cuckoo Docs - Agent - Github*. Last accessed on 11-05-2023. URL: <https://github.com/cuckoosandbox/cuckoo/blob/50452a39ff7c3e0c4c94d114bc6317101633b958/docs/book/installation/guest/agent.rst>.
- [34] *Cuckoo Monitor - Github*. Last accessed on 01-06-2023. URL: <https://github.com/cuckoosandbox/monitor>.
- [35] *cuckoo.cert.ee (Online instance of Cuckoo)*. Last accessed on 11-05-2023. URL: <https://cuckoo.cert.ee/>.
- [36] *Cybercrime To Cost The World \$10.5 Trillion Annually By 2025*. Last accessed on 04-05-2023. URL: <https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/>.
- [37] S. L. Shiva Darshan, M. A. Ajay Kumara, and C. D. Jaidhar. "Windows malware detection based on cuckoo sandbox generated report using machine learning algorithm". In: IEEE, Dec 2016, pp. 534–539. DOI: 10.1109/ICIINFs.2016.8262998. URL: <https://ieeexplore.ieee.org/document/8262998>.
- [38] Manuel Egele et al. "A survey on automated dynamic malware-analysis techniques and tools". In: ACM COMPUTING SURVEYS 44.2 (2012), pp. 1–42. DOI: 10.1145/2089125.2089126. URL: <http://dl.acm.org/citation.cfm?id=2089126>.
- [39] *Global Ransomware Damage Costs Predicted To Exceed \$265 Billion By 2031*. Last accessed on 13-03-2023. 2022. URL: <https://cybersecurityventures.com/global-ransomware-damage-costs-predicted-to-reach-250-billion-usd-by-2031/>.
- [40] *Google Scholar link - Aalborg University Library*. Last accessed on 30-05-2023. URL: <https://www.en.aub.aau.dk/software-webservices/google-scholar/>.
- [41] M. Graziano et al. *Needles in a Haystack: Mining Information from Public Dynamic Analysis Sandboxes for Malware Intelligence*. 2015. URL: <https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-graziano.pdf>.

- [42] Manabu Hirano, Ryo Hodota, and Ryotaro Kobayashi. "RanSAP: An open dataset of ransomware storage access patterns for training machine learning models". In: *Forensic Science International: Digital Investigation* 40 (2022), p. 301314. DOI: 10.1016/j.fsidi.2021.301314. URL: <https://dx.doi.org/10.1016/j.fsidi.2021.301314>.
- [43] *hook_sleep.c - Capemon Github*. Last accessed on 28-05-2023. URL: https://github.com/kevoreilly/capemon/blob/1404d7476c9049ece130c686080b4d61af9aa596/hook_sleep.c#L162.
- [44] *Hybrid Analysis*. Last accessed on 11-05-2023. URL: <https://hybrid-analysis.com/>.
- [45] Muhammad Ijaz, Muhammad Hanif Durad, and Maliha Ismail. "Static and Dynamic Malware Analysis Using Machine Learning". In: IEEE, Jan 2019, pp. 687–691. DOI: 10.1109/IBCAST.2019.8667136. URL: <https://ieeexplore.ieee.org/document/8667136>.
- [46] *Intezer Analyze*. Last accessed on 11-05-2023. URL: <https://analyze.intezer.com/>.
- [47] *JoeSandbox - cloud*. Last accessed on 11-05-2023. URL: <https://www.joesandbox.com/>.
- [48] Robert J. Joyce et al. "MOTIF: A Malware Reference Dataset with Ground Truth Family Labels". In: *Computers & security* 124 (2023), p. 102921. DOI: 10.1016/j.cose.2022.102921. URL: <https://dx.doi.org/10.1016/j.cose.2022.102921>.
- [49] Ansam Khraisat et al. "Survey of intrusion detection systems: techniques, datasets and challenges". In: *Cybersecurity* 2.1 (2019), pp. 1–22. DOI: 10.1186/s42400-019-0038-7. URL: <https://link.springer.com/article/10.1186/s42400-019-0038-7>.
- [50] Dimitrios Kouzis-Loukas. *Learning Scrapy*. Birmingham, England: Packt Publishing, 2016. ISBN: 978-1-78439-978-8. URL: <https://www.oreilly.com/library/view/learning-scrappy/9781784399788/>.
- [51] Tamas Lengyel et al. "Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system". In: ACSAC '14. ACM, Dec 08, 2014, pp. 386–395. DOI: 10.1145/2664243.2664252. URL: <http://dl.acm.org/citation.cfm?id=2664252>.
- [52] Abdullahi Mohammed Maigida et al. "Systematic literature review and metadata analysis of ransomware attacks and detection mechanisms". In: *Journal of reliable intelligent environments* 5.2 (2019), pp. 67–89. DOI: 10.1007/s40860-019-00080-3. URL: <https://link.springer.com/article/10.1007/s40860-019-00080-3>.
- [53] *Malware Attribute Enumeration and Characterization (MAEC) by Mitre*. Last accessed on 09-05-2023. URL: <https://maecproject.github.io/>.
- [54] *Malware Statistics - AV-Atlas (by AV-Test institute)*. Last accessed on 04-05-2023. URL: <https://portal.av-atlas.org/malware>.
- [55] Pascal Maniriho, Abdun Naser Mahmood, and Mohammad Javed Morshed Chowdhury. "A study on malicious software behaviour analysis and detection techniques: Taxonomy, current trends and challenges". In: *Future generation computer systems* 130 (2022), pp. 1–18. DOI: 10.1016/j.future.2021.11.030. URL: <https://dx.doi.org/10.1016/j.future.2021.11.030>.

- [56] A. Alfred Raja Melvin and G. Jaspher W. Kathrine. "A Quest for Best: A Detailed Comparison Between Drakvuf-VMI-Based and Cuckoo Sandbox-Based Technique for Dynamic Malware Analysis". In: vol. 1167. *Intelligence in Big Data Technologies-Beyond the Hype*. Singapore: Springer Singapore, 2020, pp. 275–290. ISBN: 9811552843. DOI: 10.1007/978-981-15-5285-4_27. URL: https://link.springer.com/chapter/10.1007/978-981-15-5285-4_27.
- [57] K. A. Monnappa. *Learning Malware Analysis*. Birmingham ; Mumbai: Packt Publishing, 2018. ISBN: 978-1-78839-250-1. URL: <https://www.oreilly.com/library/view/learning-malware-analysis/9781788392501/>.
- [58] *Most Common Desktop Useragents - Useragents.me*. Last accessed on 20-05-2023. URL: <https://web.archive.org/web/20230317075410/https://www.useragents.me/>.
- [59] Antonio Nappa et al. *PoW-How: An Enduring Timing Side-Channel to Evade Online Malware Sandboxes*. 2021. DOI: 10.1007/978-3-030-88418-5_5.
- [60] Ori Or-Meir et al. "Dynamic Malware Analysis in the Modern Era-A State of the Art Survey". In: *ACM computing surveys* 52.5 (2019), pp. 1–48. DOI: 10.1145/3329786. URL: <http://dl.acm.org/citation.cfm?id=3329786>.
- [61] *Pafish - Github*. Last accessed on 28-05-2023. URL: <https://github.com/a0rtega/pafish>.
- [62] *pafish.exe - Cape Analysis report*. Last accessed on 28/05-2023. URL: <https://capesandbox.com/analysis/391929/>.
- [63] *pafish.exe - Cuckoo Analysis report*. Last accessed on 28/05-2023. URL: <https://cuckoo.cert.ee/analysis/4079735/summary/>.
- [64] *Playwright for Python*. Last accessed on 30-05-2023. URL: <https://playwright.dev/python/>.
- [65] *Profile in use (parent.lock) - Mozillazine.org*. Last accessed on 28-05-2023. URL: https://kb.mozillazine.org/Profile_in_use.
- [66] *raises_exception.py (signature) - Cuckoo Community Github*. Last accessed on 01-06-2023. URL: https://github.com/cuckoosandbox/community/blob/master/modules/signatures/windows/raises_exception.py.
- [67] *ransomware_extensions (.lock) signature - Cuckoo Community Github*. Last accessed on 28-05-2023. URL: https://github.com/cuckoosandbox/community/blob/14864b9fa2ba2576ca11887caf37616eaec6d9/modules/signatures/windows/ransomware_fileextensions.py#L63.
- [68] *ransomware_extensions signature - Cape Community Github*. Last accessed on 28-05-2023. URL: https://github.com/CAPE Sandbox/community/blob/master/modules/signatures/ransomware_fileextensions.py.
- [69] Jason Reid and William Caelli. "DRM, trusted computing and operating system architecture". In: Jan 1, 2005, pp. 127–136. ISBN: 1445-1336. DOI: 10.1145/872035.872036. URL: <https://eprints.qut.edu.au/515/>.
- [70] *Retry-After header - Mozilla Developer Documentation*. Last accessed on 30/05-2023. URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Retry-After>.

- [71] *Scrapy - Documentation*. Last accessed on 11-05-2023. URL: <https://docs.scrapy.org/en/latest/>.
- [72] *Scrapy - Python Scraping Framework*. Last accessed on 11-05-2023. URL: <https://scrapy.org/>.
- [73] *scrapy-playwright: Playwright integration for Scrapy - Github*. Last accessed on 30-05-2023. URL: <https://github.com/scrapy-plugins/scrapy-playwright>.
- [74] Sebastián et al. "AVclass: A Tool for Massive Malware Labeling". In: vol. 9854. Research in Attacks, Intrusions, and Defenses. Switzerland: Springer International Publishing AG, 2016, pp. 230–253. ISBN: 9783319457185. DOI: 10.1007/978-3-319-45719-2_11. URL: https://link.springer.com/chapter/10.1007/978-3-319-45719-2_11.
- [75] Silvia Sebastián and Juan Caballero. "Avclass2: Massive Malware Tag Extraction from Av Labels". In: *Annual Computer Security Applications Conference* 12 (2020). DOI: 10.1145/3427228.3427261.
- [76] Giorgio Severi, Tim Leek, and Brendan Dolan-Gavitt. "Malrec: Compact Full-Trace Malware Recording for Retrospective Deep Analysis". In: vol. 10885. Detection of Intrusions and Malware, and Vulnerability Assessment. Switzerland: Springer International Publishing AG, 2018, pp. 3–23. ISBN: 3319934104. DOI: 10.1007/978-3-319-93411-2_1. URL: https://link.springer.com/chapter/10.1007/978-3-319-93411-2_1.
- [77] Magdy Sherif and Zohdy Mahmoud. *An In-Depth Look at Windows Kernel Threats*. Tech. rep. Trend Micro, 2022. URL: https://documents.trendmicro.com/assets/white_papers/wp-an-in-depth-look-at-windows-kernel-threats.pdf.
- [78] *Signatures Categories - Cape Docs*. Last accessed on 31-05-2023. URL: <https://capev2.readthedocs.io/en/latest/customization/signatures.html#categories>.
- [79] *sleep.c - CuckooMonitor Github*. Last accessed on 28-05-2023. URL: <https://github.com/cuckoosandbox/monitor/blob/2deb9ccd75d5a7a3fe05b2625b03a8639d6ee36b/src/sleep.c>.
- [80] Michael R. Smith et al. "Mind the Gap: On Bridging the Semantic Gap between Machine Learning and Information Security". In: (2020). DOI: 10.48550/arxiv.2005.01800. URL: <https://arxiv.org/abs/2005.01800>.
- [81] *tria.ge sandbox*. URL: <https://tria.ge/>.
- [82] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. "Survey of machine learning techniques for malware analysis". In: *Computers & security* 81 (2019), pp. 123–147. DOI: 10.1016/j.cose.2018.11.001. URL: <https://dx.doi.org/10.1016/j.cose.2018.11.001>.
- [83] *URL analysis - Cuckoo Sandbox docs*. Last accessed on 24-05-2023. URL: <https://cuckoo.readthedocs.io/en/latest/faq/?highlight=url#can-i-analyze-urls-with-cuckoo>.
- [84] Rakesh Verma, Victor Zeng, and Houtan Faridi. "Data Quality for Security Challenges: Case Studies of Phishing, Malware and Intrusion Detection Datasets". In: *CCS '19*. ACM, Nov 06, 2019, pp. 2605–2607. DOI: 10.1145/3319535.3363267. URL: <http://dl.acm.org/citation.cfm?id=3363267>.

- [85] *VirusTotal - Sample report*. Last accessed on 09-05-2023. URL: <https://www.virustotal.com/gui/file/1e3966e77ad1cbf3e3ef76803fbf92300b2b88af39650a1208520e0cdc05645b/>.
- [86] Aaron Weathersby. "Prevalence of PII within Public Malware Sandbox Samples and Implications for Privacy and Threat Intelligence Sharing". In: *Journal of Computing Sciences in Colleges* 37.3 (2021). DOI: 10.5555/3512489.3512516. URL: https://www.researchgate.net/profile/Aaron-Weathersby/publication/355493892_Prevalence_of_PII_within_Public_Malware_Sandbox_Samples_and_Implications_for_Privacy_and_Threat_Intelligence_Sharing/links/617766df0be8ec17a9304d7a/Prevalence-of-PII-within-Public-Malware-Sandbox-Samples-and-Implications-for-Privacy-and-Threat-Intelligence-Sharing.pdf.
- [87] Victor Zeng et al. "Diverse Datasets and a Customizable Benchmarking Framework for Phishing". In: IWSPA '20. ACM, Mar 16, 2020, pp. 35–41. DOI: 10.1145/3375708.3380313. URL: <http://dl.acm.org/citation.cfm?id=3380313>.
- [88] Bo Zhao. *Web Scraping*. May 2017. DOI: 10.1007/978-3-319-32001-4_483-1. URL: https://www.researchgate.net/profile/Bo-Zhao-3/publication/317177787_Web_Scraping/links/5c293f85a6fdccfc7073192f/Web-Scraping.pdf.

Appendix A

Appendix: Implementation

A.1 Scrapping

```
34 with progressbar.ProgressBar(max_value=number_of_files, redirect_stdout=True) as bar:
35     for item in os.listdir(sys.argv[1]): # input dir: ~/malware-web-scraper/malware/reports/cuckoo/zip/
36         if do_process:
37             if item.endswith(".zip"):
38                 filepath = os.path.abspath(item)
39                 with ZipFile(filepath) as zipfile:
40                     try:
41                         zipinfo = zipfile.getinfo("reports/report.json")
42                         filename = ntpath.basename(filepath) # Using ntpath instead of os to be compatible with Windows
43                                                         # style paths on Unix systems as well
44                         zipinfo.filename = filename.split(".")[0] + ".json" # "3899577.json"
45                         zipfile.extract(zipinfo, sys.argv[2]) # output dir: ~/malware-web-scraper/malware/reports/cuckoo/json/
46                     except KeyError:
47                         print("Error: {} does not contain reports/report.json.. Skipping this file..".format(filename))
48                         errors.append(filename)
49                     except:
50                         print("Error: Something went wrong while trying to extract reports/report.json from {}.. "
51                               "Skipping this file..".format(filename))
52                         errors.append(filename)
53             if i == number_of_files:
54                 if errors:
55                     print("Failed to extract the following files: {}".format(errors))
56                     print("Done.. Extracted {} json files to dir: {}".format(number_of_files, sys.argv[1]))
57                 bar.update(i)
58                 i += 1
59             else:
60                 break
```

Figure A.1: unzip_cuckoo_reports.py: Utility script to unzip json reports from Cuckoo zip files.


```

28 os.chdir(sys.argv[1])
29 print("Converting Cuckoo json reports to single file from dir: " + sys.argv[1])
30 i = 1
31 listdir = os.listdir(sys.argv[1])
32 number_of_files = len(listdir)
33 print("Converting {} files...".format(number_of_files))
34 result = {}
35 target_url_reports = []
36 errors = []
37 with open(sys.argv[2], "w") as output_file:
38     output_file.write("\n")
39     with progressbar.ProgressBar(max_value=number_of_files, redirect_stdout=True) as bar:
40         for item in os.listdir(sys.argv[1]): #input dir: ~/malware-web-scraper/malware/reports/cuckoo/zip/
41             if do_process:
42                 if item.endswith(".json"):
43                     filepath = os.path.abspath(item)
44                     filename = ntpath.basename(filepath)
45                     with open(filepath, "r") as file:
46                         try:
47                             data = json.load(file)
48                             if data["target"]["category"] == "url":
49                                 target_url_reports.append(filename)
50                             else:
51                                 md5 = data["target"]["file"]["md5"] if "md5" in data["target"]["file"] else ""
52                                 sha1 = data["target"]["file"]["sha1"] if "sha1" in data["target"]["file"] else ""
53                                 sha256 = data["target"]["file"]["sha256"] if "sha256" in data["target"]["file"] else ""
54                                 analysis_url = "https://cuckoo.cert.ee/analysis/{}/".format(data["info"]["id"])
55                                 completed_at = datetime.fromtimestamp(data["info"]["ended"]).strftime(
56                                     "%Y-%m-%d %H:%M:%S")
57                                 signatures = []
58                                 data_set = {"sha256": sha256, "sha1": sha1, "md5": md5, "analysis_url": analysis_url,
59                                           "completed_at": completed_at, "signatures": signatures}

```

Figure A.2: convert_cuckoo_json_to_single_file.py: Utility script to convert json reports from Cuckoo to a single file (part 1).

```

60         with open(sys.argv[2], "a") as output_file:
61             json.dump(data_set, output_file)
62             if i == number_of_files - 1:
63                 output_file.write("\n")
64             else:
65                 output_file.write(",\n")
66         except:
67             print("Error converting {}... Skipping this file...".format(filename))
68             errors.append(filename)
69         bar.update(i)
70         i += 1
71     else:
72         break
73 if target_url_reports:
74     print("Skipped the following reports due to target being url: {}".format(target_url_reports))
75     if sys.argv[4]:
76         with open(sys.argv[4], "wb") as target_url_file:
77             pickle.dump(target_url_reports, target_url_file)
78 if errors:
79     print("The following reports failed: {}".format(errors))
80     if sys.argv[5]:
81         with open(sys.argv[5], "wb") as errors_file:
82             pickle.dump(errors, errors_file)
83 with open(sys.argv[2], "a") as output_file:
84     output_file.write("\n")

```

Figure A.3: convert_cuckoo_json_to_single_file.py: Utility script to convert json reports from Cuckoo to a single file (part 2).

```

14 class CuckooSpider(scrapy.Spider):
15     name = "cuckoo"
16     allowed_domains = ["cuckoo.cert.ee"]
17     url = 'https://cuckoo.cert.ee/analysis/{}/export/'
18     start_id = ""
19
20     def start_requests(self):
21         path = './max_id.txt'
22         if not os.path.isfile(path):
23             sys.exit("Fatal error: 'max_id.txt' file not found. Please run cuckoo_start,"
24                     "or create this file manually first.")
25
26         with open('./max_id.txt', 'r') as max_id_file:
27             max_id = max_id_file.read()
28             if max_id.isdigit():
29                 self.start_id = max_id
30             else:
31                 sys.exit("Fatal error: max_id.txt did not contain a valid ID.")
32         if self.start_id.isdigit():
33             for i in range(0, int(self.amount_to_fetch)):
34                 yield scrapy.Request(self.url.format(int(self.start_id)-i), errback=self.get_errback)
35         else:
36             sys.exit("Fatal error: start_id is empty or not a number.")

```

Figure A.4: Cuckoo spider to fetch reports from cuckoo.cert.ee (part 1).

```

38     def parse(self, response):
39         self.state['items_count'] = self.state.get('items_count', 0) + 1
40         self.state['start_id'] = self.start_id
41         self.state['latest_url_crawled'] = response.request.url
42         self.state['last_updated'] = datetime.now(timezone.utc)
43         token = response.css("form input[name=csrfmiddlewaretoken]::attr(value)").extract_first()
44
45         formdata = {
46             "csrfmiddlewaretoken": token,
47             "dirs": "reports"
48         }
49
50         request = FormRequest(url=response.request.url,
51                               formdata=formdata,
52                               method='POST',
53                               callback=self.parse_file)
54
55         yield request
56
57     def parse_file(self, response):
58         id = response.request.url.split("/")[4]
59         with open('./reports/zip/{}.zip'.format(id), "wb") as zip_file:
60             zip_file.write(response.body)
61
62     def get_errback(self, failure):
63         self.logger.error(repr(failure))
64         if failure.check(HttpError):
65             response = failure.value.response
66             if response.status == 404:
67                 self.logger.error('404 not found on %s', response.url)
68                 self.state['not_found_errors'] = self.state.get('not_found_errors', 0) + 1
69

```

Figure A.5: Cuckoo spider to fetch reports from cuckoo.cert.ee (part 2).

```

4 class CapeStartSpider(scrapy.Spider):
5     name = "cape_start"
6     allowed_domains = ["capesandbox.com"]
7
8     def start_requests(self):
9         yield scrapy.Request("https://capesandbox.com/analysis/")
10
11     def parse(self, response):
12         with open('./max_id_cape.txt', 'w') as max_id_file:
13             all_ids = response.css('.table.table-striped.table-sm tbody tr td:nth-child(1) a').getall()
14             for id in all_ids:
15                 if "/analysis/" in id:
16                     max_id_file.write(id.split("/")[2]) # getting only the digits of the id
17                     break
18

```

Figure A.6: cape_start spider to fetch the max analysis id.

```

10 class CapeSpider(scrapy.Spider):
11     name = "cape"
12     allowed_domains = ["capesandbox.com"]
13     url = 'https://capesandbox.com/analysis/{}/'
14     start_id = "382783"
15
16     def start_requests(self):
17         path = './max_id_cape.txt'
18         if not os.path.isfile(path):
19             sys.exit("Fatal error: 'max_id_cape.txt' file not found. "
20                     "Please run cape_start, or create this file manually first.")
21
22         with open('./max_id_cape.txt', 'r') as max_id_file:
23             max_id = max_id_file.read()
24             if max_id.isdigit():
25                 self.start_id = max_id
26             else:
27                 sys.exit("Fatal error: max_id_cape.txt did not contain a valid ID.")
28
29         for i in range(0, int(self.amount_to_fetch)):
30             if self.start_id.isdigit():
31                 yield scrapy.Request(self.url.format(int(self.start_id)-i), errback=self.get_errback)
32             else:
33                 sys.exit("Fatal error: start_id is empty or not a number.")
34

```

Figure A.7: Cape spider to fetch reports from capesandbox.com (part 1).

```

38     def parse(self, response):
39         self.state['items_count'] = self.state.get('items_count', 0) + 1
40         self.state['start_id'] = self.start_id
41         self.state['latest_url_crawled'] = response.request.url
42         self.state['last_updated'] = datetime.now(timezone.utc)
43
44         md5 = "".join(response.css('div .panel.panel-default table tr').getall()).split(
45             '<th>MD5</th>\n' <td style="word-wrap: break-word;">')[1].split("</td>")[0]
46
47         sha1 = "".join(response.css('div .panel.panel-default table tr').getall()).split(
48             '<th>SHA1</th>\n' <td style="word-wrap: break-word;">')[1].split("</td>")[0]
49
50         sha256 = "".join(response.css('div .panel.panel-default table tr').getall()).split(
51             '<th>SHA256</th>\n' <td style="word-wrap: break-word;">')[1].split("\n")[0]
52
53         with open('./reports/cape/html/{0}.html'.format(response.request.url.split("/") [4]), "w") as html_file:
54             html_file.write(response.body.decode("utf-8"))
55
56         signatures = []
57         for signature in response.css("#signatures a").getall():
58             signature_name = ""
59             if "#signature_" in signature:
60                 signature_name = signature.split("#signature_")[1].split("'")[0]
61             else:
62                 print(signature)
63             if signature_name:
64                 signatures.append(signature_name)
65
66         item = CapeItem()
67         item['md5'] = md5
68         item['sha1'] = sha1
69         item['sha256'] = sha256
70         item['analysis_url'] = response.request.url
71         item['completed_at'] = response.css('#information div table tbody tr td:nth-child(4)::text').get()
72         item['signatures'] = signatures
73
74         yield item

```

Figure A.8: Cape spider to fetch reports from capesandbox.com (part 2).

```

9 class CapeLocalSpider(scrapy.Spider):
10     name = "cape_local"
11     url = 'https://capesandbox.com/analysis/{}/'
12
13     custom_settings = {
14         "DOWNLOAD_DELAY": 0,
15         "CONCURRENT_REQUESTS_PER_DOMAIN": 999,
16         "COOKIES_ENABLED": False,
17         "AUTOTHROTTLE_ENABLED": False
18     }
19
20     def start_requests(self):
21         os.chdir(self.input_dir)
22         for item in os.listdir(
23             self.input_dir): # input dir: ~/malware-web-scraper/malware/reports/cape/html/
24             if item.endswith(".html"):
25                 filepath = "file://{0}".format(os.path.abspath(item))
26                 yield scrapy.Request(filepath, errback=self.errback)

```

Figure A.9: Cape spider to parse locally stored HTML reports (part 1).

```

28 def parse(self, response):
29     sha256 = "".join(response.xpath('//div[@class="panel panel-default"]/table/tr').getall()).split(
30         '<th>SHA256</th>\n'
31         '<td style="word-wrap: break-word;">')[1].split("\n")[0]
32
33     filepath = str(response.request.url).split("/")[-1]
34
35     id = ntpath.basename(
36         filepath).split(".")[0]
37
38     analysis_url = self.url.format(id)
39
40     signatures = []
41     for signature in response.css('#statistics div div:nth-child(2) ul li::text').getall():
42         clean_string = signature.strip().replace('\n', '')
43         if not clean_string:
44             continue
45         signatures.append(clean_string)
46
47     item = CapeItem()
48     item['sha256'] = sha256
49     item['url'] = analysis_url
50     item['completed_at'] = response.css('#information div table tbody tr td:nth-child(4)::text').get()
51     item['signatures'] = signatures
52
53     yield item
54
55 def errback(self, failure):
56     self.logger.error(repr(failure))

```

Figure A.10: Cape spider to parse locally stored HTML reports (part 2).

```

17 class TooManyRequestsRetryMiddleware(RetryMiddleware):
18
19     DEFAULT_DELAY = 60 # Delay in seconds.
20     MAX_DELAY = 600 # Sometimes, retry-after has absurd values
21
22     def process_response(self, request, response, spider):
23         if request.meta.get('dont_retry', False):
24             return response
25
26         if response.status in self.retry_http_codes:
27             if response.status == 429:
28                 retry_after = response.headers.get('retry-after')
29                 try:
30                     retry_after = int(retry_after)
31                 except (ValueError, TypeError):
32                     delay = self.DEFAULT_DELAY
33             else:
34                 delay = min(self.MAX_DELAY, retry_after)
35             spider.logger.info(f'Retrying {request} in {delay} seconds.')
36             self.crawler.engine.pause()
37             time.sleep(delay)
38             self.crawler.engine.unpause()
39
40             reason = response_status_message(response.status)
41             return self._retry(request, reason, spider) or response
42
43         return response
44

```

Figure A.11: TooManyRequestsRetryMiddleware: Middleware to handle rate limiting gracefully on Cape.

A.2 Analyzing Results

```

8 potential_duplicates = {}
9 duplicates = {}
10
11 with open(sys.argv[1], "r") as file:
12     data = json.load(file)
13     j = 0
14     print("Checking for duplicate samples...")
15     unique = {each['sha256']: each for each in data}
16     print("Unique samples: {}".format(len(unique)))
17     with progressbar.ProgressBar(max_value=len(data), redirect_stdout=True) as bar:
18         for sample in data:
19             j += 1
20             bar.update(j)
21             if sample["sha256"] in potential_duplicates:
22                 potential_duplicates[sample["sha256"]].append(
23                     (sample["analysis_url"], sample["completed_at"]))
24             else:
25                 potential_duplicates[sample["sha256"]] = [
26                     (sample["analysis_url"], sample["completed_at"])]
27
28     print("Finding potential duplicates...")
29     with progressbar.ProgressBar(max_value=len(potential_duplicates),
30                                redirect_stdout=True) as bar:
31         i = 1
32         for potential_duplicate in potential_duplicates:
33             bar.update(i)
34             if len(potential_duplicates[potential_duplicate]) > 1:
35                 duplicates[potential_duplicate] = potential_duplicates[potential_duplicate]
36             i += 1
37
38     print("{} duplicate hashes found in data".format(len(duplicates)))

```

Figure A.12: remove_duplicates.py: Script to find samples with multiple analysis reports in the same sandbox (part 1).

```

39
40 newest_reports = {}
41 print("Finding newest reports of duplicates...")
42 with progressbar.ProgressBar(max_value=len(duplicates), redirect_stdout=True) as bar:
43     i = 1
44     for hash in duplicates:
45         newest_report = ()
46         #print(hash)
47         for tuple in duplicates[hash]:
48             current_datetime = datetime.strptime(tuple[1], '%Y-%m-%d %H:%M:%S')
49             if not newest_report or current_datetime > datetime.strptime(
50                 newest_report[1], '%Y-%m-%d %H:%M:%S'):
51                 newest_report = tuple
52         newest_reports[hash] = newest_report
53         bar.update(i)
54         i += 1
55
56 print("hashes with duplicate reports: {}".format(len(newest_reports)))
57 already_checked_duplicates = []
58 print("Writing output file...")

```

Figure A.13: remove_duplicates.py: Script to find samples with multiple analysis reports in the same sandbox (part 2).

```

59     with progressbar.ProgressBar(max_value=len(data), redirect_stdout=True) as bar:
60         i = 1
61         for sample in data:
62             bar.update(i)
63             data_set = {}
64             if sample["sha256"] in newest_reports:
65                 if sample["sha256"] not in already_checked_duplicates:
66                     if newest_reports[sample["sha256"]][0] == sample["analysis_url"]:
67                         already_checked_duplicates.append(sample["sha256"])
68                         #print(sample["sha256"])
69                         data_set = sample
70             else:
71                 data_set = sample
72
73         with open(sys.argv[2], "a") as output_file:
74             if i == 1:
75                 output_file.write("\n")
76             if data_set:
77                 json.dump(data_set, output_file)
78                 if i < len(data):
79                     output_file.write(",\n")
80             if i == len(data):
81                 output_file.write("\n")
82             i += 1

```

Figure A.14: remove_duplicates.py: Script to find samples with multiple analysis reports in the same sandbox (part 3).

```

6  with open(sys.argv[1], "r") as matching_hashes_file:
7      cuckoo_signatures_amount = {}
8      cape_signatures_amount = {}
9
10     cuckoo_signature_count = {}
11     cape_signature_count = {}
12     cape_signature_count_not_all = {}
13     cape_signature_count_all = {}
14
15     cuckoo_signatures_to_ignore = []
16     cape_signatures_to_ignore = []
17
18     matching_hashes = json.load(matching_hashes_file)
19     i = 0
20     with progressbar.ProgressBar(max_value=len(matching_hashes), redirect_stdout=True) as bar:
21         for matching_hash in matching_hashes:
22             i += 1
23             bar.update(i)
24             cuckoo_signatures_amount[matching_hash["sha256"]] = len(matching_hash["cuckoo_signatures"])
25             cape_signatures_amount[matching_hash["sha256"]] = len(matching_hash["cape_signatures"])
26
27             for cuckoo_signature in matching_hash["cuckoo_signatures"]:
28                 if cuckoo_signature not in cuckoo_signatures_to_ignore:
29                     if cuckoo_signature in cuckoo_signature_count:
30                         cuckoo_signature_count[cuckoo_signature] += 1
31                     else:
32                         cuckoo_signature_count[cuckoo_signature] = 1
33
34             for cape_signature in matching_hash["cape_signatures"]:
35                 if cape_signature in cape_signature_count:
36                     cape_signature_count[cape_signature] += 1
37                 else:
38                     cape_signature_count[cape_signature] = 1
39

```

Figure A.15: calculate_results_matching_hashes.py: Script to calculate results used in matching hashes results table (part 1).

```

40     for cape_signature in cape_signature_count:
41         if cape_signature_count[cape_signature] != len(matching_hashes):
42             cape_signature_count_not_all[cape_signature] = cape_signature_count[cape_signature]
43         else:
44             cape_signature_count_all[cape_signature] = cape_signature_count[cape_signature]
45
46     with open(sys.argv[2], 'w', newline='') as file:
47         writer = csv.writer(file)
48         field = ["signature", "cuckoo_count", "cape_count"]
49         writer.writerow(field)
50
51         already_seen_signatures = []
52         for signature in cuckoo_signature_count:
53             if signature in cape_signature_count:
54                 writer.writerow([signature, cuckoo_signature_count[signature], cape_signature_count[
55                     signature]])
56             else:
57                 writer.writerow([signature, cuckoo_signature_count[signature], 0])
58                 already_seen_signatures.append(signature)
59
60         for signature in cape_signature_count:
61             if signature not in already_seen_signatures:
62                 writer.writerow([signature, 0, cape_signature_count[signature]])
63
64     with open(sys.argv[3], 'w', newline='') as file:
65         writer = csv.writer(file)
66         field = ["signature", "cuckoo_count", "cape_count"]
67         writer.writerow(field)
68
69         cuckoo_signatures = []
70
71         for signature in cuckoo_signature_count:
72             cuckoo_signatures.append(signature)

```

Figure A.16: calculate_results_matching_hashes.py: Script to calculate results used in matching hashes results table (part 2).

```

73
74     for signature in cape_signature_count:
75         if signature in cuckoo_signatures:
76             writer.writerow([signature, cuckoo_signature_count[signature], cape_signature_count[
77                 signature]])
78
79     average_cuckoo_signatures = sum(cuckoo_signatures_amount.values()) / len(cuckoo_signatures_amount)
80     average_cape_signatures = sum(cape_signatures_amount.values()) / len(cape_signatures_amount)
81
82     sorted_cuckoo_signature_count = sorted(cuckoo_signature_count.items(), key=lambda x: x[1],
83         reverse=True)
84     sorted_cape_signature_count = sorted(cape_signature_count.items(), key=lambda x: x[1],
85         reverse=True)
86     print("Cuckoo top signatures: {}".format(sorted_cuckoo_signature_count))
87     print("Cape top signatures: {}".format(sorted_cape_signature_count))
88     print("Cuckoo amount of unique signatures: {}".format(len(sorted_cuckoo_signature_count)))
89     print("Cape amount of unique signatures: {}".format(len(sorted_cape_signature_count)))
90
91     print("Average Cuckoo signatures: {}".format(average_cuckoo_signatures))
92     print("Average Cape signatures: {}".format(average_cape_signatures))

```

Figure A.17: calculate_results_matching_hashes.py: Script to calculate results used in matching hashes results table (part 3).


```

5 with open(sys.argv[1], "r") as file:
6     signatures_amount = {}
7     signature_count = {}
8
9     reports = json.load(file)
10    i = 0
11    with progressbar.ProgressBar(max_value=len(reports), redirect_stdout=True) as bar:
12        for report in reports:
13            i += 1
14            bar.update(i)
15            signatures_amount[report["sha256"]] = len(report["signatures"])
16
17            for signature in report["signatures"]:
18                if signature in signature_count:
19                    signature_count[signature] += 1
20                else:
21                    signature_count[signature] = 1
22
23    average_signatures = sum(signatures_amount.values()) / len(signatures_amount)
24
25    sorted_signature_count = sorted(signature_count.items(), key=lambda x: x[1], reverse=True)
26
27    print("Count of signatures: {}".format(sorted_signature_count))
28    print("Amount of unique signatures: {}".format(len(sorted_signature_count)))
29    print("Average signatures: {}".format(average_signatures))

```

Figure A.18: calculate_results_single_sandbox.py: Script to calculate results used in full results table.

```

7 file_categories = ['antisandbox_', 'antivm_']
8 cuckoo_evasion_signatures = []
9 cape_evasion_signatures = []
10 evasion_signatures_in_both = []
11
12 os.chdir(sys.argv[1])
13 for cuckoo_signature_file in os.listdir(sys.argv[1]): # checking Cuckoo windows dir
14     if cuckoo_signature_file.startswith(tuple(file_categories)) and cuckoo_signature_file.endswith(".py"):
15         filepath = os.path.abspath(cuckoo_signature_file)
16         filename = ntpath.basename(filepath)
17         cuckoo_evasion_signatures.append(filename.split(".py")[0])
18
19 os.chdir(sys.argv[2])
20 for cuckoo_signature_file in os.listdir(sys.argv[2]): # checking Cuckoo networks dir
21     if cuckoo_signature_file.startswith(tuple(file_categories)) and cuckoo_signature_file.endswith(".py"):
22         filepath = os.path.abspath(cuckoo_signature_file)
23         filename = ntpath.basename(filepath)
24         cuckoo_evasion_signatures.append(filename.split(".py")[0])
25
26 os.chdir(sys.argv[3])
27 for cape_signature_file in os.listdir(sys.argv[3]): # checking Cape signatures dir
28     if cape_signature_file.startswith(tuple(file_categories)) and cape_signature_file.endswith(".py"):
29         filepath = os.path.abspath(cape_signature_file)
30         filename = ntpath.basename(filepath)
31         cape_evasion_signatures.append(filename.split(".py")[0])
32
33 print("Cuckoo evasion signatures: {}".format(cuckoo_evasion_signatures))
34 print("Cape evasion signatures: {}".format(cape_evasion_signatures))
35
36 for signature in cuckoo_evasion_signatures:
37     if signature in cape_evasion_signatures:
38         evasion_signatures_in_both.append(signature)

```

Figure A.19: check_evasion_signatures_from_repo.py: Script to find evasion signatures in reports from both sandboxes (part 1).

```

39
40 print("Evasion signatures in both: {}".format(evasion_signatures_in_both))
41
42 cuckoo_evasion_signatures_count = {}
43 cape_evasion_signatures_count = {}
44
45 with open(sys.argv[4], "r") as matching_signatures_file:
46     samples = json.load(matching_signatures_file)
47     for sample in samples:
48         for signature in cuckoo_evasion_signatures:
49             if signature in sample['cuckoo_signatures']:
50                 if signature in cuckoo_evasion_signatures_count:
51                     cuckoo_evasion_signatures_count[signature] += 1
52                 else:
53                     cuckoo_evasion_signatures_count[signature] = 1
54             for signature in cape_evasion_signatures:
55                 if signature in sample['cape_signatures']:
56                     if signature in cape_evasion_signatures_count:
57                         cape_evasion_signatures_count[signature] += 1
58                     else:
59                         cape_evasion_signatures_count[signature] = 1
60
61 print("Cuckoo evasion signatures count: {}".format(cuckoo_evasion_signatures_count))
62 print("Cape evasion signatures count: {}".format(cape_evasion_signatures_count))
63

```

Figure A.20: check_evasion_signatures_from_repo.py: Script to find evasion signatures in reports from both sandboxes (part 2).

```

64 for signature in evasion_signatures_in_both:
65     if signature in cuckoo_evasion_signatures_count and signature in cape_evasion_signatures_count:
66         print("Both - Signature: {} - Cuckoo count: {} - Cape count: {}".format(signature,
67             cuckoo_evasion_signatures_count[signature], cape_evasion_signatures_count[signature]))
68
69 for signature in cuckoo_evasion_signatures:
70     if signature not in evasion_signatures_in_both or \
71         (signature in evasion_signatures_in_both and signature not in cape_evasion_signatures_count):
72         if signature in cuckoo_evasion_signatures_count:
73             print("Cuckoo only - Signature: {} - Cuckoo count: {}".format(signature,
74                 cuckoo_evasion_signatures_count[signature]))
75         else:
76             print("Cuckoo only - Signature: {} - Cuckoo count: {}".format(signature, 0))
77
78 for signature in cape_evasion_signatures:
79     if signature not in evasion_signatures_in_both or \
80         (signature in evasion_signatures_in_both and signature not in cuckoo_evasion_signatures_count):
81         if signature in cape_evasion_signatures_count:
82             print("Cape only - Signature: {} - Cape count: {}".format(signature,
83                 cape_evasion_signatures_count[signature]))
84         else:
85             print("Cape only - Signature: {} - Cape count: {}".format(signature, 0))
86
87 print("Cuckoo signatures total: {} - signatures over 0: {}".format(len(cuckoo_evasion_signatures),
88     len(cuckoo_evasion_signatures_count)))
89 print("Cape signatures total: {} - signatures over 0: {}".format(len(cape_evasion_signatures),
90     len(cape_evasion_signatures_count)))
91
92 print("cuckoo_evasion_signatures_count: {}".format(cuckoo_evasion_signatures_count))
93 print("cape_evasion_signatures_count: {}".format(cape_evasion_signatures_count))

```

Figure A.21: check_evasion_signatures_from_repo.py: Script to find evasion signatures in reports from both sandboxes (part 3).

```

5 with open(sys.argv[3], "w") as output_file:
6     output_file.write("\n")
7
8 with open(sys.argv[2], "r") as cuckoo_file:
9     cuckoo_data = json.load(cuckoo_file)
10    cuckoo_sha_dict = {}
11    i = 0
12    with progressbar.ProgressBar(max_value=len(cuckoo_data),
13                                redirect_stdout=True) as bar:
14        print("Extracting Cuckoo samples...")
15        for cuckoo_sample in cuckoo_data:
16            i += 1
17            bar.update(i)
18            if "sha256" in cuckoo_sample:
19                if cuckoo_sample["sha256"] != "":
20                    cuckoo_sha_dict[cuckoo_sample["sha256"]] = cuckoo_sample["signatures"]
21        print("Done extracting Cuckoo samples...")
22    with open(sys.argv[1], "r") as cape_file:
23        cape_data = json.load(cape_file)
24        j = 0
25        print("Comparing Cape samples with Cuckoo samples...")
26        with progressbar.ProgressBar(max_value=len(cape_data),
27                                    redirect_stdout=True) as bar:
28            for cape_sample in cape_data:
29                j += 1
30                bar.update(j)
31                if cape_sample["sha256"] in cuckoo_sha_dict:
32                    data_set = {"sha256": cape_sample["sha256"],
33                               "cape_signatures": cape_sample["signatures"],
34                               "cuckoo_signatures": cuckoo_sha_dict[cape_sample["sha256"]]}
35                    with open(sys.argv[3], "a") as output_file:
36                        json.dump(data_set, output_file)
37                        output_file.write(",\n")
38    print("Done.")

```

Figure A.22: check_same_hash.py: Script to find samples where analysis reports are present in both sandboxes.

```
1 import json
2 import sys
3
4 signatures_to_find = ['antisandbox_sleep']
5
6 found_in_cuckoo = {}
7 found_in_cape = {}
8
9 with open(sys.argv[1], "r") as matching_signatures_file:
10     samples = json.load(matching_signatures_file)
11     for sample in samples:
12         for signature in signatures_to_find:
13             if signature in sample['cuckoo_signatures'] and signature not in sample['cape_signatures']:
14                 print("sha256: {} - signature {} found only in cuckoo_signatures".format(sample['sha256'],
15                 signature))
16                 if signature in found_in_cuckoo:
17                     found_in_cuckoo[signature].append(sample['sha256'])
18             else:
19                 found_in_cuckoo[signature] = [sample['sha256']]
20             elif signature in sample['cape_signatures'] and signature not in sample['cuckoo_signatures']:
21                 print("sha256: {} - signature {} found only in cape_signatures".format(sample['sha256'],
22                 signature))
23                 if signature in found_in_cape:
24                     found_in_cape[signature].append(sample['sha256'])
25             else:
26                 found_in_cape[signature] = [sample['sha256']]
27
28     print("Only found in Cuckoo: {}".format(found_in_cuckoo))
29     print("Only found in Cape: {}".format(found_in_cape))
30
```

Figure A.23: find_signature_only_in_one_sandbox.py: Script to find samples where a signature is only present in one sandbox's analysis report.

Appendix B

Appendix: Results

B.1 Comparing Reports

```
1 {
2   "sha256": "d0cc1ecb03997d41886914f7c78052b52d48c571067e016c8133afa158885350",
3   "cape_signatures":
4     [
5       "dead_connect",
6       "antidebug_setunhandledexceptionfilter",
7       "antivm_network_adapters",
8       "dll_load_uncommon_file_types",
9       "antidebug_guardpages",
10      "antisandbox_sleep",
11      "dynamic_function_loading",
12      "encrypted_ioc",
13      "powershell_download",
14      "powershell_request",
15      "stealth_window",
16      "network_cnc_http",
17      "network_http",
18      "recon_checkip",
19      "terminates_remote_process",
20      "injection_inter_process",
21      "infostealer_browser",
22      "office_cve2017_11882",
23      "office_cve2017_11882_network",
24      "ransomware_file_modifications",
25      "spawns_dev_util",
26      "injection_process_hollowing",
27      "antivm_generic_cpu",
28      "modify_proxy",
29      "infostealer_ftp",
30      "injection_runpe",
31      "network_questionable_http_path",
32      "recon_fingerprint",
33      "procmem_yara",
34      "network_ip_exe"
35    ],
36   "cuckoo_signatures":
37     [
38       "file_yara",
39       "raises_exception",
40       "application_raises_exception",
41       "snort_alert",
42       "suricata_alert",
43       "antivirus_irma",
44       "antivirus_virustotal"
45     ]
46 }
```

Figure B.1: Signatures found for sample `d0cc1ecb03997d41886914f7c78052b52d48c571067e016c8133afa158885350` in Cuckoo and Cape.

```

1  {
2    "sha256": "a294620543334a721a2ae8eaf9680a0786f4b9a216d75b55cfd28f39e9430ea",
3    "cape_signatures":
4      [
5        "ransomware_file_modifications",
6        "physical_drive_access",
7        "infostealer_cookies"
8      ],
9    "cuckoo_signatures":
10     [
11       "antivm_memory_available",
12       "creates_doc",
13       "creates_exe",
14       "creates_shortcut",
15       "privilege_luid_check",
16       "ransomware_extensions",
17       "antivirus_irma",
18       "antivirus_virustotal"
19     ]
20   }

```

Figure B.2: Signatures found for sample `a294620543334a721a2ae8eaf9680a0786f4b9a216d75b55cfd28f39e9430ea` in Cuckoo and Cape.

```

1307 {
1308   "families": [],
1309   "description": "Appends a known multi-family ransomware file extension to files that have been encrypted",
1310   "severity": 3,
1311   "ttp": {},
1312   "markcount": 1,
1313   "references": [],
1314   "marks": [
1315     {
1316       "category": "file",
1317       "ioc": "C:\\Users\\Administrator\\AppData\\Roaming\\Mozilla\\Firefox\\Profiles\\af61ph6j.default\\parent.lock",
1318       "type": "ioc",
1319       "description": null
1320     }
1321   ],
1322   "name": "ransomware_extensions"
1323 },

```

Figure B.3: Snippet of the full analysis report for sample `a294620543334a721a2ae8eaf9680a0786f4b9a216d75b55cfd28f39e9430ea` in Cuckoo, showing what file triggered the signature `ransomware_extensions`.

B.2 Tests

```
1 [pafish] Start
2 [pafish] Windows version: 6.1 build 7601 (WoW64)
3 [pafish] CPU: GenuineIntel (HV: VBoxVBoxVBox) Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz
4 [pafish] CPU VM traced by checking the difference between CPU timestamp counters (
rdtsc) forcing VM exit
5 [pafish] CPU VM traced by checking hypervisor bit in cpuid feature bits
6 [pafish] CPU VM traced by checking cpuid hypervisor vendor for known VM vendors
7 [pafish] Sandbox traced by missing mouse movement
8 [pafish] Sandbox traced by missing mouse movement or supernatural speed
9 [pafish] Sandbox traced by missing mouse click activity
10 [pafish] Sandbox traced by missing double click activity
11 [pafish] Sandbox traced by missing dialog confirmation
12 [pafish] Sandbox traced by missing or implausible dialog confirmation
13 [pafish] Hooks traced using ShellExecuteExW method 1
14 [pafish] VirtualBox device identifiers traced using WMI
15 [pafish] End
```

Figure B.4: Pafish log from Cuckoo.

```
1 [pafish] Start
2 [pafish] Windows version: 6.1 build 7601 (native)
3 [pafish] CPU: GenuineIntel Intel(R) Core(TM) i3-4130 CPU @ 2.00GHz
4 [pafish] CPU VM traced by checking the difference between CPU timestamp counters (
rdtsc) forcing VM exit
5 [pafish] Sandbox traced by missing mouse movement or supernatural speed
6 [pafish] Sandbox traced by missing or implausible dialog confirmation
7 [pafish] End
```

Figure B.5: Pafish log from Cape.