KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

Overcoming the Challenge of Evasive Malware: Building a Comprehensive Dataset for Detection and Analysis

Maxim Reynvoet

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Veilige software

Promotoren:
Prof. dr. ir. L. Desmet
Prof. dr. ir. W. Joosen

Evaluator:
Dr. ir. F. Vogels

Begeleider:
Ir. M. Shafiei

Academiejaar 2023 – 2024

Preface

With a limited background in the topics of malware and malware analysis, embarking on this research journey has been both challenging and exhilarating. Immersing myself in this field, I have grown both as a scholar and an individual, learning to navigate and solve complex problems, managing setbacks, and effectively communicate my findings. This experience has not only deepened my appreciation of scientific work but also sparked a lasting interest in malware research.

I would like to express my deepest gratitude to my assistant-supervisor, ir. M. Shafiei, for his invaluable guidance, encouragement, and patience throughout this process. His expertise and insights have been instrumental in shaping this work. I am also sincerely thankful to prof. dr. ir. L. Desmet for his feedback and support during critical moments.

Finally, I extend my heartfelt thanks to my family and friends for their unwavering support. Their belief in me has been a constant source of motivation.

I hope that this thesis contributes to ongoing research in malware analysis and provides valuable insights for future work.

Maxim Reynvoet

Contents

Abstract

This thesis presents the development of a specialized dataset focusing on evasive malware - malicious software programs designed to circumvent traditional analysis methods through various strategies known as as evasion techniques. As cyber threats become more sophisticated, the need for a dedicated dataset to study these evasion techniques is essential for advancing the analysis of malicious software and enhancing cybersecurity. Existing public datasets often mix evasive and non-evasive malware samples, complicating efforts to isolate and study evasion techniques. This research addresses that gap by providing a curated, graph-based database exclusively containing evasive malware, categorized according to a detailed taxonomy of evasion methods. The dataset was compiled using an enhanced version of the CAPEv2 malware analysis sandbox, which was improved through a methodology developed in this work to identify currently undetected evasion techniques. When applied to CAPEv2, this approach uncovered techniques that were previously undetectable, which are now within the enhanced tool's detection capabilities. Key contributions of this thesis include the creation of an up-to-date taxonomy of evasion techniques, the development of a methodology for detecting currently undetected evasion techniques in existing analysis tools, the resulting improvements to CAPEv2's detection capabilities, and a dataset containing exclusively evasive malware samples. This dataset represents a significant contribution to the field, offering researchers a valuable resource to study evasion techniques and develop new evasion detection strategies. The graph-based structure of the database, along with the modeling of the taxonomy, enables intuitive and flexible querying across various abstraction levels, detailed analysis, and easy expansion as new techniques are identified. By offering a focused resource for studying evasive malware and a robust methodology for enhancing detection tools, this work lays the groundwork for more effective analysis methods, ultimately contributing to stronger defenses against increasingly sophisticated cyber threats.

Samenvatting

Deze thesis presenteert de ontwikkeling van een gespecialiseerde dataset die zich richt op evasieve malware - kwaadaardige softwareprogramma's die zijn ontworpen om traditionele analysemethoden te omzeilen door middel van verschillende strategieën, beter bekend als evasieve technieken. Naarmate cyberdreigingen complexer worden, is de behoefte aan een toegewijde dataset om deze evasieve technieken te bestuderen essentieel voor vooruitgang in de analyse van kwaadaardige software en het verbeteren van de cyberbeveiliging. Bestaande openbare datasets mengen vaak evasieve en niet-evasieve malware, wat de inspanningen om evasieve technieken te isoleren en te bestuderen bemoeilijkt. Dit onderzoek vult die leemte door een zorgvuldig samengestelde, grafe-gebaseerde databank te bieden die uitsluitend evasieve malware bevat, gecategoriseerd volgens een gedetailleerde taxonomie van evasieve technieken. De dataset is samengesteld met behulp van een verbeterde versie van de CAPEv2 malware sandbox, die werd verbeterd door een methodologie die in dit werk is ontwikkeld om momenteel onopgemerkte evasieve technieken te identificeren. Toegepast op CAPEv2, onthulde deze methode technieken die eerder niet detecteerbaar waren, maar nu binnen de detectiecapaciteiten van de verbeterde versie vallen. Belangrijke bijdragen van deze thesis omvatten de creatie van een actuele taxonomie van evasion techniques, de ontwikkeling van een methodologie voor het detecteren van momenteel niet detecteerbare evasieve technieken in bestaande analysemethoden, de resulterende verbeteringen in de detectiecapaciteiten van CAPEv2, en een databank die uitsluitend evasieve malware bevat. Deze databank vertegenwoordigt een belangrijke bijdrage aan het onderzoeksdomein, door onderzoekers een waardevolle bron te bieden om evasieve technieken te bestuderen en nieuwe strategieën voor de detectie ervan te ontwikkelen. De grafe-gebaseerde structuur van de databank, samen met de modellering van de taxonomie, maakt intuïtieve en flexibele query's op verschillende abstractieniveaus mogelijk, gedetailleerde analyse en eenvoudige uitbreiding naarmate nieuwe technieken worden geïdentificeerd. Door een gerichte bron te bieden voor het bestuderen van evasieve malware en een robuuste methodologie voor het verbeteren van detectiemethodes, legt dit werk de basis voor effectievere analysemethoden, wat uiteindelijk bijdraagt aan sterkere verdedigingen tegen steeds geavanceerdere cyberdreigingen.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

2FA	Two-Factor Authentication
AKA	Also Known As
AV	Antivirus
C2	Command & Control
CPU	Central Processing Unit
DDoS	Distributed Denial of Service
EDR	Endpoint Detection and Response
EM	Electromagnetic
HTML	HyperText Markup Language
IoT	Internet of Things
MaaS	Malware-as-a-Service
PC	Personal Computer
RAT	Remote Access Trojan or Remote Administration Tool
SEO	Search Engine Optimization
VM	Virtual Machine
WMI	Windows Management Instrumentation

Chapter 1

Introduction

In today's world, where computing devices are increasingly ubiquitous and interconnected, cyber threats have grown more prevalent and dangerous, posing significant risks to our daily lives and society as a whole [57]. Among these threats, malicious software, AKA malware, stands out as one of the most common and severe [6], making it critically important to understand and combat these harmful programs. However, the rise of evasive malware variants has added a new layer of complexity to this task. These sophisticated programs employ advanced tactics such as anti-debugging, anti-virtualization, and anti-instrumentation techniques to evade detection and analysis by conventional security systems.

The use of such evasion techniques presents significant challenges for security analysts, making it difficult to accurately assess the behavior of these malware samples and identify them as malicious. This challenge is further compounded by the lack of publicly available datasets that focus specifically on evasive malware and clearly indicate the evasion techniques used in each instance. Publicly available datasets often mix evasive and non-evasive samples, complicating the study of evasive malware by requiring researchers to manually isolate relevant samples. Although some studies have extracted subsets of evasive malware from broader datasets for their evasion detection approaches, a notable gap remains: a dedicated dataset that exclusively contains evasive malware samples, each categorized according to the specific evasion techniques employed.

This thesis aims to develop an evasion detection approach to identify currently undetected techniques in existing malware analysis tools, ultimately creating a database of malware samples that documents the evasion techniques used by each sample.

1.1 Outline

This thesis is organized into six chapters, each contributing to a comprehensive examination of evasive malware detection and analysis. The chapters are structured to guide the reader through the development of an up-to-date taxonomy of evasion techniques, the implementation of an approach for identifying undetected techniques,

and the creation of a specialized dataset. The following sections will provide a brief overview of each chapter's focus, highlighting how they build upon one another to advance the understanding and effectiveness of malware detection.

Chapter 2 - Background & Related Work

This chapter provides a comprehensive overview of evasive malware and malware analysis, with a detailed introduction to the CAPEv2 tool, including its functionalities and components. This foundational understanding of evasive malware and CAPEv2 is essential for appreciating the subsequent enhancements discussed in later chapters. Additionally, the chapter reviews significant related works, discussing various taxonomies, frameworks, and tools for categorizing and detecting evasion techniques. By examining these studies, the chapter situates the current research within the broader context of malware detection and analysis, highlighting both existing gaps and advancements in the field.

Chapter 3 - Methodology

This chapter describes the approach developed to identify currently undetected evasion techniques within existing malware analysis tools, specifically CAPEv2. It details the application of this approach to enhance CAPEv2's capabilities, leading to the identification of previously undetected evasion methods. The chapter highlights the creation of a comprehensive, graph-based dataset of evasive malware, which is a key contribution of this work. This dataset, exclusively composed of evasive samples, addresses the lack of publicly available resources in this field. The database is modeled using an up-to-date taxonomy, developed through an extensive literature review, which is provided in this chapter. The graph-based structure of the dataset is emphasized for its flexibility and extensibility, underscoring its significance in advancing malware analysis and detection.

Chapter 4 - Evaluation

This chapter evaluates both the effectiveness of the new detections incorporated into the improved version of CAPE and the utility of the newly created graph-based database. The effectiveness of the enhanced detections is assessed through tests on a control set and a broader range of malware samples. The results are compared with findings from related academic work to gauge the effectiveness of using CAPE as an evasion detection tool. Additionally, the chapter explores the utility and limitations of the graph database, focusing on how effectively it supports querying at different abstraction levels of the taxonomy. It also discusses how the approach for identifying undetected techniques and the graph-based database contribute to advancing the field of evasive malware detection and analysis, highlighting both the strengths and potential areas for further development.

Chapter 6 - Conclusion

This chapter summarizes the main contributions of the thesis, emphasizing the development of a dedicated dataset of evasive malware, which fills a critical gap in the field. The dataset's graph-based structure offers significant advantages, including flexibility, extensibility, and the ability to model the taxonomy of evasion techniques. The chapter also reflects on the enhancements made to CAPE, which now includes improved detection of previously undetected evasion techniques. Overall, the work advances methodologies in malware detection and analysis, offering valuable resources for future research.

Chapter 2

Background & Related Work

This chapter provides a comprehensive overview of key concepts and techniques essential for understanding malware analysis. It begins by introducing malicious software, defining its various forms, and explaining the challenges associated with detecting and analyzing such threats. The discussion then transitions to evasion techniques employed by malware to circumvent detection and analysis, highlighting the need for effective countermeasures. Static and dynamic analysis methods are examined in detail, along with their respective strengths and limitations. Hybrid analysis is introduced as a method that integrates both static and dynamic techniques to mitigate their individual shortcomings. This foundational understanding is crucial for applying these concepts to CAPEv2, an open-source sandbox used in our approach to identify evasive techniques that are currently undetected. By leveraging CAPEv2 and developing new detection methods, we aim to create a dataset of malware exhibiting evasion techniques, addressing the scarcity of datasets that exclusively contain evasive malware. This dataset will provide a valuable resource for advancing research in the field.

2.1 Malicious Software

In the early 1970s, a program called "Creeper" spread itself across *ARPANET*, the precursor to the Internet, printing out a whimsical message on the teleprinters of the machines it resided on: "I'M THE CREEPER: CATCH ME IF YOU CAN". It was the first *computer worm* [60], a program that replicates itself on a host machine, scans its operating environment for potential victims and propagates to infect them, continuing this process [59]. Although Creeper was a benign experiment designed to explore the possibilities of self-replicating programs, it laid the groundwork for the development of software with malicious objectives capable of spreading autonomously.

With an understanding of the origins of malware, the terminology used to describe this harmful software can now be delved into.

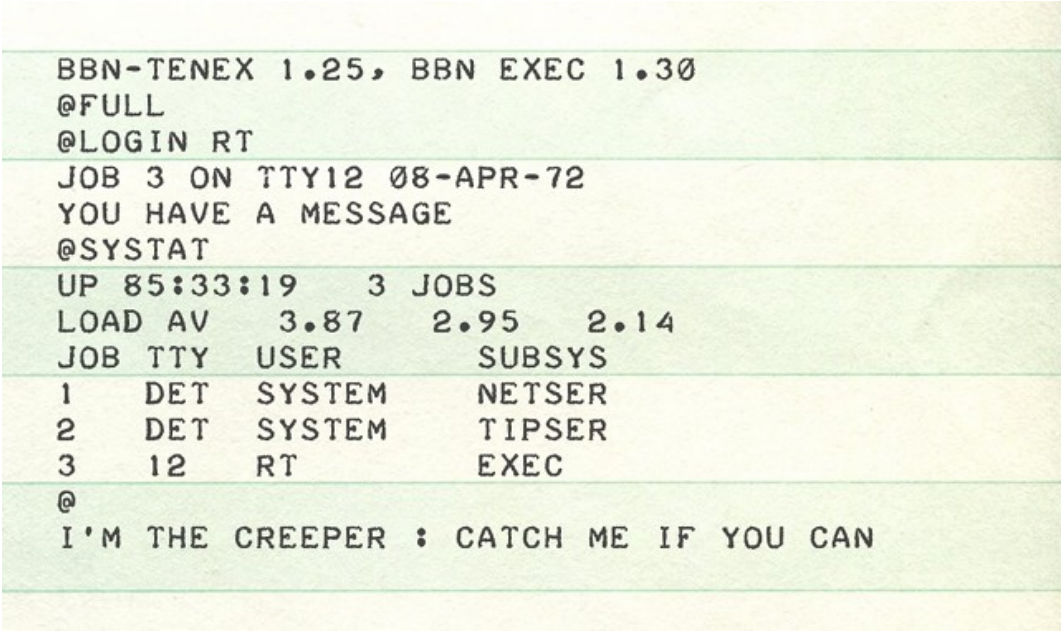


Figure 2.1: Message printed out by Creeper [5]

2.1.1 Etymology

Malware Definition The word malware is a portmanteau of "**mal**icious soft**ware**". It is an umbrella term for any software that is purposefully designed to harm to a computer system or its users [62]. Cybercriminals create and use malware for various reasons, including the following [21]:

- **Stealing** login credentials, credit card information, intellectual property or other valuable data.
- **Extracting a ransom** by holding data, devices or networks hostage with the promise of release upon payment of a large sum of money.
- **Disrupting** critical systems or infrastructure, from important company assets to public infrastructure such as electricity networks.
- **Gaining unauthorized access** to sensitive data or digital assets.
- **Remotely controlling** an infected machine, often by connecting it to a Command and Control (C2) server that allows issuing commands on the infected machine.

Payload Definition This malicious purpose is executed by what is called a *payload*, which is the specific component of the malware that performs the harmful behavior [20]. The payload does not have to execute immediately; it can remain dormant, waiting for the right opportunity to strike.

Having defined malware and its various purposes, it is crucial to explore how these malicious programs are delivered to their targets.

2.1.2 Attack Vectors

The malicious payload must somehow be delivered to the target. The method used for this delivery is known as the *attack vector*. After the payload is in place, it can be executed in many different ways, and may remain dormant until the right time. Below are some of the most common attack vectors [21]:

- **Social engineering scams.** In social engineering, victims are manipulated into performing actions they should not do, such as downloading and executing malware. There are many ways in which social engineering can aid in performing successful malware attacks. One prominent type of social engineering scam is *phishing*, where attackers deceive people into revealing usernames, passwords, credit card numbers, bank account information or other sensitive data through fraudulent mails, text messages or website forms. According to IBM's X-Force Threat Intelligence Index, phishing takes a role in about 41% of malware infections [21]. Another example is malspam - a portmanteau of malware spam - where spam mails are sent out with malware as the payload, often through links or attachments. Here, social engineering tricks the receiver into opening the links or attachments, thus infecting their system with malware.

- **System vulnerabilities.** Exploiting security holes, or *vulnerabilities*, in software, devices, and networks allows cybercriminals to inject malware into these systems. Sometimes these vulnerabilities are unknown to the owners of the computer system, the developers of the software, or anyone capable of mitigating them. These types of vulnerabilities are referred to as *zero-days*. They are powerful tools for attackers, enabling malware to propagate quickly, defend itself against security mechanisms, or gain a stronger foothold in the system.

- **Removable media.** According to a recent study, 37% of known cyberthreats exploit removable media such as USB drives [21]. Hackers may leave USB drives in public spaces like parking lots, office spaces or coffee shops. Unsuspecting victims plug in these USB drives, thinking they found a free device or wanting to see its contents, thereby infecting their system with malware.

- **Fake software and file downloads.** Malware such as trojans and adware often disguise themselves as free content or useful programs. Users may believe they are downloading a free anti-virus, a performance-enhancing application, or a free copy of a film or song, when in reality, they are downloading malware. Peer-to-peer file sharing networks are notorious for spreading malware, but sometimes malware can even find its way to legitimate marketplaces like the Google Play Store [11].

- **Malvertising, drive-by downloads and watering hole attacks.** Hackers can use advertisements to spread malware by placing malicious ads in legitimate ad networks or hijacking real ads. Unsuspecting users may click on these hacker-controlled ads, leading to malware downloads. For example, the Bumblebee malware used a malicious Google ad posing as Cisco's AnyConnect Secure Mobility Client software [12]. Attackers can also set up malicious websites that automatically start a download when visited, known as *drive-by downloads*. These attacker controlled websites often appear higher in the search results through deliberate manipulation of search engine indexing, known as *Search Engine Optimization (SEO) poisoning*. Methods for SEO poisoning include stuffing a website with irrelevant keywords, linking several attacker-controlled websites to each other, and presenting different content to search engines and users. Additionally, attackers can use *watering hole attacks*, where they compromise a legitimate website frequently visited by employees of a target company to deliver malware [67].

- **Supply chain attacks.** Complex multi-step processes called supply-chains produce and distribute products or services to consumers. This is no different for software. If a vendor high up in the supply chain is compromised, all downstream entities can receive malware. An infamous example is the SolarWinds hack, where the Orion network management software was used to deliver malware to many multinationals and government agencies using the software to manage their IT resources [18].

Once the malware has successfully infiltrated a system, the next challenge for an attacker is ensuring the malware remains active even after system restarts. This is achieved through various persistence mechanisms.

2.1.3 Persistence

After initial compromise, the attacker may want to strengthen their foothold in a system or network. Achieving *persistence* allows malware to continue operating even after reboots, log-offs, and other such events on the infected machine. Several techniques are commonly employed to achieve persistence [4]:

- **Modifying registry keys.** Registry keys are container objects similar to folders that can be found in the *Windows Registry*. The Windows Registry is a hierarchical database that contains information, settings, options and other values for programs and hardware installed for the Windows operating system. Windows has many *Auto-Start Extensibility Points* (ASEP), which allow a program to start without any explicit user invocation and are prime targets for malware. These points are also present in the registry, with the most well-known keys being the *Run keys*. They are located in the `Software\Microsoft\Windows\CurrentVersion\Run` or the `Software\Microsoft\Windows\CurrentVersion\RunOnce` registry path and make a program run when a user logs in. By creating and adding a registry key to the

Run keys, arbitrary payloads can be executed during Windows logon, allowing the malware to re-execute.

- **Shortcut hijacking.** The target location of a shortcut icon can be hijacked by malware to download content from a malicious website, for instance, through drive-by downloads, or to load a malicious browser extension along with launching the normal application. For example, Chromium-based browsers like Google Chrome and Microsoft Edge offer the `-load extension` command-line option, which silently loads the extension at the provided path in the background while launching the browser. This is how the ChromeLoader malware, a type of downloader malware[1], delivers its malicious browser extension payload to the victim's browser.

In addition to establishing persistence, malware often employs techniques to conceal its presence and evade detection.

2.1.4 Evasive Malware

> *All warfare is based on deception. Hence, when we are able to attack, we must seem unable; when using our forces, we must appear inactive; when we are near, we must make the enemy believe we are far away; when far away, we must make him believe we are near.*
>
> Sun Tzu, *The Art of War*

Etymology

Evasive Malware Definition Regardless of the task, malware needs time to perform its operations. To maximize its effectiveness, it is advantageous for the malware to remain undetected and to stall security researchers for as long as possible. Malware that employs techniques to evade detection or delay analysis by tools or researchers is referred to as *evasive malware*.

Evasion Technique Definition Various techniques exist for evasion, collectively referred to as *evasion techniques*. These techniques can be categorized based on the type of analysis the malware seeks to bypass and the specific methods employed. For instance, malware attempting to evade detection within a sandbox will utilize techniques classified under the anti-sandbox category. An up-to-date taxonomy of evasion techniques will be presented in Section 3.4. As context for the upcoming discussion on malware analysis, the obfuscation class of evasion techniques is examined below.

[1] Refer to Section 2.1.6 for an explanation of the different types of malware.

Obfuscation

The Oxford English Dictionary defines obfuscation as "*concealment or obscuration of a concept, idea, expression, etc.*" [16]. In the context of software, obfuscation refers to the practice of making a program's code - such as that of malware - difficult to understand, without altering its functionality. This technique can be used to hide the malicious behavior of the code, helping malware evade detection by security tools (such as antivirus scanners) and complicating analysis for security experts. It can also be used by legitimate software to protect intellectual property or trade secrets by making reverse engineering more challenging[2]. The following categories of obfuscation illustrate the evolving sophistication of malware [68]:

- **Encrypted malware.** In this category, malicious code or data (e.g., a harmful URL) is transformed into an alternative form that obscures its original meaning. This transformation can only be reversed using a secret key. By employing different keys for each instance, the malicious behavior appears different in its encrypted form, making it harder to detect through signature-based methods. Nonetheless, if the decryptor remains unchanged, it can still be identified by antivirus scanners, prompting the development of more advanced obfuscation methods.

- **Oligomorphic malware.** Oligomorphic malware slightly modifies its decryptor with each new generation. While these variations produce a range of decryptor versions, the limited number of possible changes means that these variants can still be detected by signature-based antivirus systems. This category represents an earlier stage of obfuscation sophistication.

- **Polymorhpic malware.** This category employs a range of obfuscation methods to generate numerous distinct decryptors, significantly complicating signature-based detection. The complexity is further increased by tools like "The Mutation Engine", which can transform non-obfuscated malware into polymorphic variants. Although detection remains possible due to the constant code revealed after decryption, it requires executing the malware, which makes detection more challenging. Having realized this weakness, the next evolution of obfuscated malware would attempt to solve it.

- **Metamorphic malware.** Metamorphic malware advances beyond oligomorphic and polymorphic malware by completely altering its code body between iterations while maintaining its functionality. By avoiding methods that expose constant code (such as encryption), metamorphic malware becomes extremely difficult to detect. This presents the highest level of obfuscation sophistication. A notable tool used for obfuscation by malware authors is the *packer*. Originally designed for file encryption and compression, packing tools have been adapted

[2] Reverse engineering is the process of analyzing a device, system, process or piece of software to understand its inner workings, typically with little to no initial insight, through deductive reasoning [65].

by malware creators to apply various obfuscation categories. When a malware file is packed, the result is two components: the packed executable and a stub or wrapper program. The packed executable is either a compressed or encrypted version of the original file, and the stub or wrapper program handles the decompression or decryption process. A significant advantage of using a packer is that even minor changes to the source code can result in drastically different executable files. As a result, different variants of the same malware family may not share recognizable patterns, despite having similar underlying code.

Beyond evading detection, malware also needs to be adaptable to meet specific user requirements and objectives. This adaptability is achieved through configuration.

2.1.5 Configuration

Malware often needs to fulfill specific user requirements, such as exfiltrating data to a particular network address, installing a persistence mechanism in a specific location, or targeting only specific victim machines [15]. To address these requirements, malware authors design their programs to allow for fine-tuning through configuration. This enables the malware to be customized in terms of behavior, interaction, and functionality by adjusting parameters, settings or sets of rules. In practice, configuration serves multiple purposes: it allows the malware to adapt to changing circumstances, evade detection, update its capabilities, tailor itself to specific use cases and target infrastructures, achieve various objectives, and hinder analysis through obfuscation [15].

To fully appreciate the diversity of malware, it is helpful to classify it based on its objectives or the family it belongs to. The following section explores the different types of malware.

2.1.6 Types of malware

Malware can be classified in various ways, with two of the most common methods being by the goals it seeks to achieve or by the malware family to which a sample belongs.

Objective-based Malware is classified based on its goals. Some common types based on their objectives include [21]:

- **Virus.** Attaches itself to legitimate software, spreading similarly to a biological virus infecting a host cell.
- **Ransomware.** Holds a victim's files, devices, or systems hostage, rendering them unusable and inaccessible until the required ransom is paid. For example, it can encrypt files that can only be recovered with a decryption key provided by the attacker after payment.

- **Remote access trojan (RAT).** Grants an attacker access to computers, servers, or other devices from his own device through the creation or exploitation of a backdoor.

- **Botnets.** A network of malware-infected devices, known as *zombies*, under the attacker's control. These devices, ranging from mobile devices and IoT-devices to PCs, are often used to launch distributed denial-of-service (DDoS) attacks, overwhelming a target service with requests to slow it down or render it inaccessible.

- **Cryptojackers.** Uses the host device to mine cryptocurrencies, such as Bitcoin, without the owner's knowledge.

- **Fileless malware.** Injects malicious code directly into the computer's memory by exploiting vulnerabilities, leaving no traces on the file system.

- **Worms.** Self-replicating malware that spreads to other apps or devices without human interaction.

- **Trojan (horses).** Disguises itself as legitimate software or hides within useful programs, creating a backdoor on the system or installing additional malware.

- **Rootkits.** Grants the attacker privileged, administrator-level access to a device's operating system or other assets.

- **Scareware.** Frightens the computer user into installing malware or divulging sensitive information through alarming pop-up messages.

- **Spyware.** Secretly monitors the user and sends gathered information back to the attacker.

- **Adware.** Floods a device with unwanted advertisements.

- **Downloader.** Designed to download other programs onto the infected machine, often additional malware. This type is frequently referred to as a *dropper*, with the act of installing other malware known as *dropping*.

- **Information stealer.** Also known as an *infostealer*, it gathers data, typically credentials, and sends it to the attacker.

Family-based Classifying malware by its family involves grouping samples that share similar properties. These malware samples exhibit a "signature" of similar behavior and characteristics, providing valuable insights into triage, remediation, and attribution efforts for new samples. One of the most prevalent malware families in Q1 2024 is AgentTesla [19]:

- **Agent Tesla.** This .NET-based[3] (pronounced "dot-net-based") RAT specifically targets Windows systems to steal information. Initially available as Malware-as-a-Service (MaaS), where professional cybercriminals conducted attacks as a service with different versions available based on the price paid, its builders[4] were eventually leaked, making it accessible to all threat actors. AgentTesla's signature capabilities include logging keystrokes, accessing the clipboard, capturing screenshots, harvesting saved credentials from browsers or from disk, and sending all this information back to the attacker via various channels or potentially loading other malware.

2.2 Malware Analysis

Creeper's challenge gave rise to another program called "Reaper". Similar to Creeper, the Reaper program propagated throughout the ARPANET, searching for machines running Creeper. Upon detecting Creeper, Reaper would terminate and delete it. Often regarded as the first *antivirus* program, Reaper marked a significant milestone in cybersecurity history [60]. Initially, an antivirus (AV) was a program designed to detect and remove computer viruses. Today, the term encompasses programs that prevent, detect, and remove any type of malware, also known as *anti-malware* software [2].

Whether Reaper qualifies as the first antivirus depends on one's definition, given that Creeper was not malicious and thus not true malware. However, it is undeniably the first *nematode*, a worm or virus that removes another worm or virus from a system [1], representing the earliest countermeasure against unwanted (malicious) programs. As contemporary malware has grown increasingly sophisticated, antivirus software and other security technologies have had to advance significantly to detect these evolving threats. Central to this effort is a thorough understanding of the malware itself, achievable through *malware analysis*.

Definition Malware analysis is the process of investigating a piece of malware to understand its functions, methods of operation, and intentions, with the goal of mitigating its threats. The field of malware analysis can be broadly categorized into two main approaches: *static analysis* and *dynamic analysis*. Each approach provides different insights into the malware and contributes to a comprehensive understanding necessary for effective defensive strategies.

To better understand how malware can be analyzed and the methods used, static analysis will first be explored.

[3] A free, open-source, and cross-platform developer platform by Microsoft for building client, cloud and other applications [36][63].

[4] A tool for designing and constructing malware, allowing easy creation of robust and evasive malware by providing an interface for selecting desired traits [22].

2.2.1 Static Analysis

In static analysis, a malware program is analyzed without executing it. This approach allows for analysis to be performed without a live environment, making it both simple and fast.

Static Analysis Techniques

Patterns indicative of malicious or evasive behavior can be extracted through various methods, including Windows API calls, string signatures, control flow graphs, opcode frequency, and byte sequence n-grams. Other techniques like file sizes, function lengths and network behavior, are also utilized [55]. The most relevant methods, specifically Windows API calls and string signatures, are discussed in further detail below:

- **Windows API calls.** Programs use the Windows API to communicate with and perform specific tasks within the operating system. This means API calls reveal a lot about the behavior of a program and thus are a staple technique in understanding malware behavior. For instance, a combination of the `WriteProcessMemory`, `LoadLibrary` and `CreateRemoteThread` API calls is indicative of a technique known as *DLL injection*, as these calls are rarely seen together in benign instances.
- **String signatures.** The presence of certain strings can also provide clues about the malware's behavior. Specific strings are sometimes necessary for certain actions, which can reveal the malware's intent. For example, strings such as "`KVMKVMKVM\0\0\0`", "`Microsoft Hv`", "`VMwareVMware`", "`XenVMMXenVMM`", "`prl hyperv  `", and "`VBoxVBoxVBox`" are known hypervisor[5] signatures. The presence of these strings suggests that the malware may be attempting to determine if it is running in a virtualized analysis environment.

To effectively analyze malware, one must understand the limitations inherent in static analysis techniques. While static analysis provides critical insights through methods like examining Windows API calls and string signatures, it does not capture all aspects of malware behavior.

Limits to Static Analysis

There are limitations to using static analysis for malware analysis. Since static analysis involves examining the code without executing it, sophisticated malware can employ obfuscation techniques or include malicious actions that only become apparent during runtime. This can cause such malware to evade detection by traditional static methods. For example, malware might split a string needed for a malicious operation,

[5] A hypervisor is a type of software, firmware or hardware that allows a system to create and run virtual machines (virtualizations or emulations of computer systems that provide the functionality of a physical computer; essentially a computer running within a computer) [61].

such as downloading a harmful file, and then reassemble it only during execution. This tactic allows the string to bypass static analysis undetected.

Moser et al. have highlighted these limitations by demonstrating that static analysis has fundamental limits. They translated this problem into a 3SAT[6] problem, illustrating that analyzing some malware samples statically is provably difficult [47]. While it is possible to enhance static analysis techniques to address some advanced obfuscation techniques, these limitations indicate that static analysis alone may not be sufficient.

To address these limitations, dynamic analysis provides an alternative approach, where the malware is executed in a controlled environment to observe its behavior in real-time. For this reason, for the remainder of this thesis, the focus will be on dynamic analysis, or the hybrid approach of combining both static and dynamic analysis, known as hybrid analysis.

2.2.2 Dynamic Analysis

In dynamic analysis, the malware program is executed in a controlled and safe environment. There are several safe environments, or *layouts* to choose from, each paired with specific analysis tools that perform the actual analysis [49]. These layouts differ in their level of transparency - essentially, how effectively they conceal the fact that they are being used for analysis. In other words, a system is more transparent if it reveals fewer indications of being an analysis environment.

Building on the understanding that dynamic analysis involves executing malware in controlled settings, it is crucial to examine the various environments used for this purpose in order to select an appropriate layout for the use case presented in this work.

Dynamic Analysis Environments

Common layouts for manual or automated dynamic analysis include bare metal, virtual machines (on a type2 hypervisor) with in-guest/in-host components, virtual machines on a type 1 hypervisor with analysis components in VMX root, full system emulation, and containers. These layouts are described in more detail below.

Bare metal In a bare metal configuration, malware is executed on a machine with an unmodified operating system running on real hardware. This setup makes it nearly indistinguishable from a real host computer. Essentially, it is equivalent to unpacking a new system, installing an OS and, optionally, an analysis tool on it, and then running the malware. Alternatively, the analysis tool can be positioned outside the bare metal instance, focusing on disk and network level behavior, thus enhancing transparency [43].

[6] Also known as the boolean satisfiability problem. It is a problem belonging to the NP-complete class of problems in computational complexity theory [58], which is a class where solutions can be verified quickly, but cannot be found quickly [64].

Virtual Machine When analyzing malware within a virtual machine (VM) or a type 2 hypervisor, the separation between the guest's kernel (the guest being where the VM runs) and the host OS (the host being the native system) provides protection for the host. This setup facilitates quick and reliable restoration of the environment to a clean state after each analysis, offering a significant advantage over bare metal systems. The analysis tool can be deployed in one of two ways: either inside the guest VM or on the host machine. If installed within the guest VM, the tool will have user mode privileges, limiting its use to user mode malware. Conversely, placing the analysis tool on the host machine allows for monitoring of malware behavior from outside the guest VM using Virtual Machine Introspection (VMI). VMI involves an in-guest component, typically a kernel driver, that tracks how the malware interacts with the guest OS. However, there are notable risks: if the malware detects the analysis tool or kernel driver within the guest, it may identify the virtual environment and alter its behavior to avoid detection. Additionally, vulnerabilities in the virtual machine software could potentially allow the malware to escape the guest environment and compromise the host system, thus threatening the entire analysis platform.

Hypervisor A type 1 hypervisor allows multiple operating systems to run concurrently on the same physical hardware, each within its own virtual machine (VM). Resource sharing is managed by the hypervisor, which operates in VMX root mode on the host kernel, while the guest operating systems run in VMX non-root mode on their respective guest kernels. This separation provides a robust mechanism for running and analyzing malware. When a guest OS attempts to access hardware resources, control is transferred to the hypervisor in VMX root mode, which processes the request and then returns control to the guest. This setup enhances the difficulty for malware to detect the analysis environment, as the hypervisor holds all control and can even manipulate values within the guest VMs, thereby concealing its presence and activity more effectively.

Emulation In full system emulation, a software tool called an emulator replicates hardware at a low level, creating a virtual environment that simulates the operation of a physical system. This method enables the analysis of malware behavior without the need to run it on actual hardware. Similar to the setup using virtual machines, the emulator hosts a guest VM, but instead of a type 2 hypervisor an emulator is used. Additionally, the emulator can restore the operating system's state from a memory dump if required, providing a robust means of observing malware behavior in a controlled environment.

Containers Containers provide another layer of separation between applications and the operating system, enabling the development and execution of applications across various operating systems. However, unlike virtual machines, containers share the host machine's OS kernel and are not completely isolated from each other. This reduced level of separation can be a concern when running malware. Although this

layout has not been extensively studied, experimental results indicate that it shows promise as an analysis environment [41].

These layouts can be utilized for both manual or automated analysis. When a layout automatically analyzes a program within a contained and isolated environment, it is referred to as a (malware analysis) *sandbox*. Some other layouts build upon the previous ones to perform the analysis in specific ways. These layouts are briefly discussed below.

Volatile Memory Extraction. The volatile memory, this is the random access memory (RAM), of the OS can be analyzed for malware behavior without the need for an in-guest component. By avoiding the use of an in-guest component, the acquisition process does not leave any clues that malware can detect, provided the memory dumping process is performed quickly enough to avoid time-based detection techniques. The primary drawback of this layout is that continuous memory dumps are not feasible, only periodical ones are possible. This means that the granularity of single instructions is not achievable, and malware that completes its execution before the next dump may go undetected.

Side Channel Data Extraction. A side channel is an unintentional communication channel through which information leaks due to the fundamental properties of electronic devices [66], such as power consumption and electromagnetic (EM) emission. By analyzing these side channels, a behavioral pattern of the system's operation can be established and examined for signs of malware behavior. This layout is the most transparent among those listed because data acquisition is completely transparent to both the malware and the OS. By avoiding any in-guest component and performing data analysis on a separate machine, the risk of time-based detection is minimized, as it incurs almost no additional overhead.

Analysis Techniques

While a safe environment is crucial for dynamic malware analysis, it is equally important to gather and scrutinize relevant information to accurately identify the behavior of a sample.

Function Call Analysis Processes utilize functions, whether internal or external (e.g., system calls), to perform their designated tasks. By employing a technique called *hooking*, the functions invoked by the malware and their parameters can be monitored. Functions can be hooked through the Windows OS hooking mechanism or by using one of many code injection techniques. Typically, code is inserted at the beginning of the function to alert the analysis process when the current (hooked) function is called. Once notified, the analysis process can take additional steps to gather further information from the OS about the function and its parameters by accessing the stack of the process.

Execution Control Execution control involves a suite of techniques that periodically halt execution to inspect the state of the (malware) process and the OS. These techniques include debugging, binary instrumentation, dynamic forward symbolic execution and multiple path exploration.

Flow Tracking The flow of information, such as results being passed between functions, is tracked throughout the execution of the (malicious) process. Techniques used for this purpose include data tainting, control flow tracking, information flow tracking, among others.

Tracing Analyzing clues or *traces* left after execution can provide insights into the (malicious) process, without the need for an in-guest component. Examples of this include network tracing and volatile memory forensics.

Side-channel Analysis Side-channel analysis examines the behavior of what is running on a device through side-channels on the physical components of the device, such as power consumption, electromagnetic emissions, or internal CPU events. Because this method does not offer deep insights into internal processes, the extracted behavioral data can be classified as either "normal behavior" or "infected behavior".

Having explored various dynamic analysis environments, hybrid analysis, which integrates both static and dynamic techniques to enhance malware detection and understanding, will now be examined.

2.2.3 Hybrid Analysis

Though mentioned that there are only two broad categories of analysis techniques, these techniques are not mutually exclusive. Combining static and dynamic analysis techniques is sometimes referred to as *hybrid analysis* [32], and allows to cover the shortcomings of one category using the other, enabling static analysis of malware that employs runtime malicious behavior and obfuscation [14].

Given the benefits of hybrid analysis, which combines static and dynamic approaches to provide a comprehensive view of malware behavior, it is essential to choose a tool that embodies these strengths. CAPEv2 sandbox is a prime example, offering a sophisticated form of hybrid analysis that effectively integrates static signature-based detection with dynamic behavioral analysis.

Moreover, CAPEv2's open-source nature and popularity within the cybersecurity community, including its adoption by companies for custom sandbox solutions [25][40], further justify its selection for this research. The widespread use of CAPEv2 not only ensures its reliability and effectiveness but also facilitates the potential integration of this research into existing security infrastructures, enhancing its practical relevance.

By leveraging CAPEv2's advanced hybrid analysis capabilities, this research aims to accurately detect and categorize evasive malware, contributing valuable insights and tools to the field of malware analysis.

The following section will delve into the functioning of CAPEv2, detailing its analysis flow and the various components involved. This exploration will provide the necessary background for understanding the tool's capabilities and limitations, thereby setting the stage for a deeper comprehension of the enhancements introduced in this thesis.

2.3 CAPEv2

In 2015, Kevin O'Reilly, while working at Context Information Security, began developing a command-line configuration and payload extraction tool called *CAPE* (an acronym for "Configuration And Payload Extraction") [50]. Originally, the tool utilized API hooking through Microsoft's Detours library to capture and extract unpacked malware payloads and configurations. However, these methods proved inadequate in both power and precision when dealing with arbitrary malware. Consequently, research commenced on a novel type of debugger that would allow for precise instrumentation and control of target malware while maintaining a high level of stealth by avoiding the use of Microsoft's well-known debugging interfaces.

This novel debugger was eventually integrated into the command-line tool, greatly enhancing its practical capabilities. With the aim of replacing the Microsoft Detours API hooking mechanism, the project incorporated a modified API hooking engine originally developed for Cuckoo Sandbox. This integration lead to the creation of *CAPE Sandbox*, which combined parts of Cuckoo Sandbox with the new debugger, automated unpacking, integrated configuration extraction and YARA-based classification.

Since its inception, the project has evolved significantly. It was ported to Python 3, resulting in the current version known as *CAPEv2*, which includes numerous advanced features such as dynamic detonation of CAPE's debugger using YARA scans, an interactive desktop, Anti-Malware Scan Interface (AMSI) payload capture, *system call* hooking using *Windows Nirvana*, and debugger-based countermeasures for both direct and indirect system calls [10]. Throughout this thesis, the term "CAPE" will refer to the latest instance of *CAPEv2*.

Having explored the development and evolution of CAPEv2, it's now crucial to delve into what CAPE actually is and how it functions within the realm of malware analysis.

2.3.1 What is CAPE?

So, what exactly is CAPE? According to the documentation [3]:

> "*CAPE is an open-source automated malware analysis system.*"

CAPE enables the automatic execution of malware within an isolated environment, such as a Windows virtual machine (VM), and generates a detailed report that outlines the malware's behavior. This report includes the following information [3]:

- Traces of API calls made to the OS, such as win32 API calls, performed by processes spawned by the malware.
- Files that were created, downloaded, or deleted by the malware during its execution.
- Memory dumps of the processes spawned by the malware.
- Network traffic traces captured in PCAP format.
- Screenshots of the desktop taken during the malware's execution.
- A full memory dump of the machine.

Given that CAPE runs the samples in a contained environment and automatically generates an analysis report, it can be classified as a sandbox. However, CAPE goes beyond just dynamic analysis; it includes several static analysis features that are applied to artifacts such as memory dumps collected during the execution of the malware. This combination of static and dynamic capabilities places CAPE within the category of tools used for hybrid malware analysis, as previously discussed.

To understand how CAPE operates and manages the analysis process, it's important to delve into its architectural components and workflow.

2.3.2 Architecture

CAPE's architecture consists of a host machine, running central management software, and some guest (virtual) machines used for analysis [3]. The central management software forms the core of the sandbox, overseeing the entire analysis process and executing samples in the isolated, safe environments provided by the guest machines. The following image illustrates the architecture of Cuckoo Sandbox, which is analogous to CAPE's architecture [3]. This overview provides a high-level understanding of the roles of the host and guest machines.

To gain a deeper insight into how CAPE functions, the analysis flow will now be examined in more detail [13].

Analysis Flow

The analysis flow outlines the complete analysis process, beginning with the submission of an analysis target and culminating in the generation and reporting of comprehensive results results.

Submission A URL or path to a file or folder is submitted, which triggers the creation of an entry in a central *database*. This entry specifies the type of *analysis package* to be used for the target, as well as some analysis specific options such as the critical timeout.

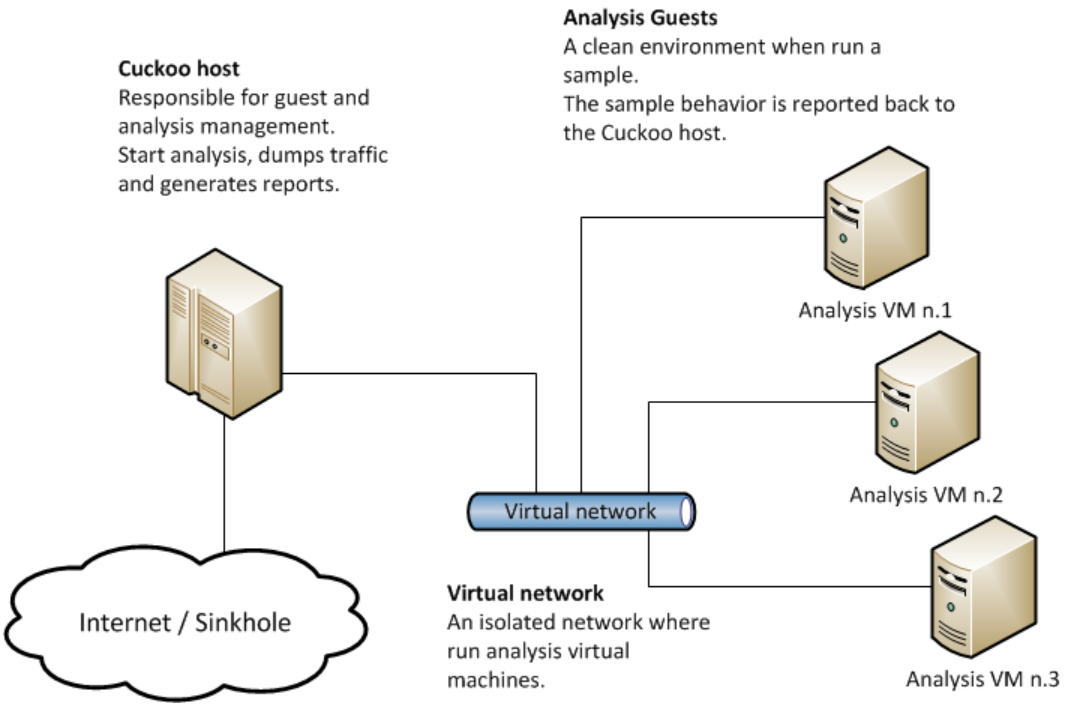


FIGURE 2.2: Main architecture [3]

Scheduling Tasks The *task scheduler* component continuously monitors for pending tasks and assesses the availability of appropriate guest machines. When a suitable machine is identified, a task is chosen based on its priority, and the selected task is then handed to the *analysis manager*.

Starting Analysis Once the analysis manager receives the selected task, a suitable VM from the available pool is chosen for the task, and the analysis is initiated. One of the initial steps in commencing an analysis involves notifying the *result server* of the new task.

Prior to starting the guest machine, *auxiliary modules*, such as the *human module* which simulates human behavior through mouse movements and clicks, are activated. After these modules are operational and the guest machine is up and running, the *guest manager* uploads the *monitor*, *analyzer* and *configuration* and optionally the target file to the agent within the guest. The agent then launches the analyzer, which subsequently starts or opens the target (file) and injects the monitor into the spawned process.

Stopping analysis Throughout the execution of the target, the monitor and analyzer relay the collected behavioral information back to the *result server*. The analysis manager waits until the guest manager determines that the target has been analyzed - either by verifying that the analyzer has stopped or by reaching the critical timeout. Following this, the guest machine and all auxiliary modules are terminated.

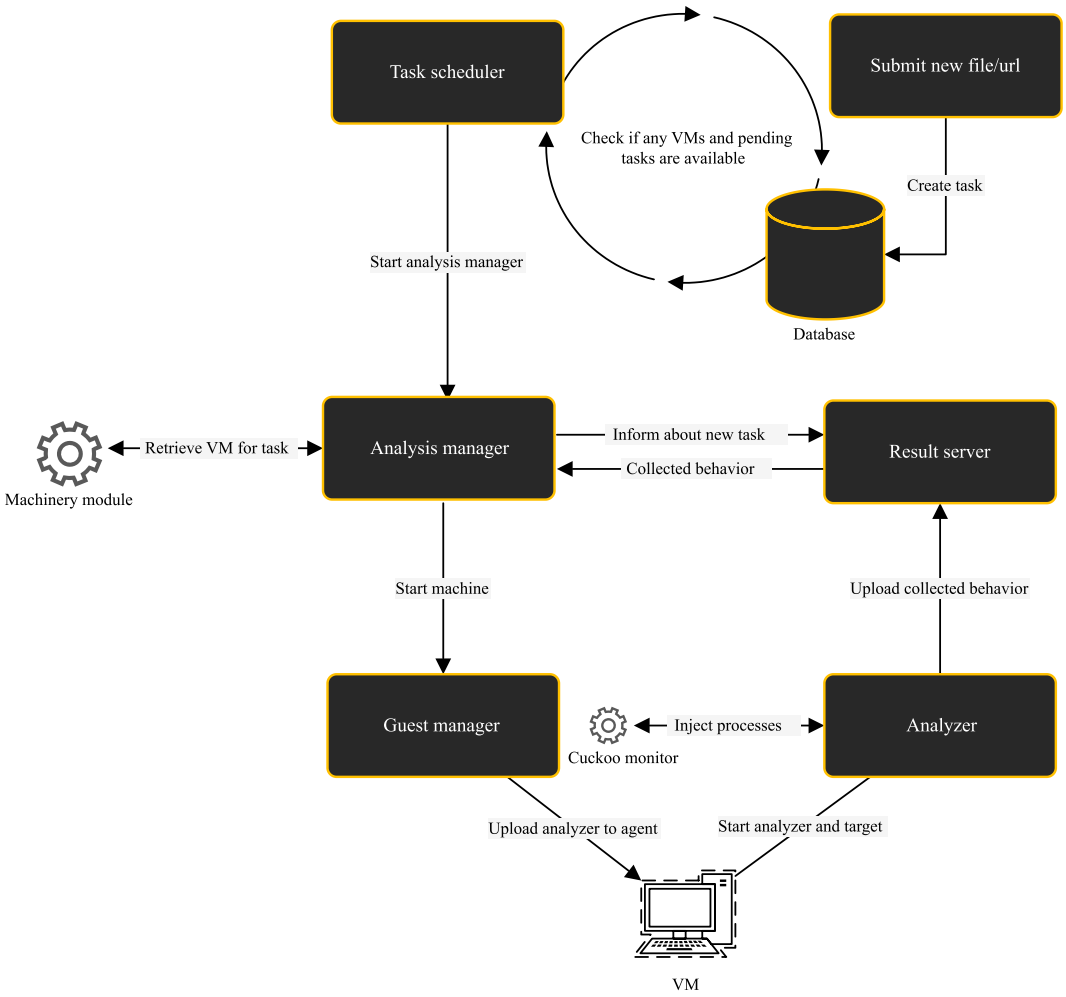


Figure 2.3: Analysis Flow [13]

Processing Results Once the guest and auxiliary modules have been stopped by the analysis manager, the processing modules begin analyzing the collected information and generating usable results. Signature matching for various patterns of malicious behavior is then conducted on the results. Finally, the reporting modules compile and summarize the findings in a comprehensive analysis report in an accessible format such as JSON.

The entire analysis flow is depicted in the Figure 2.3. Although this figure illustrates the analysis flow for Cuckoo Sandbox, it is important to note that the flow for CAPEv2 is essentially the same, with the primary difference being the name of the monitor. In CAPEv2, the monitor is referred to as “capemon” rather than the “Cuckoo monitor”.

The following section will provide a detailed overview of the components involved,

including some that are not shown in the diagram.

Components

The CAPE malware analysis sandbox is a complex tool that comprises many components, each with their own specific tasks in the analysis process. These components are detailed below.

Machinery Modules Being initialized by the Task Scheduler, the purpose of machinery modules is interacting with the hypervisor or the physical machine. More specifically, a machinery module is used to manage all the guests, through for instance starting, stopping or restoring the guest machines to a clean state, while CAPE is running.

Task Scheduler The task scheduler is responsible for initializing the (singular, as per configuration) machinery module and starting new tasks when sufficient resources are available and the maximum number of running guest machines has not been exceeded. This component must operate continuously. It regularly checks for available guest machines and pending tasks, then selects a task and provides the task's information to the analysis manager.

Auxiliary Modules Auxiliary modules define procedures that must be executed in parallel to the analysis process [3]. These modules are started prior to the initiation of the analysis machine (the guest) and are stopped at the conclusion of the analysis process, before the processing procedure begins. For example, the *sniffer module* is responsible for creating a dump of the network traffic generated during the analysis [3].

Analysis Manager Initiated by the task scheduler, the analysis manager oversees the entire analysis flow of the given task. This component is responsible for decisions regarding the starting and stopping of machines, as well as the initiation of other modules as permitted. Upon being started by and receiving task information from the task scheduler, the analysis manager identifies a suitable guest machine based on the task's requirements. For example, a task may require to be ran on a specific machine or environment because of operating system requirements or the need for specific software to open the file. It then starts the required auxiliary modules and informs the result server of the new task before starting the guest machine for analysis. Once the guest machine is running, the guest manager takes over the remainder of the analysis flow until the analyzer completes its task or a critical timeout is reached.

Agent process The agent is an HTTP server that operates on the guest machine, tasked with receiving uploaded files - the analyzer, monitor, configuration, and potentially a target file - from the guest manager. It's primary function is to initiate processes, particularly starting the analyzer. The agent process is expected to be started automatically when the guest machine is initialized.

Analysis package Components that define the method by which CAPE's analyzer should perform the analysis of a target [3]. For example, the *EXE package* is used for analyzing generic Windows executables, which involves executing them, optionally with specified arguments [3].

Analyzer The analyzer contains all the logic and the required auxiliary modules for the analysis. A platform-specific version of the analyzer (either x86 or x64), matching the analysis machine, is selected by the guest manager and executed within the machine by the agent process. Once initiated, the analyzer searches for the configuration file containing information about the target. It then selects the appropriate analysis package or attempts to identify the best one based on the provided details. Before beginning the analysis, the required auxiliary modules are started. When the analysis on the target begins, a process spawned by the target is injected with the monitor component, known as *capemon*. This Dynamic Link Library (DLL) logs behavior through function hooking, process tracking, and other methods. It also incorporates much of CAPE's core functionality, such as its novel debugger, payload extraction, process memory dumping, and import reconstruction [9]. The analyzer transmits all collected data to the result server for storage and further processing, continuing as long as target processes remain or until the critical timeout is reached.

Guest manager The guest manager communicates with the agent on the guest machine to upload the analyzer, monitor, configuration and target, and then initiates the analyzer. Once the analyzer is running, the guest manager enters a waiting state, repeatedly querying the agent to see if the analyzer has reported that it has completed its work. If the analyzer does not finish by the critical timeout, it is forcefully stopped.

Result server Before starting the guest machine for analysis, the analysis manager registers the guest machine's IP address and the task ID with the result server. The result server itself is then responsible for receiving and handling the incoming data streams from the analyzer, ensuring they are stored in the correct format and location for further processing.

Processing modules These modules are tasked with processing the collected behavioral data. Processing involves analyzing the raw generated results and appending information to a data structure known as the *global container*, which allows this information to be used by signatures and reporting modules [3]. The tasks performed by processing modules include parsing raw behavioral logs, performing initial transformations and interpretations, running memory forensics tools on full system memory dumps, extracting and parsing network information from the network dump, analyzing process memory dumps and conducting static analysis [3].

Signatures Once processing is complete and the data has been added to the global container, signatures are run against this data. These signatures are used to identify malicious behavior or other indicators of interest based on predefined patterns, which are then incorporated into the final set of results. For example, signatures in CAPE can identify specific malware families, detect system modifications, or classify malware into categories such as ransomware, RATs, or cryptojackers [3]. A wide range of signatures is provided by CAPE administrators and users through the CAPESandbox Community repository [28].

Reporting modules. After the raw results have been processed and the global container has been generated, the reporting modules utilize the global container to present the final results in various formats suitable for different purposes. For example, a JSON module converts the global container into a JSON format and writes it to a file [3].

2.4 Related Works

This section provides a comprehensive overview of the relevant literature and advancements related to malware analysis, evasion techniques, and detection methodologies. By examining various taxonomies of malware classification, analysis environments, and evasion strategies, this review highlights the evolution and current state of research in these fields. The focus is on understanding the diverse approaches taken by different researchers to classify and address malware behaviors and the techniques used to evade detection. Key contributions from notable works are discussed, including advancements in automated analysis tools and detection mechanisms that offer insights into the effectiveness of various strategies against modern evasion techniques.

2.4.1 Malware Classification

Or-Meir et al. present a comprehensive survey on dynamic malware analysis, detailing various methods for classifying malware, including by type, malicious behavior, and privilege level [48]. The distinction between type and behavior is particularly noted, with type classifications commonly used in mass media and behavior-based classifications preferred by malware professionals. The classification by privilege, which refers to the privilege level the malware executes with (such as user mode or kernel mode), is less commonly used in public discourse.

A thorough classification of dynamic analysis evasion techniques is provided by Afianian et al., which has played a pivotal role in shaping the taxonomy used in this thesis [23]. Their work offers detailed categorizations of anti-debugger and sandbox evasion techniques, distinguishing between those that are detection-dependent - relying on the environmental detection - and those that are detection-independent, where the malware behaves consistently regardless of the environment.

A significant contribution by Chen et al. was the first to categorize evasion techniques targeting monitoring systems based on virtualization and debugger characteristics [31]. Their taxonomy groups fingerprinting techniques by system abstraction,

including hardware, environment, application and behavioral categories. These are further subdivided based on specific virtualization or debugging characteristics, such as device, drivers, memory, system, installation, execution, and timing. In addition to presenting this taxonomy, the authors introduce a novel technique to detect remote network virtual machine monitoring systems and propose an innovative approach to deceive evasive malware by making target systems appear as monitoring systems. This strategy effectively protects these systems, as the malware neutralizes its malicious behavior upon detecting what it perceives to be an "analysis environment".

Sharma et al. offer a taxonomy of evasion techniques used by advanced persistent threat (APT) malware, categorized into stealth mechanisms, anti-analysis mechanisms, and covert communication [54]. Stealth mechanisms aim to maintain a prolonged foothold on the target by concealing the malware, such as hiding within a legitimate process. Anti-analysis mechanisms detect the analysis environment to avoid detection. Covert communication refers to methods that allow the malware to communicate stealthily with command and control servers.

Geng et al. divide evasion techniques into transformation-based, concealment-based, attack-based, and combined strategies [35]. Transformation-based techniques are obfuscation-related, including methods like packing and metamorphism. Concealment-based techniques aim to evade detection by analyzing the environment. This category includes the subcategories of environment analysis, delayed execution, reverse Turing tests, and conditional execution. These techniques are context-dependent. Similarly, attack-based techniques, which focus on exploiting vulnerabilities or attacking detection mechanisms, also vary depending on the environment.

A comparative analysis of previous taxonomies is conducted by Galloro et al., leading to their own proposed taxonomy [34]. Their classification groups anti-VM and anti-sandbox together as anti-VM, and anti-debugging and anti-instrumentation techniques, as anti-debugging. Additionally, Galloro et al. developed a tool based on dynamic binary instrumentation using Intel Pin to analyze the most common evasion techniques, correlations between evasion techniques and malware families, and trends in evasion technique usage over the past decade.

2.4.2 Malware Analysis Environments

Or-Meir et al. present a taxonomy of malware analysis environments and related academic tools in [48]. This taxonomy categorizes different analysis layouts, focusing on how malware samples, hardware, operating systems, and analysis tools are organized within these environments. Detailed explanations of these layouts can be found in Section 2.2.2.

Bulazel et al. discuss various system architectures used for malware analysis, outlining their advantages, disadvantages, and overall transparency [27]. They also list common techniques that evasive malware may employ to detect each type of environment. Additionally, the paper explores mitigation strategies for these evasion techniques, addressing how they can identify an environment as one intended for malware analysis.

2.4.3 CAPE Evasion Techniques

The most pertinent work on evasion techniques is that of Skuratovich et al., who present various evasion techniques primarily designed to target Cuckoo Sandbox [30]. Given the similarities between Cuckoo and CAPE, some of the techniques presented for Cuckoo are also applicable to CAPE. They also propose countermeasures for these techniques and developed InviZzzible, a tool for detecting malware analysis environments. In this thesis, InviZzzible was used as an example of an automated tool for identifying undetected evasion techniques in CAPE.

2.4.4 Evasion Detection

In [43], an automated system for detecting evasive malware is proposed, which relies on malware analysis on a bare-metal system with no in-guest monitoring component, an emulation-based system and two virtualization-based systems. This system transparently extracts behavioral profiles from disk and network level activity on the bare-metal system and performs hierarchical similarity-based behavior comparisons across different analysis systems to identify evasion. Although the dataset used in this study was not publicly released, it was obtained from the authors and includes five categories of evasion techniques: hardware id-based evasion, bios-based evasion, processor feature-based evasion, exception-based evasion, and timing-based evasion.

Polino et al. introduce ARANCINO in [53], a framework built on top of the dynamic binary instrumentation tool Intel Pin. ARANCINO enhances the stealth of Intel Pin against anti-instrumentation techniques and addresses four classes of such techniques by using countermeasures: code cache artifacts, environment artifacts, just in time compiler detection and overhead detection. By leveraging ARANCINO, the authors developed a generic, dynamic, and evasion-resilient unpacker capable of analyzing malware employing both anti-instrumentation and anti-static-analysis techniques. This allows to analyze malware that employs both anti-instrumentation and anti-static-analysis techniques. The dataset used in their experiments is publicly available. [53].

In [46], the dynamic sandbox reconfiguration tool MORRIGU is proposed. This tool automates the rapid reconfiguration and deployment of virtualized testing environments, enabling the testing of various configurations to trigger different evasive malware. For example, MORRIGU includes a “Registry Edit” configuration that modifies the registry to obscure virtualization artifacts, thereby triggering evasive malware that checks for these artifacts [46].

Although [37] focuses on evasion detection for benign samples, the detection tool called PINvader that is based on the work of [34] is applicable to malware as well. It uses Intel Pin to instrument the target binary with function-level hooks, instruction-level hooks, and syscall-level hooks to monitor the behavior of the program.

Kim et al. develop EvDetector in [42], a tool that detects evasive malware by hooking APIs using Intel Pin. EvDetector uses a compiled list of APIs commonly employed for evasion. It monitors each invoked API call to check if it matches an entry in this list. When a relevant API is detected, the tool inserts hooking code

to trace the use of dynamically-allocated buffers of interest. The tool then analyzes how these buffers influence conditional branches affecting the execution flow. A large-scale study using EvDetector revealed that 8% to 21% of over 700,000 analyzed malware samples could detect sandbox-based analysis.

Finally, [52] investigates evasion techniques using API calls. The study leverages anti-analysis signatures from Cuckoo Sandbox to assess the prevalence of evasion techniques in malware. Given the original Cuckoo Sandbox project has been largely abandoned, CAPE represents a promising alternative that has gained significant attention in recent years. While Cuckoo Sandbox has seen limited updates, with only an unofficial development version by Estiona's national CERT (CERT-EE) [29] still active, CAPE has emerged as notable alternative in the field.

2.5 Conclusion

In conclusion, the chapter has established the necessary context for the approach to detecting evasive malware. The discussion encompassed the various malware analysis methods, including static, dynamic, and hybrid techniques, with an emphasis on their benefits and limitations. The functionalities and relevance of CAPEv2 as the chosen sandbox environment were detailed. Additionally, significant contributions in the related works section were reviewed, covering various taxonomies of evasion techniques, the capabilities and limitations of different malware analysis environments, and prominent tools and frameworks for detecting evasive behaviors. This review underscored the importance of continually evolving analysis techniques to counter increasingly sophisticated evasion methods.

The information provided sets the stage for the next chapter, where a novel approach to identifying currently undetected evasion techniques within the analysis environment will be introduced. Using this approach, a specialized dataset of evasive malware will be constructed, aiming to address existing gaps in research resources. These upcoming contributions are expected to significantly advance the field by providing an approach and specialized dataset that support further research into malware evasion and detection.

Chapter 3

Methodology

As modern malware continues to evolve in sophistication, it becomes crucial to enhance the effectiveness of existing analysis tools. This chapter first outlines the methodology for identifying evasion techniques that remain undetectable by current analysis environments, specifically CAPEv2, and for developing detection mechanisms to address these gaps. The effectiveness of these new detections will be assessed in the subsequent chapter. Second, this chapter details the methodology used to construct a database of evasive malware samples. Each sample in the database is annotated with the evasion techniques from an up-to-date taxonomy, which is introduced here. The creation of this database represents a significant contribution of this thesis.

3.1 Overview

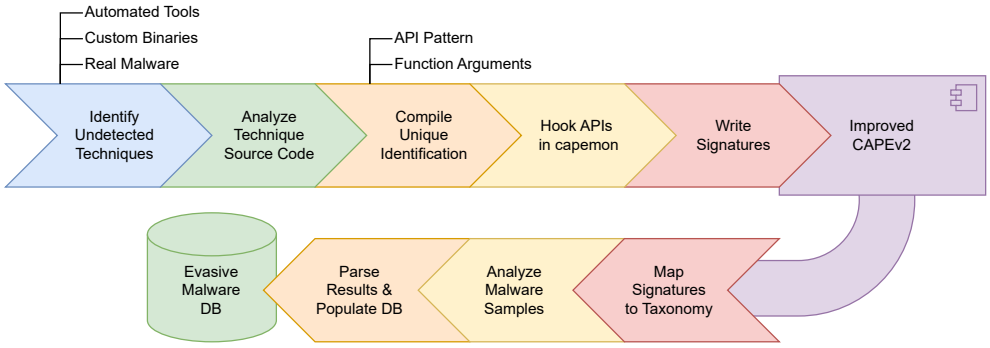


FIGURE 3.1: Overview of the Methodology

Figure 3.1 provides a visual overview of the methodology detailed in this chapter. The key steps of the process are briefly outlined below, offering a concise guide to the approach taken in this thesis.

Step 1: Identify Undetected Techniques This step involves identifying the techniques that CAPE currently fails to detect. Three generic approaches are

proposed for this task, which are adaptable to various analysis tools: utilizing automated malware analysis environment detection tools, utilizing custom binaries that employ a single evasion technique, and analyzing real-world malware samples that successfully evade the analysis tool. A detailed explanation of these approaches is provided in Section 3.2.

Step 2: Analyze Technique Source Code After identifying undetected techniques, the next step involves a thorough analysis of their source code to understand the underlying mechanisms. Particular focus should be placed on the usage of specific API functions, including their arguments and return values, to gain a comprehensive understanding of how these techniques operate.

Step 3: Compile Unique Identification With a clear understanding of the the technique's operations and mechanisms, the next step is to identify the elements essential for its unique detection. By focusing on distinctive aspects such as API call patterns and specific arguments, a detection scheme can be conceptualized that leverages these unique characteristics to reliably identify the technique.

Step 4: Hook APIs in capemon In this step, the relevant API functions are hooked within capemon to ensure they appear in the report. This allows the unique API patterns and specific function arguments to be used for detecting the evasion technique. The process of hooking API functions using capemon's hooking engine is detailed in Section 3.3.

Step 5: Write Signatures To enable the detection of evasion techniques within the analysis results, the processor requires signatures that can identify specific patterns in the report. These signatures are crafted based on the unique identification compiled in step 3. The general structure and functioning of a signature are explained in Section 3.3. This process culminates in an enhanced version of CAPE.

Step 6: Map Signatures to Taxonomy To construct the database of exclusively evasive malware, the signatures in CAPE - both for existing evasion techniques and the newly identified undetected techniques - are mapped to the taxonomy that is provided in this section. This mapping is currently stored in an SQLite database, offering flexibility for adjustments to new taxonomies if needed.

Step 7: Analyze Malware Samples Using the enhanced CAPE tool, a dataset of malware samples is analyzed. Each analysis result will include all signatures triggered by the sample. If any mapped signatures have been triggered, the malware will be added to the database. This step can be performed before or after the previous step.

Step 8: Parse Results & Populate DB After the complete analysis of the malware dataset, the results are be parsed. Triggered signatures are collected, and if a mapping exists for any of them, a node for the malware is created and linked to the corresponding technique from the mapping. The triggered signature is added as a property to the edge connecting the malware node to the technique, serving as evidence of the association.

3.2 Identifying Undetected Techniques

The analysis tool under scrutiny for detecting undetected techniques is CAPEv2, an open source malware analysis sandbox known for its extensive use of signatures [28] for detecting evasive techniques. Several approaches have been developed to identify techniques that elude detection by CAPEv2: automated tools designed to detect malware analysis environments, custom binaries created to evaluate techniques based on existing literature, and the identification of malware samples or families that employ specific evasion techniques. Each of these approaches will be discussed in terms of its advantages and challenges in the following sections. It is important to note that the focus is on identifying existing techniques that evade detection, as the creation new techniques specifically targeting CAPEv2 falls outside the scope of this work.

3.2.1 Using analysis environment-detecting tools

Analysis Environment-Detecting Tools Several automated, open source tools are available for assessing the transparency of a VM or sandbox. Common examples include Pafish [51] and al-khaser [45]. Additionally, InviZzzible [56], which may be less widely known, will also be examined. These automated tools are designed to execute a wide range of evasion techniques aimed at detecting analysis environments, similar to the methods employed by actual malware. They report the results of these detections on-screen. A result of "succes" indicates that the analysis environment was correctly able to hide itself, causing the evasion technique to fail. Conversely, a result of "failure" means that the analysis environment does not adequately conceal itself, allowing the evasion technique to succeed. Figure 3.2 displays a sreenshot of Al-Khaser running within CAPE to illustrate.

Such tools enable users to evaluate the transparency of their analysis environments and determine whether additional countermeasures are needed against evasive malware. They offer an effective means to test numerous evasion techniques and report their efficacy without the concerns associated with running real malware. Furthermore, being open-source allows for inspection of the code to understand the implementation of these techniques. However, challenges include the difficulty of isolating and compiling tools to test individual techniques and potential restrictions on adding custom techniques, depending on the tool's implementation and use of its techniques.

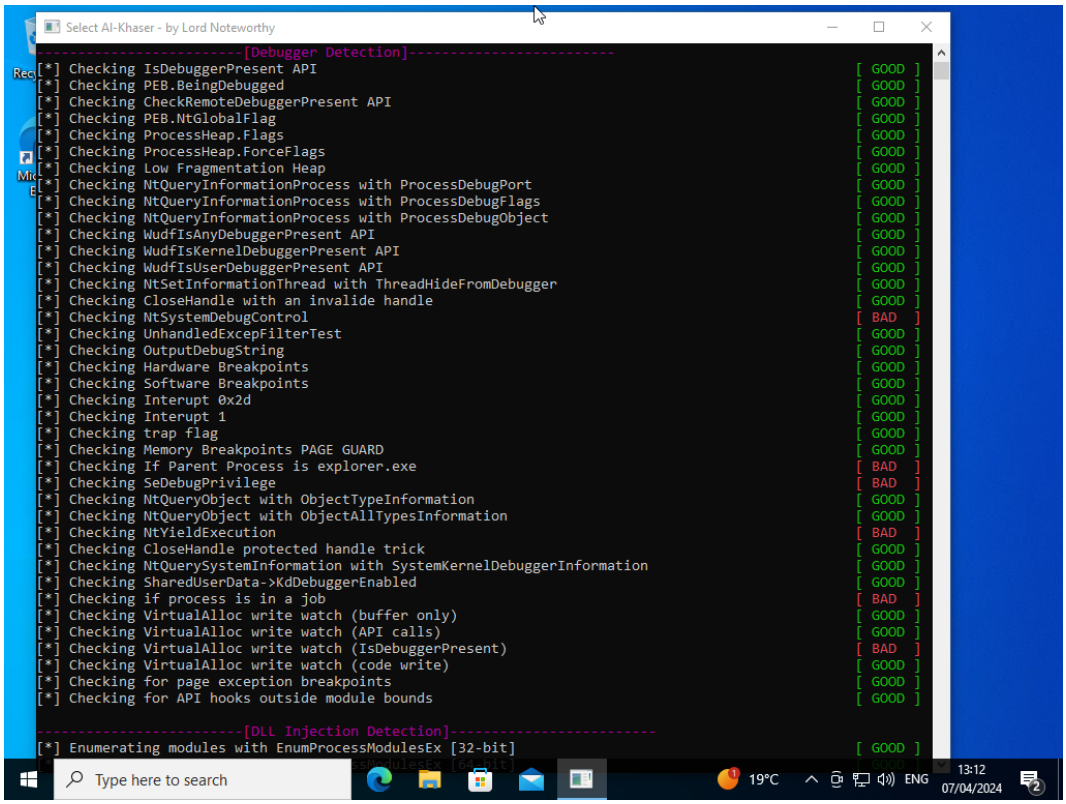


Figure 3.2: Al-Khaser Running Within CAPE

Challenges in Identifying Undetected Techniques What has not been explained yet is how testing these techniques will lead to identifying those that are currently not detected by CAPE. By submitting these malware analysis-environment-detecting tools to CAPE and using either the screenshot module or manually taking screenshots from the guest VM (ensuring not to influence the analysis), it is possible to observe which detections of the analysis environment were successful and which detections were not. Post-analysis, the signatures that were triggered may indicate the presence of an evasive technique. From this, it might be inferred that all undetected techniques can be identified by comparing each detection from the automated tool - regardless of whether the technique succeeded or not - with the missing signatures in CAPE. In this line of reasoning, every missing signature would correspond to an undetected technique. However, this inference is not entirely accurate.

For example, a simple technique to check for the presence of a debugger is the `IsDebuggerPresent()` API function. This function checks whether the current process is being debugged by a user-mode debugger by examining the BeingDebugged flag in the Process Environment Block (PEB). Evasive malware can use this function to determine if a user-mode debugger is attached and may decide not to execute its malicious behavior if the function returns that the process is being debugged. CAPE hooks this function, allowing the manipulation of its output if desired. In other words, if a regular debugger (not CAPE's transparent debugger) were attached

to the process within CAPE, the automated tool would incorrectly report that no debugger is attached, even though one actually is. CAPE also enables dynamic countermeasures against evasion by manipulating the control flow of evasive malware, using a combination of its transparent debugger and Yara signatures. This ability to manipulate output and perform dynamic countermeasures is referred to in CAPE terminology as an *anti-evasion bypass*. In the example given, however, the analysis report would not show any signature, even though the technique was employed by the automated tool. This occurs because, at the time of writing, CAPE lacks a specific signature to check for this technique. The absence of a signature might be due to the technique's infrequent use in practice or because CAPE's special debugger cannot be detected using this technique; however, the exact reason is not documented. This illustrates that while some techniques are clearly recognized as evasion techniques by CAPE - being hooked and thus recorded in the analysis report - they aren't necessarily "detected" to the sandbox user, who does not see any corresponding signature. Therefore, it is necessary to define what "undetected" means in the context of this work.

Undetected Technique Definition By "undetected techniques", techniques are meant that can detect CAPE's presence and, given the current state of CAPE, cannot have signatures written for them. This definition excludes the previous example for two reasons. First, the `IsDebuggerPresent()` API function is unable to detect CAPE's debugger, and therefore cannot detect CAPE's presence. Second, even if the technique could detect CAPE, a signature could be created for it. CAPE hooks this function and logs its calls, so a signature could be developed despite its current absence. To identify truly "undetected techniques", as defined, a new approach is necessary.

Proposed Approach The approach presented is based on the following principle: if an evasion technique in the automated tool successfully detects CAPE, it indicates that CAPE lacks an *anti-evasion bypass* for this technique. Therefore, the technique is deemed undetected (or the technique is incorrectly indicated as successful). It should be noted that the enumeration of techniques is not exhaustive but limited to those implemented in the automated tool. The Venn diagram below illustrates the relationships among the techniques used by these tools, the evasion techniques that can detect CAPE, anti-sandbox and the complete set of all evasion techniques.

Discussion The Venn diagram demonstrates how the search space has been narrowed, assuming no additional implementations are included by the user. It shows that evasion techniques capable of detecting CAPE form a subset of all evasion techniques. Some of these techniques apply to CAPE's analysis environment, while others, such as anti-AV and anti-instrumentation methods, do not. This approach may leave some undetected techniques undiscovered, as it does not cover all possible techniques. Automated tools typically focus on anti-sandbox (or anti-VM) techniques, which makes them the predominant type of technique discovered using this

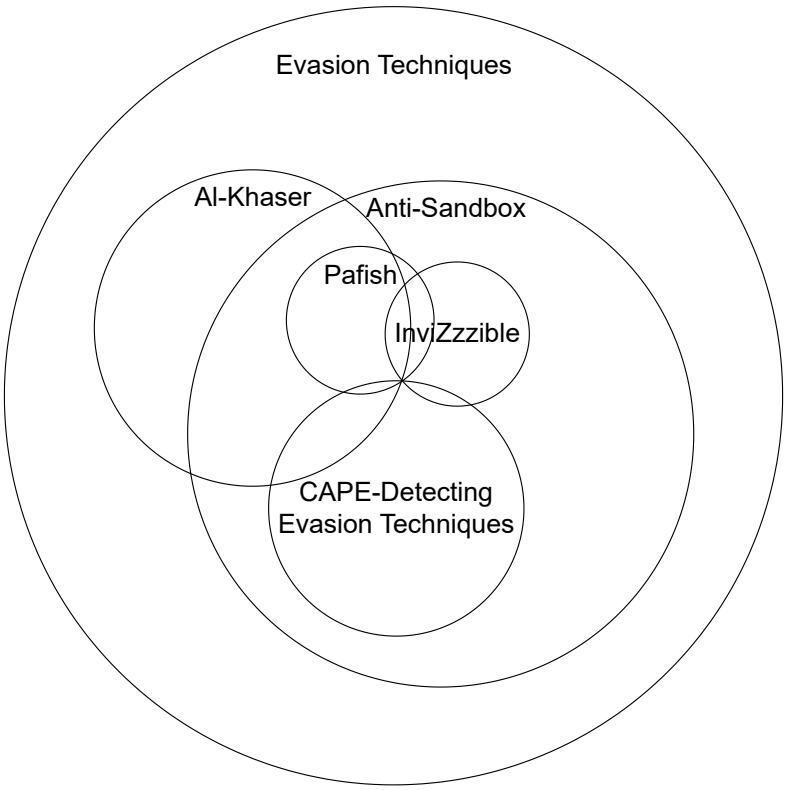


Figure 3.3: Evasion Technique Relations

method. Techniques that do not specifically target CAPE will not be uncovered by this method and are out of scope of what is defined as an undetected technique.

Note that the search space can be expanded by incorporating additional techniques into the automated tools. Popular tools like Pafish and al-khaser, receive periodic updates, gradually extending this search space. Moreover, since all the tools mentioned are open source, users have the option to implement their own additions.

Results See table 3.1.

3.2.2 Using custom binaries

Custom Binaries Instead of relying on open source automated tools, custom programs can be developed to test evasion techniques. These custom binaries should function similarly to automated tools, but offer greater control over the implementation process. Without the need to adhere to an existing framework to manage tasks like retrieving function addresses, the development-to-testing timeframe can be significantly shortened, and finer control over the testing process is achieved. This allows for the isolation of specific techniques, resulting in less crowded and

Pafish	**Al-Khaser**	**InviZzzible**
CPUID hypervisor vendor	ParentProcess	TimeTampering
	SeDebugPrivilege	Unbalanced Stack
	NtYieldExecution	TickCount
	JobProcess	AgentArtifacts
	WriteWatch IsDebuggerPresent	
	MemoryWalk GetModuleInformation	
	MemoryWalk Hidden Modules	
	CPUID hypervisor field CPUID hypervisor vendor	
	No. of Cores WMI Current Temperature WMI ProcessId WMI Power Capabilities CPU Fan WMI Win32_CacheMemory WMI Win32_PhysicalMemory WMI Win32_MemoryDevice WMI Win32_MemoryArray WMI Win32_VoltageProbe WMI Win32_PortConnector WMI Win32_SMBIOSMemory WMI ThermalZoneInfo WMI CIM_Memory WMI CIM_Sensor WMI CIM_NumericSensor WMI CIM_TemperatureSensor WMI CIM_VoltageSensor WMI CIM_PhysicalConnector WMI CIM_Slot WMI	
	PiratedWindows	
	ACPI Tables	
	HyperV Global Objects	

Table 3.1: Identified Undetected Techniques from Automated Tools

more interpretable analysis results - an important factor when developing detections for specific techniques. The approach remains largely the same as when analyzing automated tools.

Discussion A simple console application that implements an evasion technique and outputs the result - such as 0 for failure or 1 for success, is sufficient.

Figure 3.4 illustrates an example of such a console application. This particular application also outputs additional debugging information to assist the user in

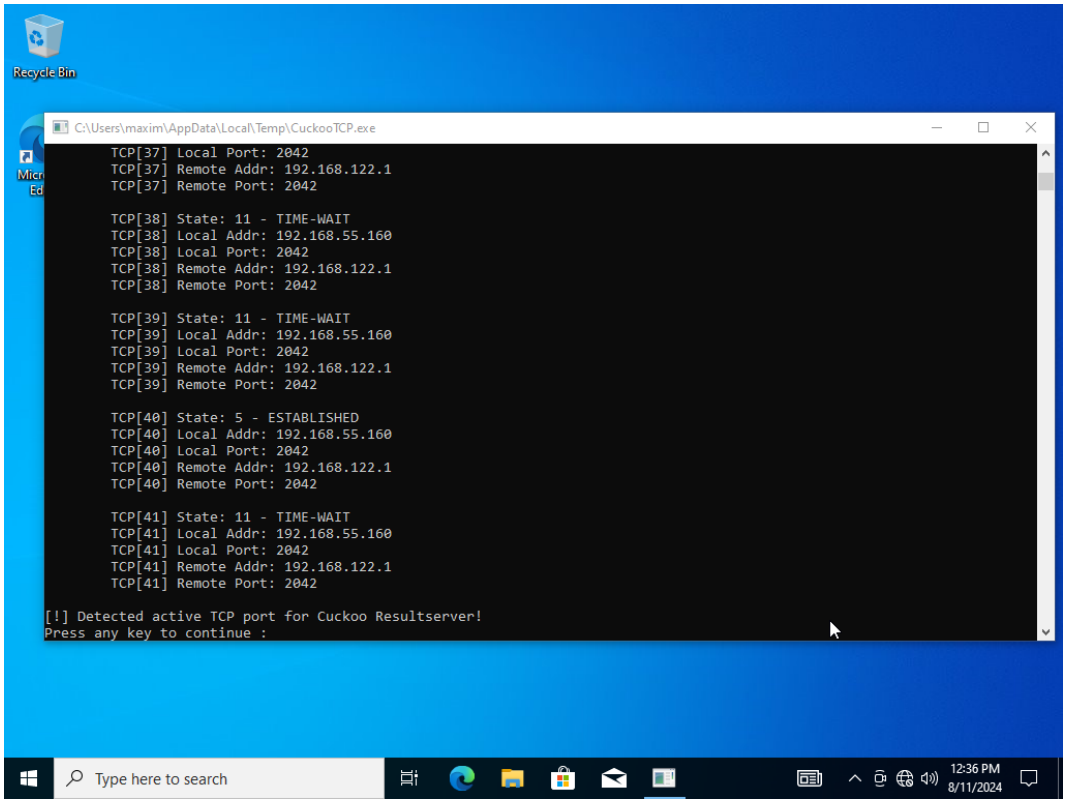


Figure 3.4: Custom Binary for CuckooTCP Technique

validating the correctness of the technique.

By using the screenshot module or manually inspecting the guest VM and taking screenshots, the results can be analyzed in the same way as before. A comparison between the screenshots and CAPE's returned signatures can reveal whether a technique is undetected by CAPE, based on the success of the technique and the absence of corresponding signatures. Unlike using automated tools, which require minimal knowledge of evasion techniques, writing custom binaries necessitates some understanding of the techniques being tested. However, the implementation process is simplified by resources like the Unprotect Project [44] and Check Point Research Evasion Techniques [8], which provide knowledge bases and sample code for various evasion techniques, making it easier for authors to develop these binaries.

The search space of this approach becomes limited to the specific techniques implemented by the author, rather than the broader range of techniques included in automated tools. Automated tools often come with a substantial number of pre-implemented techniques, making it a significant challenge to surpass their effectiveness through manual binary implementation alone. However, there is merit in combining both approaches. By leveraging the existing techniques tested by automated tools, small binaries can be created to quickly prototype different evasion techniques, which can then be integrated into the automated tool.

When adding new techniques, selecting which ones to include can be challenging

due to the vast number of possibilities. One effective approach is to focus on the specific properties of the analysis tool in use. For instance, CAPE, being a sandbox, would benefit from the analysis of anti-sandbox techniques. Since the default recommended setup for CAPE involves using VMs for guest systems, anti-VM techniques would also be relevant. Additionally, considering CAPE's use of a specialized debugger, anti-debugging techniques could be of interest as well.

Results See table 3.2.

Check Point	**Unprotect Project**	**BlackHat**	**Apriorit**
SreenResolution	ParentProcess (2) [17]	ParentProcess (2) ParentProcess (3) ParentProcess (4) [26]	ParentProcess (5) [24]
Uptime			
HDDName			
HookDetection			
GetLastInputInfo			
CuckooTCP			
SleepSkipping Detection			

TABLE 3.2: Identified Undetected Techniques from Custom Binaries

3.2.3 Using real malware

Real Malware Another approach is to examine real malware samples. If malware samples capable of evading CAPE are identified, analyzing these samples could reveal the techniques they employ, including those that CAPE cannot detect. This approach is not limited to techniques that detect the analysis environment, allowing for the discovery of detection-independent techniques as well.

Challenges of the Proposed Approach Identifying the undetected techniques using this approach presents several challenges. Firstly, there is a significant lack of publicly available evasive malware datasets, as mentioned throughout this thesis. Assembling a sufficiently large set of evasive malware to ensure a high probability of finding CAPE-evading samples is difficult, and obtaining a diverse yield of different techniques is even more challenging. Secondly, uncovering the specific techniques used by a sample is not straightforward, largely due to the scarcity of labeled datasets that document the evasion techniques for each sample - an issue this thesis seeks to address. In the absence of such datasets, manual analysis is likely necessary, requiring advanced knowledge of reverse engineering and malware analysis - skills that are beyond the scope of this work. Additionally, analyzing real malware requires caution, as there is a risk that the malware could exploit vulnerabilities and escape the sandbox. This is unlike automated tools, where it is assured that no malicious behavior occurs. Finally, once a CAPE-evading malware sample is found, careful

examination of the functions hooked by CAPE is required to determine whether a signature can be written for the technique. This step is essential to confirm whether the technique qualifies as undetected according to the proposed definition.

Alternative Approach A sample-searching strategy like the one above presents practical challenges. An alternative approach is to reverse the process: start with specific evasion techniques and then search for malware samples that utilize them. This method has its own difficulties. First, identifying which techniques are likely to go undetected by CAPE. This is similar to the challenge of selecting techniques for custom binaries and can be tackled the same way - by understanding the analysis tool's mechanisms and assessing the effectiveness of different categories. Second, finding a sample that employs the selected technique can be challenging, especially given the scarcity of datasets labeled with evasion techniques. This task often proves difficult in practice, requiring reliance on previous research. Examining malware families in previous studies might also help in identifying samples and their associated techniques.

Malware Families Numerous articles and blog posts discuss the analysis of malware samples, often focusing on variants of popular malware families rather than obscure or random samples. These analyses typically provide detailed insights into the new and innovative techniques used by the malware families, as demonstrated in sources like [33]. Such articles can serve as valuable starting points for locating samples that incorporate the evasion techniques of interest. Additionally, as the threat landscape evolves, focusing on more recent samples can further narrow the search space, as newer malware are likely to employ more relevant and effective techniques. For instance, [34] illustrates that malware authors tend to discard techniques that have become ineffective over time. However, it is important to recognize that techniques not widely adopted by commonly studied malware families may be significantly underrepresented, making them more challenging to identify and analyze.

Discussion While exploring undetected techniques through the use of real malware samples could potentially yield valuable insights, this approach was not pursued for several compelling reasons. The primary challenge lies in identifying malware samples that can effectively evade detection by CAPE. Given CAPE's ability to detect and bypass a wide range of evasion techniques employed by various malware families, finding samples that specifically evade its advanced detection mechanisms is particularly difficult and resource-intensive. Furthermore, when using real malware, the techniques employed are often complex and intertwined, making it difficult to isolate specific evasion methods for study. This complexity can obscure the results and limit the effectiveness of the analysis. Moreover, handling real malware introduces significant risks, such as the potential for system compromise if the malware escapes the sandbox by exploiting vulnerabilities. These risks, combined with the complexities and challenges outlined, led to the decision to exclude the "real malware"

approach from this research.

The next steps in the methodology involve analyzing the source code of the identified techniques and compiling unique identifications for them. The source code analysis is crucial for gaining a deep understanding of how these evasion techniques function. This includes examining why specific functions are invoked, the significance of the arguments they are invoked with, and how the results of these functions contribute to detecting or evading the analysis environment. This understanding helps in identifying the core components essential for the technique to work, allowing for the creation of a unique identification for each technique. For instance, techniques often involve distinct patterns of API calls or the use of specific arguments to extract information from the analysis system. Once these unique identifications are compiled, they can be translated into CAPE signatures, enabling practical detection of the techniques. The process of translating these identifications into CAPE signatures is discussed in the next section.

3.3 Creating Detections for Undetected Techniques

Once techniques have been identified that CAPE cannot detect, the next step is to create detections for them. However, it is important to remember that, according to the earlier definition of undetected techniques, it was not possible to create signatures in CAPE for these techniques. This limitation exists because, in CAPE's current state, insufficient information is gathered to accurately and precisely describe these techniques.

Considerations for Rule-Based Systems In a rule-based system like CAPE's signatures, a certain level of precision is crucial. If a rule is too generic, it becomes overly broad and fails to classify the technique accurately, leading to numerous false positives. Conversely, if a rule is too narrow, its precision can hinder its generalization, making it applicable only to a specific implementation of the technique. This is problematic if slight variations of the technique are used in practice. Not all techniques can be identified using a single function, its arguments, or its return value. Often, a distinct pattern of functions is needed to capture the essence of the technique. Due to the challenges of generalization in rule-based systems, capturing this essence is difficult. Many different combinations of functions might perform the same general idea, making it challenging to write rules that cover all possible variations.

The primary reason signatures cannot be created for these specific techniques in CAPE is the absence of essential information in the analysis report. Specifically, one or more functions used by the technique, which are crucial for its unique identification, are not logged. In CAPE, API calls are logged through hooking, as explained in the background section. The following sections will explain how new functions can be hooked and how to write signatures in CAPE.

Step 1: Hook relevant function(s) in capemon

In CAPE [50], hooks are managed through the *capemon* DLL, which is injected into the analyzed process. Guidance on adding new hooks is provided in Capemon's github repository [9]: "There are three main files that define the hooks implemented in capemon:

1. hooks.h. This file contains the definition of the hook (`HOOKDEF`) using Windows SAL notation. That is, `HOOKDEF(ReturnValue, CallingConvention, ApiName, _ParameterAnnotation_ ParameterName)`.

2. hooks.c. This file defines the hooks that will be employed depending upon the configuration selected when submitting the analysis. Please notice there are several `hook_t` arrays. For example, `hook_t full_hooks[]`, `hook_t min_hooks[]` or `hook_t office_hooks[]`, among others. You should add the hooks you want capemon to perform in the corresponding array. By default, `full_hooks` is executed (so probably you want to add your hooks there). The hooks must be added using the following naming pattern: `HOOK(dllname, ApiName)`.

3. hook_category.c *(Link is just an example, in this case hook_process.c)*. This set of files is where the implementation of each hook is defined. When defining the behavior of a given hook, you must copy the corresponding definition from the hooks.h file and write the code. Remember you can call the original function with `Old_{ApiName}`."

The following describes the process for a trivial undetected technique called "Pirated Windows", which was discovered. The source code for this technique, implemented by al-khaser, is provided in Appendix A. This technique uses the `SLIsGenuineLocal` API function to discern whether the Windows installation is genuine based on the local. In this method, `SLIsGenuineLocal` is the primary and sole API function employed.

Defining a Hook First, the *hooks.c* file is checked to ensure that no hook for `SLIsGenuineLocal` is currently implemented. Upon reviewing `hook_t full_hooks[]`, it is confirmed that no hook definition of the form `HOOK(..., SLIsGenuineLocal)` is present. To add the hook (as described above), the DLL containing the API function must be identified. According to the documentation for the SLIsGenuineLocal function [39], the required DLL is *slwga.dll*. Therefore, the following line should be added to `hook_t full_hooks[]`: `HOOK(slwga, SLIsGenuineLocal)`.

Second, with the hook now set to be implemented, a definition must be provided in the *hooks.h* file. Since no definition for `SLIsGenuineLocal` exists yet, one must be added. The definition can be derived from the *Syntax* section of the `SLIsGenuineLocal` documentation. The syntax can be copied and then adapted for *hooks.h* using the following steps:

1. Replace the opening parenthesis with a comma, and place two commas between the return type and the function name. Between these commas, insert the appropriate calling convention (for Windows API functions, this is `WINAPI`).
2. Prepend the definition with `HOOKDEF(`. At this point, the structure of the definition should be correctly formatted.
3. Remove the square brackets and replace the contents using the following scheme:

`in`	→	`__in`
`out`	→	`__out`
`in, out`	→	`__inout`
`in, optional`	→	`__in_opt`
`out, optional`	→	`__out_opt`
`in, out, optional`	→	`__inout_opt`

Following these steps, the resulting definition for `SLIsGenuineLocal` is as follows:

```
HOOKDEF(HRESULT, WINAPI, SLIsGenuineLocal,
  __in                const SLID                *pAppId,
  __out               SL_GENUINE_STATE          *pGenuineState,
  __inout_opt         SL_NONGENUINE_UI_OPTIONS *pUIOptions
);
```

Hook Implementation Next, an implementation of the hooked function must be provided in the appropriate category file, following the naming scheme *hook_category.c*. The existing categories are *clr* (compiler), *crypto*, *file*, *misc* (miscellaneous), *network*, *process*, *reg* (registry), *reg_native*, *services*, *sleep*, *socket*, *special*, *sync* (synchronization), *thread*, *tls* (Transport Layer Security, also known as TLS), and *window*. For `SLIsGenuineLocal`, the most fitting category is *hook_misc.c*. Below is the implementation:

```
HOOKDEF(HRESULT, WINAPI, SLIsGenuineLocal,
  __in                const SLID                *pAppId,
  __out               SL_GENUINE_STATE          *pGenuineState,
  __inout_opt         SL_NONGENUINE_UI_OPTIONS *pUIOptions
) {
  HRESULT ret = Old_SLIsGenuineLocal(pAppId, pGenuineState, pUIOptions);
  LOQ_hresult("misc", "i", "GenuineState", *pGenuineState);
  return ret;
}
```

CAPE's hooking engine allows the original function to be called by prefixing the function name with `Old_`. Since bypassing the evasion technique is outside the

scope of this thesis, the result of the original API function is returned by calling `Old_SLIsGenuineLocal` with its corresponding parameters. If bypassing the evasion were desired, it could be easily achieved by setting the `SL_GEN_STATE_IS_GENUINE` field of `SL_GENUINE_STATE` to 0 before returning `ret`[1]. The function call is logged in the analysis report using one of the various LOQ functions, which differ depending on the function's return type: *LOQ_bool*, *LOQ_handle* (for file handles, etc.), *LOQ_hresult*, *LOQ_msgwait*, *LOQ_nonnegone* (non-negative one), *LOQ_nonnull*, *LOQ_nonzero*, *LOQ_ntstatus*, *LOQ_sock* (sockets), *LOQ_sockerr* (socket errors), *LOQ_void*, and *LOQ_zero*. Since `SLIsGenuineLocal`'s return type is `HRESULT`, the appropriate LOQ function to use is the *LOQ_hresult*. The parameters for the LOQ functions follow a consistent structure.

The first parameter of the LOQ functions is a string indicating the category, which in this case is `"misc"`, as the hook falls under the miscellaneous category. The second parameter is a string that specifies the format of the subsequent arguments. Each character in this format string corresponds to the type of the following two or more sequential arguments, which are then transformed into a supported format for inclusion in the analysis report. It is crucial to use the correct format options, as incorrect formats can lead to undefined behavior and potentially crash the analyzed program. A list of the available format options can be found in Appendix B. After the format string, the remaining parameters consist of tuples in the form (name, value) or (name, value, length/size) that are to be included in the analysis report.

In the case of `SLIsGenuineLocal`, the key interest lies in the value of the `SL_GENUINE_STATE` enum, as indicated by the source code. This value is logged using the parameters `"i"` (the format for an integer, representing the integer value of the enum), `"GenuineState"` (the name part of the (name, value) tuple in the report), and `*pGenuineState` (the dereferenced integer value of the pointer to the `SL_GENUINE_STATE` enum, which serves as the value part of the tuple).

Step 2: Write the signature

Structure of a Signature Writing signatures is a straightforward process that requires only a basic understanding of Python programming. The first step involves creating a new Python file. Next, you need to import the `Signature` class from `lib.cuckoo.common.abstracts`. With the necessary dependencies in place, the third step is to create a Python class that inherits from the imported `Signature` class and to define some essential attributes for it. Below is a list of some key attributes relevant to signature creation [3]:

- *name*: A unique identifier for the signature.
- *description*: A brief explanation of what the signature detects or represents.
- *severity*: A value between 1 and 3 indicating the severity level of the matched concept.

[1] See https://learn.microsoft.com/en-us/windows/win32/api/slpublic/ne-slpublic-sl_genuine_state [38].

- *confidence*: A percentage between 0 and 100 that reflects the confidence level in the signature's accuracy.
- *categories*: A list of relevant categories for the matched concept (e.g., anti-debugging, anti-VM).
- *authors*: A list of the signature's authors.
- *minimum*: The minimum version of CAPE required to run the signature.

In the fourth and final step, the Python class must implement a `run` method that defines the events the signature is designed to detect. Events can be matched by directly accessing the *global container* (See the section on CAPE), which is similar to parsing a JSON structure, or by using the helper functions provided in lib.cuckoo.common.abstracts [10]. The global container's structure can be examined in an analysis report (formatted in JSON). A signature is considered to match successfully if the `run` method returns `True`. While other methods, including those for more efficient *evented signatures*, are available, they will be left for future work.

Run method implementation Ultimately, the signature for the *Pirated Windows* technique only needs to verify if the `SLIsGenuineLocal` function is invoked by the analyzed program. The global container can be accessed using `self.results`, where the collection of called APIs for each spawned process is located under *behavior*/*processes*/[index of spawned process]/*calls*. The API calls must be checked across all spawned processes. The resulting signature is as follows:

```
from lib.cuckoo.common.abstracts import Signature

class AntiSandbox_PiratedWindows(Signature):
    name = "PiratedWindows"
    description = "Check if the local is a genuine Windows local."
    severity = 3
    categories = ["anti-sandbox"]
    authors = ["Maxim Reynvoet"]
    minimum = "0.5"

    def run(self):
        for process in self.results["behavior"]["processes"]:
            for call in process["calls"]:
                if call["api"] == "SLIsGenuineLocal":
                    return True
        return False
```

The signatures for the remaining uncovered undetected techniques are provided in a github repository. The effectiveness of these detections is evaluated in Chapter 4.

With the signatures now implemented to detect previously undetected techniques, the focus shifts to a broader objective: constructing a comprehensive database that catalogs these evasive techniques and the malware employing them. The initial step in creating this database involves mapping the developed and existing signatures to a taxonomy of evasion techniques, thereby facilitating classification.

3.4 Creating and Mapping to a Taxonomy

This section provides an up-to-date taxonomy of evasion techniques based on existing research in malware analysis. Signatures are mapped to this taxonomy, by assigning each signature to a specific technique. The technique must be classified under the appropriate tactic, which, in turn, falls under the relevant evasion type. The evasion type is determined by the analysis environment targeted by the technique, while the tactic is based on the resources used by the technique or its intent or result. It is important to note that this mapping can be subjective, depending on how clearly the descriptions of each technique are defined.

3.4.1 Preliminary Taxonomy Definitions

To provide a clearer understanding of the taxonomy of evasion techniques, several commonly used terms that will be referenced throughout this section will be defined:

- **Evasion type**: The overarching goal of evasion and the general attitude towards achieving it.
- **Tactic**: Specific maneuvers or approaches for evasion, aligned with the objective of its parent evasion type.
- **Technique**: Practical methods for implementing tactics.

3.4.2 Proposed Classification

The proposed classification is illustrated in Figure 3.5.

Anti-Sandbox / Anti-VM Classification

A classification of techniques aimed at evading virtual machines, that are often used as contained and safe environments for malware analysis, and automated malware analysis sandboxes. For instance, the malware can try to detect the environment to verify whether the host is a sandbox (or VM) or not. Alternatively, the evasive malware does not rely on detecting the presence of a specific sandbox in the target environment and instead uses techniques to hinder the analysis process in the sandbox, e.g. through sleeping until the critical timeout.

- **Fingerprinting**: A tactic pursued by malware to detect the presence of sandboxes by looking for environmental artefacts or signs that could reveal the indications of a virtual/emulated machine e.g. device drivers, overt files on disk

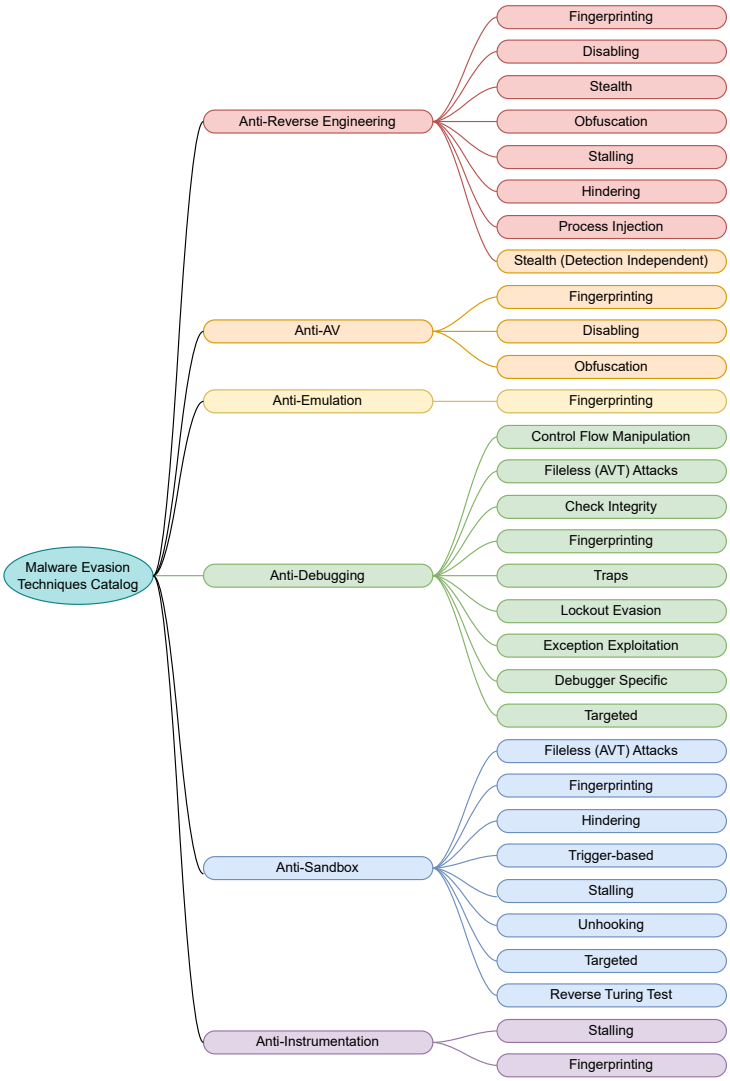


FIGURE 3.5: Proposed Evasion Technique Taxonomy

and registry keys, discrepancies emulated/virtualized processors. Sandboxes leave their marks on different levels:

– **Hardware**: Look for devices and drivers, e.g. *VMWare Ethernet* device to more subtle marks. Specific drivers are also employed by VMs to interact properly with the host OS, for instance, in the path `C:\Windows\System32\Drivers` exist such signs that could expose VMWare, VirtualBox, etc.

– **Execution Environment**: Malware inside a sandbox experiences subtle

differences in the environment within which they are executed, e.g. kernel space memory values are different between a sandbox and native system. Artifacts manifest themselves in memory or execution, for instance to allow inspection and control between host and guest OS.

  - **Application**: Artifacts of installation and execution of an analysis application. Even if not executed, evidence might be residing on disk, registry keys, etc. These can be readily found by the malware, especially if they contain a well-known filename or locations (e.g. `SYSTEM\CurrentControlSet\Control\VirtualDeviceDrivers`) or if the names are not altered or the corresponding processes not concealed, they can be enumerated (e.g. *VMtools.exe, Vmwareuser.exe, vboxservice.exe*).
  - **Behavior**: Leakages caused by imperfect virtualization/emulation, or innate characteristics, e.g. discrepancies between an emulated CPU and a physical one (e.g. performance or relative performance[2], effects of caching[3], red pill tests[4]).
  - **Network**: E.g. fixed IP addresses, emulated or prevented internet access or an extremely fast internet connection.
  - **ParentCheck**: parent process ID is retrievable, could be the name of a debugger or something else that is not equivalent to the process name *explorer.exe*. Using *CreateToolhelp32Snapshot()* for instance one could check if the parent process name is the name of an in-guest analysis component.

- **Reverse Turing Test**: checking for human interaction with the system[5], e.g. await mouse left-click to detect human interaction or look for last user's inputs using *GetTickCount()* and *GetLastInputInfo()* functions to compute the idle time of the user, scrolling to the second page of a rich text format or getting the cursor position and measuring its speed or seeking wear and tear signs of a production system, etc.

  - **I/O**: Tries to detect human interaction with input/output devices.
  - **Wear and Tear**: Tries to detect signs that attest to the age of the system and its frequent use.

- **Targeted**: Malware fingerprints to verify if the host is precisely the intended (targeted) machine.

[2] Compare the performance ratio of two or more operations on the same system. If it varies significantly among emulation systems the chance for sandbox presence is high.

[3] A function is executed a number of times, then the test is performed again in the absence of caching. This shows the effectiveness of caching. Simulation of a processor cache is a complex task and emulators may not support it.

[4] Testing for imperfect simulation of the CPU and residing instruction bugs (e.g. finding such bugs and noticing mishandled instructions during execution means the presence of a sandbox).

[5] Simulated human behavior can also be detected.

  - **Environmentally Targeted**: e.g. looking for the presence of a specific industrial control system.
  - **Individually Targeted**: Attackers make sure that their malicious code is directly delivered to their target.
  - **Environment-Dependent Encryption**: The malware has an encrypted payload of which the decryption key is derived from the victim's environment.

- **Stalling**: Malware postpones its malicious activity to the post-analysis stage.
  - **Simple Sleep**: Sleep for n minutes, as to timeout the sandbox. Counter: sleep patching = accelerating time.
  - **Advanced Sleep**: Detect sleep-patching using the *rdtsc* instruction in combination with *Sleep()* to check the acceleration of execution.
  - **Code Stalling**: The malware executes irrelevant and time-consuming benign instructions to avoid raising any suspicions. Two hurdles (*Rombertik*):
    1. Inability to suspect the suspect stalling as the sample is running actively.
    2. Excessive writing would overwhelm tracking tools

    Or the malware encrypts the payload with a weak encryption key and brute-forces it during the execution.
  - **Onset Delay** Delay execution through means other than sleeping or code stalling, e.g., performing a ping during a defined time.

- **Hindering**: Malware tries to hinder the analysis, for instance by trying to stop it.
  - **Reboot/Shutdown/Sleep**: Attempts to shutdown or reboot the system or guest VM.

- **Unhooking**: Verifies the integrity of the APIs it is trying to call before calling them. If a hook is noticed, the malware will try to remove the hook before calling the API.

- **Trigger-Based** (Trigger tactic = Logic Bomb): Wait for a trigger to activate, i.e. specific system date or specially crafted network instruction. A number of triggers:
  - **Keystroke-Based**: The malware gets triggered if it notices a specific keyword, i.e. the name of an app or title of a window.
  - **System Time**: System date or time serves as trigger.
  - **Network Inputs**: Triggered when they receive certain inputs from the network.

  - **Covert Trigger-Based**: Use instruction-level stenography to hide malicious code from the assembler and implement trigger-based bugs to provision stealthy control transfer, making dynamic analysis to discover proper triggers difficult.

- **Fileless Malware**: (= Non-malware = Advanced Volatile Threats [AVTs]): Requires no file to operate, resides purely in memory and takes advantage of existing system tools such as *PowerShell*, making forensics much harder.

Anti-Reverse Engineering Classification

Included in this classification are evasion techniques aimed at evading generic analysis tools that are used by security researchers, such as network analysis, reverse engineering tools, etc. as to reverse-engineer the inner workings of the malware. It also includes evasion techniques against the very process of analysis undertaken by researchers to analyse samples using these tools. For instance, malware can try to appear as a legitimate file or process as to not raise suspicion, thus may not be noticed and analyzed by the researchers.

- **Fingerprinting**: Analogous to Anti-Sandbox:Fingerprinting, but for the presence of analysis tools e.g. WinDump, Wireshark, etc.

- **Stealth**: Avoids leaving traces through special means or attempts to clear traces left behind by its execution.

  - **Clear Artifacts**: Clears events or logs, for instance the event logs stored in `C:\Windows\System32\winevt\Logs`, browser history or deletes files that were executed.

  - **Covert Channel**: Avoids traces by using a covert communication channel for its communication, e.g. manipulating data from or to the recycle bin or using Tor.

  - **Masquerading**: A process or file pretends to be another process or file, attempts to use the system's user agent for requests, etc.

- **Obfuscation**: The malware hides malicious and/or known malware family patterns in its code as to mask its malevolent behavior, e.g. through encrypting its payload, changing (decryptor) code slightly, etc.

  - **Encoding**: Applies a bidirectional transformation to the code or data that changes it into a well-known (e.g. Base64, hexadecimal, etc.) or custom format that does not change its meaning.

  - **Encryption**: The malware's payload is transformed into an alternative form that hides the original meaning. It can only be reverted to its original form using a key that must be kept secret. Commonly used encryption

schemes are XOR encryption, Advanced Encryption Standard (AES) and custom encryption schemes.

- **Compile-After-Delivery**: Deliver the payload as uncompiled code and compile it before execution.
- **Packing**: Makes use of a packer to hide its malicious behavior.
- **Oligomorphism**: Oligomorphic malware changes its decryptor slightly between generations.
- **Polymorphism**: Polymorhpic malware use mutation engines to alter code or appearance between generations. Mutation engines often utilize new key generation algorithms between generations to enact encryption and decryption for their variants.
- **Metamorphism**: Metamorphic malware rewrites its entire codebase to a functionally equivalent one between generations.

- **Hindering**: See Anti-Sandbox:Hindering
- **Disabling**: Tries to disable the antivirus, either by stopping or modifying it.
  - **Set Registry Keys**: See Anti-AV:Disabling:Set Registry Key
  - **Set Software Restriction Policy**: See Anti-AV:Disabling:Set Restriction Policy
  - **Command Line Tools**: See Anti-AV:Disabling:Command Line Tools
  - **Disable Warnings**: Disables warnings for the antivirus software, e.g., Windows Security Center warnings.
  - **Disable Windows Firewall**: The malware disabled Windows firewall.
- **Stalling**: See Anti-Sandbox:Stalling
- **Process Injection**: Arbitrary code is injected into another process, which allows running this code from the memory space of the other process.
  - **DLL Injection**: A malicious Dynamic Link Library (DLL) is injected into the target process. The process will execute the DLL's code when it is loaded, thus making the legitimate process execute the malicious behavior.
  - **Process Hollowing**: The memory is hollowed out of a legitimate process that was started in a suspended state and malicious code is injected, after which the process is resumed, thus executing the injected code.
  - **Image File Execution**: The malware sets the execution options of an image using File Execution Options (IFEO).
  - **Silent Process Exit**: The malware uses the Monitoring Silent Process Exit mechanism, allowing to set the execution of an application or script, when a process terminates after result of `ExitProcess` call or `TerminateProcess` called by another process.

  - **Extra Window Memory Injection**: The malware may inject malicious code into a process via the Extra Window Memory (EWM), avoiding detection as well as possibly elevating privileges.

Anti-AV Classification

This classification focuses on techniques aimed at evading antivirus software and other similar security technologies, such as Endpoint Detection and Response (EDR) technologies. Evasive malware can probe the environment for traces of the presence of an AV or EDR technology and not execute its malicious behavior. Alternatively, the malware tries to hide from and make analysis harder for any antivirus software present. For instance, it can alter its malicious code in such a way that the antivirus software does not recognize it as malicious.

- **Fingerprinting**: Spot, find or detect signs that attest to the presence of an antivirus e.g. Avast Antivirus, Bitdefender Antivirus, etc.
  - **Application**: Artifacts of installation and execution of an analysis application. Even if not executed, evidence might be residing on disk, registry keys, etc. These can be readily found by the malware, especially if they contain a well-known filename or locations or if the names are not altered or the corresponding processes not concealed, they can be enumerated.
  - **Library**: Artifacts of installation and execution of an antivirus application. Even if not executed, evidence might be residing on disk, registry keys, etc. These can be readily found by the malware, either through libraries specific to an antivirus (e.g. the "avcuf32" DLL in the case of Bitdefender Antivirus), or especially if they contain a well-known filename, registry keys or locations (e.g. the installation directory of `C:\Program Files\Bitdefender` or the `Software\(Wow6432Node\)Bitdefender` registry key) or if the names are not altered or the corresponding processes not concealed, they can be enumerated (e.g. *bdagent.exe* is one of the processes part of the BitDefender Security Suite).
- **Disabling**: Tries to disable the antivirus, either by stopping or modifying it.
  - **Stop Service**: Attempts to stop the active service e.g. through calling the ControlService API function with ControlCode `SERVICE_CONTROL_STOP`.
  - **Set Registry Keys**: Attempts to modify permissions for the antivirus in such a way as to hinder it from performing (part of) its duties through modification of registry keys.
  - **Set Software Restriction Policy**: Analogous to stop service, but by setting the Software Restriction Policies via Powershell.
  - **Command Line Tools**: Analogous to stop service, but using command line tools, such as DISM.
- **Obfuscation**: See Anti-Reverse Engineering: Obfuscation.

- **Process Injection**: Arbitrary code is injected into another process, which allows running this code from the memory space of the other process.
  - **DLL Injection**: See Anti-Reverse Engineering:DLL Injection.
  - **Process Hollowing**: See Anti-Reverse Engineering:Process Hollowing.
  - **Reflective DLL Injection**: DLL Injection is performed, but the DLL is loaded and executed directly from memory (nothing is written to disk).
- **Fileless Malware** See Anti-Sandbox:Fileless Malware

Anti-Debugging Classification

Anti-debugging is the application of one or more techniques to fend off, impede, or evade the process of (manual or) dynamic analysis using debuggers. For instance, the `IsDebuggerPresent` function allows an application to determine whether or not it is being debugged.

- **Fingerprinting**: Spot, find or detect signs that attest to the presence of an analysis environment or debuggers.
  - **Analyzing Process Environment Block (PEB)**: PEB is a data structure that exists per process in the system and contains data about that process, e.g. *BeingDebugged* field which can be read directly or through specific APIs i.e. *isDebuggerPresent()*, *CheckRemoteDebuggerPresent()*. Counter: alteration of *BeingDebugged* bit or API hook.
  - **Search for Breakpoints**: With hardware breakpoints the breakpoint address can be stored in CPU DR registers, with software breakpoints the debugger writes the 0xCC opcode into the process (specifically designated for setting breakpoints). The malware does a self-scan or integrity check by looking for 0xCC or using *GetThreadContext* to check CPU registers. Counter: (software breakpoint) debugger has to keep and a feed copy of the original byte that was replaced by 0xCC.
  - **Probing for System Artefacts**: Debugger leaves behind traces in OS (e.g., file system, registry, process name) which the malware can look for (for instance using APIs as *FindWindow() and* FindProcess(). I.e. give the name of a debugger to the *FindProcess() API*). Counter: randomize names or alter results of the API queries through API hooks.
  - **Parent Check**: parent process ID is retrievable, could be the name of a debugger or something else that is not equivalent to the process name *explorer.exe*. Using *CreateToolhelp32Snapshot()* for instance one could check if the parent process name is the name of a well known debugger. Counter: skipping the relevant APIs.

  - **NtQuerySystemInformation mining**: *ntdll NtQuerySystemInformation()* accepts a parameter, which is the class of information to query for. Counter: patching the kernel.
  - **NtQueryInformationProcess mining**: *ntdll NtQueryInformationProcess()* accepts a parameter, which is the class of information to query for. For instance, using the ProcessDebugPort information class, a process can infer it is currently being attached to a debugger. Counter: patching the kernel.
  - **Check Execution Yielding**: When a debugger is single stepping through a program, the context can't be switched to other threads. The *ntdll NtYieldExecution()* function will return `STATUS_NO_YIELD_PERFORMED`, which leads to the *kernel32* function `SwitchToThread()` returning zero.
  - **Check job object**: Debuggers and analysis applications usually place processes inside a job such that any child processes exit when the parent process exits.
  - **Timing-Based Detection**: setting time thresholds in the presumption that a particular function or instruction set requires only a minuscule amount of time. Use of local APIs such as *GetTickCount(), QueryPerformanceCounter(), etc.*, CPU *rdtsc* (read timestamp counter) or inquiring external sources through the network. Counter: (local) Difficult to prevent; (external) open problem.

- **Exception Exploitation**: Exploits exception or error mechanisms to infer the presence of a debugger.
  - **OutputDebugString**: The malware can set the last error code for the thread to an arbitrary value using *SetLastError()*, then display a message to the potential debugger using *OutputDebugString()*, after which it can check the last error code again using *GetLastError()*. If the value does not match what was set originally, a debugger is present.
  - **UnhandledExceptionFilter**: A custom exception filter is set using `SetUnhandledExceptionFilter`, however, when a process is being debugged the exception will be passed to the debugger instead of the filter.
  - **Custom Exception Handler**: Hide the control flow of the program by registering several exception handlers that pass the exceptions to each other before going to the real procedure.

- **Traps**: Debugger produces specific information or lack thereof when stepping through the malware. Exploits the logic of the system, e.g. *SEH* (Structured Exception Handling): embed *int 2dh* instruction, which raises an exception in the absence of a debugger that the malware can handle in a try-catch structure. With a debugger, the instruction is transferred to the debugger instead of the malware, causing the malware to notice the absence of an expected exception. Other possibilities include embedding specific instruction prefixes or other instructions such as *41h*. Counter: Skip these instructions in the debugger.

- **Check Integrity**: The program checks its own integrity, either by inspecting its own memory, modules or by adding a checksum.
  - **Reads Self**: Reads data out of its own binary image, presumably to check its integrity.
- **Debugger Specififc**: Exploits vulnerabilities specific to a certain debugger. E.g. OllyDBG had a format string bug which can be exploited to cause it to crash.
- **Targeted**: Malware encrypts malicious payload with an encryption key that is an attribute or variable that can only be found on the target system, e.g. a serial number of a component in the target system, specific environment settings, etc.
  - **Environment Keying**: key is derived from the target environment. Counter: brute-force the payload to find the encryption key (not always feasible) or path exploration techniques (not effective against AI powered techniques).
  - **AI-Powered Keying**: uses the black-box nature of Deep Neural Networks (DNNs) to its advantage to replace the *if this, then that* condition into a convolutional network, making it virtually impossible to exhaustively enumerate all possible pathways and conditions.
- **Control Flow Manipulation**: Exploits implicit control flow mechanism conducted by Windows OS through callbacks, enumeration functions, thread local storage (TLS), etc.
  - **Thread-Hiding**: Prevents debugging events from reaching the debugger. This is done through the *NtSetInformationThread()* function to set the *HideThreadFromDebugger()* field of the *ETHREAD* kernel structure. Counter: hooking the involving functions.
  - **Suspending Threads**: Halting the process of the (user-mode) debugger, leveraging *SuspendThread()* or *NtSuspendThread()* from *ntdll*.
  - **Multi-Threading**: Spawn a separate thread to perform decryption routines or execute part of its malicious code. Counter: set breakpoint at every point.
  - **Self-Debugging**: Prevents the debugger from successfully attaching to the malware, e.g. running a copy of itself and attaching to it as a debugger. This works because by default each process can be attached to only one debugger.
  - **Load Malicious Libary**: Uses dynamic library functions to perform malicious behavior.
- **Lockout Evasion**: Impedes the working of the debugger without having to look for its presence. For instance the *BlockInput()* function could be used,

through which the malware prevents mouse and keyboard inputs until its conditions are satisfied, or a feature in Windows NT-based platforms that allow the existence of multiple desktops through the *CreateDesktop()* and *SwitchDesktop()* functions.

- **Fileless Malware**: See Anti-AV:Fileless Malware.

Anti-Emulation Classification

A collection of techniques aimed specifically at full system emulation through the use of emulators.

- **Fingerprinting**: Analogous to 3.4.2, but for emulators, such as Wine[6].

Anti-Instrumentation Classification

A collection of techniques aimed specifically at dynamic binary instrumentation tools, such as Intel Pin.

- **Code Cache Artifacts**: DBI tools execute an instrumented version of the program in a particular memory region called a *code cache*.
  - **EIP Detection**: A sample can detect if the Instruction Pointer (EIP) is different from the expected one. It exploits instructions that save the execution context e.g. current value of the EIP register differs from the base address of the main module of the program. Counter: track instructions that leak the effective EIP and block this information leakage to execute a user defined function that patches the value inside registers or memory with the expected one.
  - **Self-modifying Code**: Code that changes its instructions at runtime by overwriting itself. Counter: force the DBI tool to build a new trace with the correctly patched code and abort the execution of the wrong one.
- **Environment Artifacts**:
  - **Parent Detection**: The instrumented program can check if the parent process is the expected one and not a DBI tool. Counter: hide the parent process name e.g. by hooking a function and faking the returned structure or denying the program to open *CSRSS.exe*.

[6] https://en.wikipedia.org/wiki/Wine_(software)

  - **Memory Fingerprinting**: Identify presence of artifacts that DBI tool leaves in memory e.g. scan entire memory of the process looking for allocated pages containting DBI artifacts (strings, code patterns). Counter: hide all memory regions where DBI related code and data reside (whitelisting of memory pages).

- **JIT Compiler Detection**

  - **DLL Hook**: DBI tools insert jumps (hooks) at the beginning of some functions to intercept the execution at the end of a not instrumented trace. Instrumented programs can check if initial instructions of common functions have been modified. Counter: label the addresses as *protected* and store the original values in a separate memory region that the program is redirected to.
  - **Memory Page Permissions**: JIT compilers write and execute significant amounts of code, hence the number of pages marked RWX inside the address space is considerably larger in respect to the execution of the non-instrumented version. A binary can scan the process address space and count the number of pages with RWX permissions to see if a DBI is present. Counter: whitelisting of memory pages.
  - **Memory Allocations**: JIT compilers intensely use RWX memory pages to instrument and cache instrumented traces. To achieve this, memory must be allocated by the DBI tool, of which the number of invocations of the low-level Windows API *ZwAllocateVirtualMemory* can be counted and compared to the number for non-instrumented execution. counter: redirect every write attempt of the process inside the `.text` segment of libraries + fake the results of suspicious read instructions that have their target address inside the `.text` segment of libraries.

- **Overhead Detection**: Tools introduce measurable overhead. Countermeasure tries to fool the process into thinking less time has elapsed.

  - **Windows Time**: Windows APIs *GetTickCount* and *timeGetTime* retrieve time information by accessing fields in a shared user page at the end of the process memory space. Counter: implement "Fake memory", i.e. intercept read of *TickCountMultiplier* and divide it by *TICK DIVISOR* (user defined) before returning it or reassemble *High1Time* and *LowPart* of the *_KSYSTEM_TIME* struct, divide it by user defined divisor, split it again and return the read instruction or hook *NtQueryPerformanceCounter* and divide the field *QuadPart* of the *LARGE_INTEGER* struct by a user defined constant.
  - **CPU Time**: The *rdtsc* instruction returns the processor timestamp defined as the number of clock cycles elapsed since last reset, of which the high part and low part are respectively in the *EDX* and *EAX* registers. Counter: recognize *rdtsc* instruction in a trace and insert a call to a

function that composes the integer by combining the *EDX* and *EAX* registers, divides it by a user defined constant, and patches the value of the two registers using this new value.

- **Stalling**: See Anti-Sandbox:Stalling.

3.4.3 Discussion

This taxonomy was formed through an extensive review of existing literature, as listed in Section 2.4.1. Now, the taxonomy provided in this work is compared to these existing works to highlight its differences and improvements.

Compared to [23] and the anti-analysis techniques in [54], this taxonomy expands on their classification by including anti-instrumentation techniques targeting binary instrumentation tools like Intel Pin (see Section 2.4.4). Additional tactics and techniques that were underrepresented in existing taxonomies were also added, such as spoofing and checking integrity. These additions were informed by the existing CAPE signatures, classifications from automated malware analysis environment detection tools, and the CheckPoint Research Evasion Encyclopedia [8].

In [34], the lumping together of distinct environments such as debuggers and instrumentation tools into one anti-debugging category and VM and Sandbox environments into anti-virtualization can obscure the specific techniques used, making it difficult to discern what environment malware targets when using anti-debugging or anti-virtualization without the additional context. The classification proposed in this work avoids this issue by organizing techniques according to their specific environments.

3.5 Creating a Database

In this work, an approach for identifying undetected techniques in existing analysis tools, specifically CAPEv2, has been outlined and practically applied. However, the primary contribution of this research is the creation of an evasive malware database. This section will delve into the design considerations for developing such a database.

Selecting the Database Type Given to the vast number of categories, tactics, and techniques in the field, a database limited to querying malware by specific techniques would be insufficiently flexible. Therefore, the goal of the database presented in this work is to be modelled after a defined taxonomy, allowing users to query malware at various levels of the hierarchy, providing greater flexibility.

There are several types of databases to consider, including relational, NoSQL, object-oriented, hierarchical and network databases. When selecting the database type, it is crucial to evaluate their respective advantages and disadvantages in relation to the application context. In this particular context, the most important factor is the relationship between malware and the techniques they employ. Additionally, the techniques exist within a complex, extended hierarchy that must be accurately

represented in the final database. Due to the limitations of classic relational databases in modeling these intricate relationships, they will not be considered futher.

Both hierarchical and object-oriented databases organize data in a tree-like structure, with parent-child relationships where each child has a single parent, though a parent can have multiple children. While these types of database are simple and efficient for data retrieval, they lack flexibility in managing more complex relationships. Although the taxonomy outlined in this thesis currently presents clear hierarchical relationships, it is foreseeable that certain techniques or tactics may eventually span multiple categories due to their commonality across different environments. In fact, introducing shared relationships in this taxonomy - or future ones based on it - could simplify and enhance the structure. Therefore, selecting a database that limits this flexibility would be counterproductive, leading to the decision not to use hierarchical or object-oriented databases for this purpose.

A network-type database addresses the limitations of hierarchical models by allowing many-to-many connections between parents and children, offering greater flexibility and the ability to manage more complex relationships. This makes it a strong candidate for our database, especially given its efficiency in enforcing hierarchical structures. However, even within our current taxonomy, situations may arise where nodes need to connect arbitrarily to one another, which strict hierarchies cannot accommodate. For example, consider a deeper categorization within the *Reverse Turing Tests* tactic. The I/O technique alone, could branch into multiple subtechniques, such as mouse events - tracking mouse movement speed, clicks or other interactions. This introduces a challenge to the strict hierarchy, as malware might need to link to both the technique and subtechnique level, a scenario unmanageable in a network-type database. To maintain the flexibility for deeper classification, this option must be set aside in favor of NoSQL graph databases.

NoSQL graph databases provide the highest flexibility in modeling relationships, as their graph structure allows for non-strict hierarchies where any node can connect to any other node. This makes them the ideal solution for the purposes of this work. These databases are particularly suited for use cases where relationships are more critical than the data itself. They excel in querying and uncovering patterns within complex relationships and offer a flexible data model that can easily expand over time, making them a perfect fit for the needs of this thesis.

Modeling the Database The taxonomy is modeled using a directed graph, where each hierarchical level - such as evasion-type (e.g., Anti-Debugging), tactic (e.g., Fingerprinting), technique (e.g., Parent Check) and malware - is represented as a node with the respective label. Directed edges connect these nodes, indicating parent-child relationships within the hierarchy. For example, a *tactic* node is connected to an *evasion-type* node via a *tactic_of* edge, and a *technique* node is connected to a *tactic* node via a *technique_of* edge. Malware sample nodes are linked to the techniques they use through a *uses* edge. The highest level category, the catalog category, serves as the root and contains a single node, the “Malware Evasion Technique Catalog”. Evasion-type nodes are linked to this root node via an

evasiontype_of edge. The structure of the graph database is illustrated in Figure 3.6.

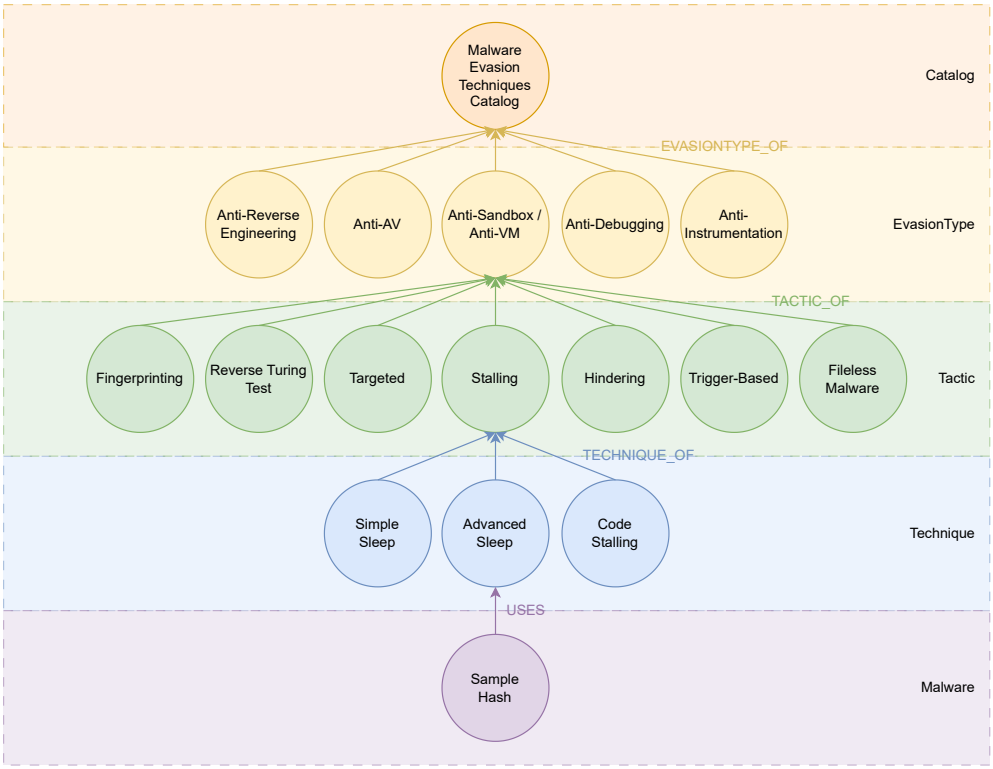


Figure 3.6: Structure of the Graph Database

The evasive malware database developed by this work will include not only the techniques identified as undetected by CAPE but also leverage existing CAPE signatures. This approach broadens the detection surface, thereby increasing the number of samples classified as evasive and reducing the number of samples required to build a substantial database. To achieve this, all signatures have been compiled into an SQLite file, with an associated column indicating the respective identifier for each evasion technique node in the modeled graph database. An example row from this mapping table is shown in Table 3.3, where “T00002” serves as the identifier for the “Execution Environment” technique under the Fingerprinting tactic within the Anti-Sandbox evasion type. This setup facilitates the direct mapping of CAPE signatures to the taxonomy, enabling the programmatic population of the graph database with malware samples through the parsing of analysis results.

id	filename	name	description	category	mapping
22	...	"PiratedWindows"	"Check if the..."	["anti-sandbox"]	T00002

Table 3.3: Example Entry in the Mapping Table

The graph is created using Apache AGE, and it can be easily visualized with the open-source Apache AGE Viewer tool[7].

Discussion The novel modeling of a database of evasive malware using a graph-based approach is believed to be the first of its kind. This approach is anticipated to offer several key benefits, including a clear visual representation of complex relationships among evasion techniques and malware samples. Additionally, the graph structure is designed to support querying at various abstraction levels derived from the taxonomy, which is expected to facilitate more nuanced analysis. The effectiveness of this querying capability will be evaluated in the next chapter, which will assess how well it supports detailed exploration and understanding of the data. Moreover, the graph-based model's scalability is expected to provide long-term advantages as new techniques and samples are incorporated, potentially enhancing its value as a research resource.

3.6 Conclusion

This chapter detailed the methodology for extending CAPEv2's malware detection capabilities and constructing a comprehensive evasive malware database. Initially, techniques were identified and signatures developed to enhance CAPEv2's detection of previously undetected evasion methods. The process of creating and implementing these signatures was outlined, including the necessary steps for integrating new hooks and defining their behavior within CAPEv2.

Following the signature development, the chapter described the design of a NoSQL graph database using Apache AGE to model the complex hierarchical relationships of the malware taxonomy. This database design supports flexible querying and efficient management of hierarchical data, allowing for detailed analysis and flexible querying of malware data. By incorporating both new and existing CAPE signatures, the database enhances detection coverage and facilitates automatic population with relevant malware samples. This approach ensures a robust framework for analyzing and understanding evasive malware techniques.

[7] https://github.com/apache/age-viewer

Chapter 4

Evaluation

In this chapter the effectiveness of the developed signatures is evaluated using a control set that includes custom binaries and modified automated tools. Additionally, the evaluation extends to a broader dataset comprising both evasive and non-evasive samples. The chapter also examines the querying capabilities and limits of the database.

4.1 Outline

The key steps of the evaluation process are briefly outlined below, offering a concise guide to the different steps involved in evaluating the signatures of the previously undetected techniques and the graph-database.

Environment Setup The environment setup is detailed to ensure the results of the experiments are reproducible.

Undetected Technique Signature Evaluation The signatures for the previously undetected techniques, which will from now on be referred to as the "new signatures" will be evaluated in two separate steps:

1. Evaluation on a control set.
2. Evaluation on a general distribution set.

The meanings of the control and general distribution set are detailed below.

Control Set The effectiveness of the signatures is first evaluated using a control set. This control set includes binaries compiled from automated malware analysis environment detection tools, designed to test only one technique at a time (refer to subsection 3.2.1), as well as the custom binaries developed based on existing literature on evasion techniques (see subsection 3.2.2). By creating binaries that isolate individual techniques, the analysis reports contain only relevant information, facilitating a clearer assessment of the results and simplifying the identification of

false positives. Additionally, these control binaries are relatively easy to create. Automated tools typically offer configurations that allow for the creation of binaries tailored to specific techniques. Similarly, custom binaries based on literature often come with code samples that expedite development, as seen with Checkpoint Evasions and the Unprotect Project). Since no reverse engineering knowledge is required - thanks to open-source nature of the tools and the comprehensive explanations in literature for the others - creating these binaries is straightforward.

General Distribution Set The effectiveness of the signatures is next evaluated using a large set of malware that represents a more general distribution of malware samples than the control set. This general distribution set includes a combination of real-world evasive and non-evasive malware. The set is not labeled, meaning there is no indication of the samples as being evasive or not.

Database Evaluation Evaluation of the database is directed towards the flexibility of querying within the database. With flexibility of querying the following is meant:

Flexibility The flexibility of a database involves assessing if the database can handle queries at various abstraction levels. In terms of flexibility the focus will be on the different abstraction levels present because of the inherent modeling of the taxonomy.

4.2 Environment Setup

All malware samples are executed within a guest system configured with the Windows 10 22H2 operating system (x86-64 architecture). The decision for Windows 10 is based on its dominant market share among Windows versions since January 2018 [7], making it a representative and relevant choice for analyzing malware. The guest system operates under a Kernel-based Virtual Machine (KVM) hypervisor hosted on an Ubuntu 22.04 LTS machine with a AMD Ryzen 7 7735HS@4829 MHz (max). The Ubuntu 22.04 LTS version was selected because it was the latest Long-Term Support (LTS) version available at the onset of this thesis work on CAPE. This setup aligns with the recommended setup for CAPE [3].

The guest machine is provisioned with 2 virtual CPU cores and 4096 MB of RAM, providing sufficient processing power and memory to handle the execution of potentially resource-intensive malware. A 20 GB virtual hard disk is utilized, offering ample storage space for the operating system, some default user programs, fabricated personal files, and any temporary files created during the malware analysis process.

A critical timeout of 180 seconds is set for each sample, striking a balance between allowing enough time for evasive behaviors to manifest and avoiding unnecessary delays in the overall analysis timeline. No additional analysis options within CAPE's submission utility are enabled during the testing phase.

4.3 Evaluation on Control Set

Goals of the Evaluation The aim of this evaluation is twofold:

1. Remove any false positives within the techniques.

2. Verify whether the signatures correctly trigger for the intended techniques, without incurring false positive detections from the other signatures.

Through the control set it should be possible to analyze the technique in a practical setting that enables to verify its behavior is correct. The Furthermore, in this control set there should be minimal interference for each technique, meaning false positives can be easily spotted.

False Positive Techniques Thanks to the evaluation on the control set, discrepancies can be observed. For instance, a technique using the Windows API function `NtSystemDebugControl` resulted in a false positive. This was evident from comparing the analysis report with the source code, which is as follows:

```
BOOL NtSystemDebugControl_Command() {
        auto NtSystemDebugControl_ =
        ↪  static_cast<pNtSystemDebugControl(API ::
        ↪  GetAPI(API_IDENTIFIER :: API_NtSystemDebugControl));

        auto status = NtSystemDebugControl_(SYSDBG_COMMAND ::
        ↪  SysDbgCheckLowMemory, 0, 0, 0, 0, 0);

        const auto STATUS_DEBUGGER_INACTIVE = 0xC0000354L;
        const auto STATUS_ACCESS_DENIED = 0xC0000022L;
        if (status == STATUS_DEBUGGER_INACTIVE) {
                return FALSE;
        } else {
                // kernel debugger found
                if (status != STATUS_ACCESS_DENIED) {
                        // usermode debugger too
                }
                return TRUE;
        }
}
```

Upon executing the binary with the modified capemon that hooks the relevant functions, the analysis report showed the return value of `NtSystemDebugControl` as `0xffffffffc0000354`. This suggests that the check should have returned false, indicating that this particular implementation is a false positive from the Al-Khaser tool and cannot be used for CAPE detection.

Evaluation Methodology After eliminating false positives, the effectiveness of the signatures must be evaluated. CAPE's web interface provides a convenient way to verify this, as it displays all the signatures triggered by a given sample in the "Signatures" section. Figure 4.1 illustrates this for a custom binary that employs the CuckooTCP technique. For clarity, the descriptions of new signatures are prefixed with "[ByMaxim]", making them easily identifiable. Each binary in the control set is analyzed using CAPE. Next, the Signatures section of the analysis result is viewed in the web interface. If the description of the correct signature appears, and no other signature that is not expected, this signature can be marked as working successfully. If the expected description does not appear, the signature is marked as unsuccessful as it failed to detect the evasion technique in question. Signatures may be unsuccessful because they are flawed or they are too narrow and have to be reworked or improved. Signatures that are meant to detect other evasion techniques may also appear. In this case, these signatures also need to be marked as unsuccessful. This is because its presence likely indicates that the signature is too broad, as it is detecting an evasion technique in a binary that does not explicitly use that evasion technique. These signatures should also be reworked or improved and then tested on the same binary that previously triggered them unjustly. Evaluation and signature improvement should happen in an iterative procedure to guarantee optimal resulting signatures.

False Positive Signatures When false positives occur in the new signatures, the contents of the signature are re-evaluated in order to eliminate the false detection. As the signature is too broad, it has to be made more specific. If the granularity is smaller than using Windows API functions, CAPE's debugger allows to capture instruction traces of malware execution [3]. As this involves experience with assembly and reverse engineering, this is beyond the scope of this work.

Figure 4.1, in addition to the expected signature confirming that the CuckooTCP technique was correctly detected, shows a signature for `SetUnhandledExceptionFilter`. This signature, which is part of CAPE's existing set, is triggered whenever the `SetUnhandledExceptionFilter` function is called - a function that can be used by malware to set a custom exception handler, potentially to evade debugging. However, since this technique is not part of the custom binary, its detection suggests that the technique may be overly broad, leading to a false positive. This assumption is supported by the low confidence score of the signature, which is only 40%. As refining existing signatures is beyond the scope of this work - and considering that this was the only unexpected signature in our testing, this signature is excluded from the evaluation.

The other results of the control set evaluation are provided in Appendix C. All signatures provided are able to effectively detect the intended evasion techniques, while no other unexpected signatures appeared.

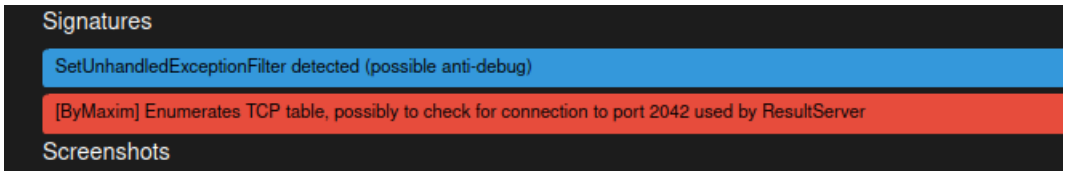


FIGURE 4.1: Signatures Section for CuckooTCP Custom Binary

Analysis of Results The first goal of the evaluation on the control set is to identify false positive techniques, so they can be removed. The results in Appendix C indicate that all techniques save one were behaving as expected. The `NtSystemDebugControl` technique turned out to be a false positive technique, meaning the implementation of the technique returned true while the result should have been false, thus incorrectly detecting the analysis environment where it should not have. This result highlights that the evaluation on the control set was effective at tracking false positive techniques, allowing to exclude these in the final set of evasion techniques that are able to detect CAPE.

The second goal of the evaluation on the control set is to assess the ability for the new signatures to correctly detect the techniques in their respective controlled custom binaries that employ only one particular technique, without incurring detections from the other new signatures. The results indicate that the new signatures were successfully able to detect their respective evasion techniques without incurring false positives from each other. However, some of the existing signatures within CAPE did appear unexpected - meaning a technique was detected when it wasn't explicitly used in the binary - when evaluating the new signatures, suggesting the set of existing signatures may contain some signatures that potentially have issues with specificity. These findings suggest that the new signatures are generally effective in detecting evasion techniques, though some refinement may be necessary for the existing signatures as well. The presence of false positive signatures such as the signature for UnhandledExceptionFilter highlights a limitation in the current signature use that does not exclude signatures with low confidence scores.

The successful detection of evasive techniques by these new signatures shows it is feasible to develop simple signatures using the available source code for techniques with minimal false positives when running unrelated binaries. A more detailed discussion of the limitations, comparisons with related work, future work, and possible threats to validity is presented in Section 5. The effectiveness of the new signatures on a larger scale, however, must be evaluated separately.

4.4 Evaluation on General Distribution Set

The analysis of the control set provides insights into whether the signatures are working correctly for specific implementations and whether the techniques were falsely flagged. However, this does not guarantee that the signatures will perform well across a larger set of real-world malware samples. Variations in implementations of these

techniques might exist, and signatures that perform well in controlled conditions may not necessarily exhibit the same effectiveness in a more diverse set of samples. Additionally, developing effective rule-based signatures is challenging, as they must strike a balance between being too narrow and too broad. Therefore, the primary goal of this evaluation is to measure the effectiveness of the signatures on a broader scale by evaluating on a set that represents a more general distribution of malware samples.

Goals of the Evaluation The primary aim of this evaluation is to assess the effectiveness of the new signatures in detecting evasion techniques that are able to evade CAPE in a real setting. Through the general distribution set it should be possible to get a realistic view of how the new signatures fare against actual malware instead of the safe binaries from the control set.

Evaluation Methodology To assess the effectiveness of the signatures on a broader scale, all signatures were tested on an extensive set of 1000 malware samples (evasive and non-evasive). Each signature's performance was measured by tracking how frequently it was triggered on this set of evasive samples, expressed as a percentage. The existing signatures in CAPE that were determined evasive and mapped to the taxonomy serve as a baseline for the evaluation of the new signatures for undetected techniques. The new signatures are compared with the existing CAPE signatures by organizing both in percentiles. This gives a broader view of what the performance is of most CAPE signatures for evasion techniques and how well the new signatures fare compared to these. The expectation is that the performance of the new signatures should be situated around where the largest group of CAPE signatures is.

Trigger Rates The trigger rates for the new signatures, expressed as percentages, are summarized in Table 4.1, while those for CAPE's existing evasion technique signatures evasion technique signatures - specifically those mapped to our taxonomy - are provided in Appendix D. Signatures that were not triggered by any of the analyzed malware samples have been omitted from both summaries to focus on the active detections. Of the 39 new signatures, only five (12.82%) were triggered: GetLastInputInfo, NtYieldExecution, ParentProcess, Uptime and Sleepskipping (ParallelDelays). In comparison, 89 out of 159 existing CAPE signatures (approximately 55.97%) were not triggered. This notable difference may indicate that the new signatures are highly specialized and tailored for specific techniques that the tested malware did not employ. This raises questions about whether the new signatures are too narrowly defined or whether the malware samples used were not representative of the full range of evasion techniques the signatures were designed to detect.

A deeper investigation into the correlation between the design of these signatures and the evasion strategies of the malware samples could provide more clarity. If the dataset used for this evaluation influenced the results by not being diverse enough to encompass the specific evasion techniques of the new signatures, future evaluations

Triggered (%)	GetLastInputInfo	NtYieldExecution	ParentProcess	Uptime	ParallelDelays
[90, 100[					
[80, 90[					
[70, 80[					
[60, 70[					
[50, 60[					
[40, 50[					
[30, 40[					
[20, 30[					
[10, 20[	13.2%		18.7%		
]0, 10[		5.3%		1.2%	1.2%

TABLE 4.1: New Signatures for General Distribution Set of Malware

could benefit from a more diverse set of samples that better align with the expected evasion behaviors these signatures were designed to detect. Furthermore, it should be noted that the NtYieldExecution signature is tied to a technique known for its inherent unreliability, which negatively impacts the accuracy of the signature and could lead to a higher rate of false positives, thus affecting the overall comparison.

Histogram The normalized bihistogram in Figure 4.2 compares the performance of the new signatures against CAPE's existing ones, with signatures that were not triggered being omitted to avoid artificially inflating the 0-10% range in the percentage of times signatures were triggered. This exclusion ensures a clearer differentiation between signatures that were triggered infrequently and those that did not trigger at all. This histogram reveals that the majority of existing signatures, were triggered between 0% and 10% of the time, with only 9 signatures (12.86% of those triggered) exceeding this threshold. This might suggest that many of CAPE's existing signatures are effective in only a small subset of cases, potentially indicating that these techniques are either not widely used or that signatures may be too specific. In contrast, 2 out of 5 new signatures (40%) were triggered between 10% and 20% percent of the time. This suggests that while the new signatures were generally less likely to trigger, those that did may offer a higher detection quality relative to their number. However, given the small sample size of new signatures, these observations should be interpreted with caution, and further testing is required to validate these preliminary findings.

Statistical Significance To further assess the difference between the new and existing signatures, a statistical analysis was conducted. The mean trigger rate for the new signatures was found to be 7.20%, with a standard deviation of 7.06%, compared to the existing signatures' mean trigger rate of 4.61% and a standard deviation of 11.18%. The null hypothesis of this experiment is: "The distributions of trigger rate of the new signatures and the existing CAPE signatures are not significantly different." and the alternative hypothesis is: "The distributions of the

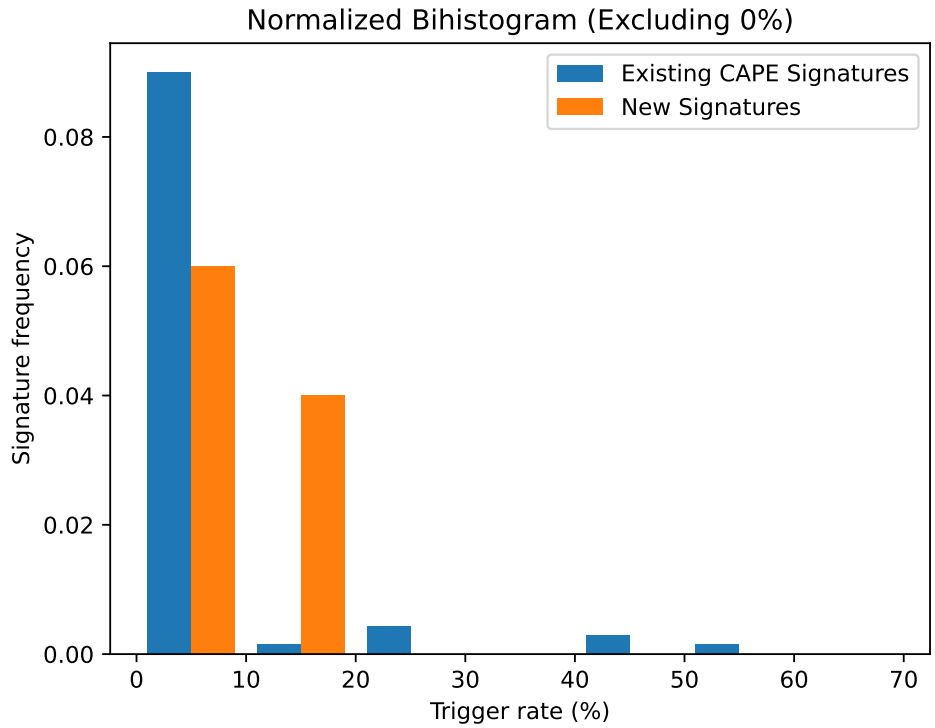


Figure 4.2: Normalized Bihistogram of the New and Existing Signatures

trigger rate of the new signatures and CAPE's existing signatures are significantly different.". A Kolmogorov-Smirnov test was performed that compares the cumulative distributions of the two datasets. The test statistic, known as the D-statistic, was calculated to be 0.657, and the corresponding p-value was 0.019. Given that the p-value of 0.019 is below above the common significance level of 0.05, it indicates that there is enough evidence to reject the null hypothesis. Thus, we conclude that there is a significant difference between the distributions of the trigger rates of the new signatures and the existing CAPE signatures. To complement this analysis, Cohen's D was calculated and found to be -0.24, indicating a small effect size. This negative value suggests that, on average, the new signatures trigger slightly more frequently than the existing ones. However, a 95% confidence interval for the difference in means was calculated to be [-9.307, 4.133]. The inclusion of zero within this interval indicates that, while the K-S test detected a significant difference in the overall distributions, the difference in means between the new and existing signatures is not statistically significant.

Although the effect size indicates a small advantage of the new signatures, the difference is not statistically significant, meaning that while there is a trend towards better performance with the new signatures, this trend is not strong enough to rule out the possibility that it could be due to random variation. Therefore, it cannot be conclusively proven that there is a statistical difference in signature quality between

the existing and new signatures at this time. Further testing with a larger sample size would be necessary to more definitively evaluate their comparative effectiveness.

Analysis of Results The primary goal of the evaluation on the general distribution set is to assess the effectiveness of the new signatures in detecting evasion techniques that are able to evade CAPE in a real setting. The results described above indicate that the trigger rate of the new signatures is at an acceptable level, with their trigger rate potentially even being slightly above the existing signatures for evasion techniques within CAPE, although the difference is not statistically significant to be able to conclude this without further testing with a larger sample size of new signatures. However, a large group of signatures did not trigger at all, suggesting potential issues in the signatures with specificity or the diversity of the database. These findings suggest that the new signatures are generally effective in detecting evasion techniques within the used dataset, though some refinement may be necessary to the signatures or the dataset to improve detection rates. While the evaluation was thorough, the presence of potential false positives such as indicated for the signature of the NtYieldExecution technique highlights potential limitations in the current signature designs and calls for further refinement and testing to ensure more precise detection capabilities.

The success of some of the new signatures is proof that the approach taken to enhance existing analysis tools such as CAPE could be a valid approach to detect techniques of which it was not possible to detect them prior. However, this approach has limitations, which are outlined in Section 5, along with comparisons with related work, future directions and possible threats to its validity. This approach provides a foundation for further improving real-world detection systems and keeping them relevant in the eye of growing malware sophistication. Furthermore it solidifies CAPE's flexibility as an analysis tool that is able to detect these increasingly sophisticated evasive malware.

The new signatures developed here aim to detect evasion techniques able to circumvent CAPE. Consequently, some evasion techniques used by malware to bypass other sandboxes or analysis environments may not be detected by these CAPE-specific signatures.

Having evaluated the effectiveness of the signatures on both a controlled set and a broader distribution of malware samples, the next step involves exploring the querying capabilities of the graph-based database designed to organize and analyze these findings.

4.5 Evaluation of the Database

In this section the querying capabilities of the graph-based database are explored. It does this by examining the flexibility of the database by showing how well it can handle queries at various abstraction levels of the modelled taxonomy.

4.5.1 Querying the Database

Goals of the Evaluation The primary goal of this evaluation is to assess the flexibility of the querying of the graph database. Better flexibility allows to maximally utilize the modelling of the taxonomy and for users to extract more specific and relevant data with their queries. With better query flexibility, users can explore data in various ways, which may lead to relevant insights and trends that may not have been apparent from other database structures.

Evaluation Methodology The flexibility of the graph database will be examined. Flexibility is evaluated through example queries that verify querying at different abstraction levels of the taxonomy, that is modeled within the database, is possible. Example queries, that would be found in practical use cases, are presented for the evaluation, allowing a realistic but practical setting for the exploration of query flexibility. The results of the queries are visually examined for the query's correctness and are displayed as evidence if the result is visually comprehensive. Below, the various abstraction levels of the taxonomy at which querying can be done are explored, one at a time.

Querying at the Malware-level

Being primarily a database of evasive malware classified on evasion techniques, it should be possible to query the malware and retrieve the techniques it employs. Malware can be directly queried using the name of the malware, which is usually the hash of the file:

```
SELECT * from cypher('metx', $$
MATCH (h: Malware {name:
'0b34f158e68709d2b7905fd2a596ac3b6b93db66c48dd82d818da786a604d61b'
})-[i:USES]-(j:Technique)
RETURN h, i, j $$) AS (h agtype, i agtype, j agtype)
```

The techniques the malware employs, in accordance with the provided taxonomy, are listed as well by this query. The edges between the malware node and the technique node contain the implementation of the technique, which is denoted through the signature that detected the technique, and is visible through hovering over the edge in AGEViewer. The result of the query, visualized by AGEViewer, is presented in Figure 4.3.

Having shown that querying at the Malware-level - the lowest possible abstraction level - is possible, the next step is to query the malware from the second lowest abstraction level, namely the technique-level.

Querying at the Technique-level

This query allows to find all malware connected to the *I/O* technique of the *Reverse Turing Test* tactic, located in the *Anti-Sandbox* evasion type. The result of the query, visualized by AGEViewer, is presented in Figure 4.4.

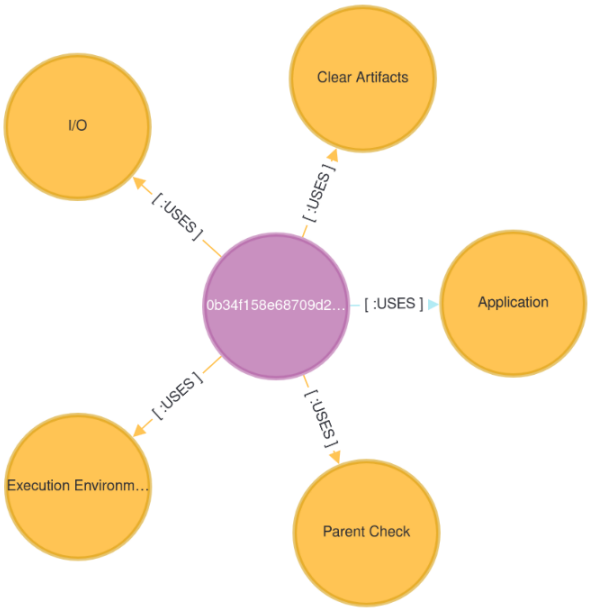


FIGURE 4.3: Querying Malware by Name and Retrieving Techniques

```
SELECT * FROM cypher('metx', $$
MATCH (h : Malware) - [i : USES] -
(j : Technique {name : 'I/O'}) - [k : TECHNIQUE_OF] -
(l : Tactic {name : 'Reverse Turing Test'}) - [m : TACTIC_OF] -
(n : EvasionType {name : 'Anti-Sandbox'})
RETURN h, i, j, k, l, m, n $$) AS (h agtype, i agtype, j agtype , k
↪  agtype, l agtype, m agtype, n agtype);
```

Much of this query is actually redundant. The reason the higher vertex levels in the hierarchy - such as the tactic and the evasion type - are even returned is purely for illustration purposes, showing that the graph database is able to find the correct technique that was specified. It is also possible to be much less specific:

```
SELECT * FROM cypher('metx', $$
MATCH (h : Malware) - [i : USES] -
(j : Technique {name : 'I/O'})
RETURN h, i, j $$) AS (h agtype, i agtype, j agtype);
```

This query will not match the higher abstraction levels to the right names and will not return them either. This is completely fine if the name of the technique is unique and the user only requires the technique vertex and its connections to the malware using it. Note that it is also possible to only return the malware vertices, without the edges and the technique vertex.

While mentioned that there is no problem if the vertex name is unique, this is not necessarily a problem if it was not the case either, as vertices all have unique

ids which enables identical names for vertices. Combined with the flexibility of the graph structure, this allows for very flexible querying. For illustration, assume there is another technique in the taxonomy called *I/O* as well. This new one could be placed anywhere in the taxonomy. When querying non-specifically like this, the user will retrieve two separate groups of connected vertices: a group with the one under *Reverse Turing Test* of the *Anti-Sandbox* evasion type and a group with the new one. This could give interesting insights to the user, such as different implementations being present under different evasion types like anti-sandbox and anti-debugging, or it could show him related techniques with similar names. This further solidifies the usefulness of a graph-based approach.

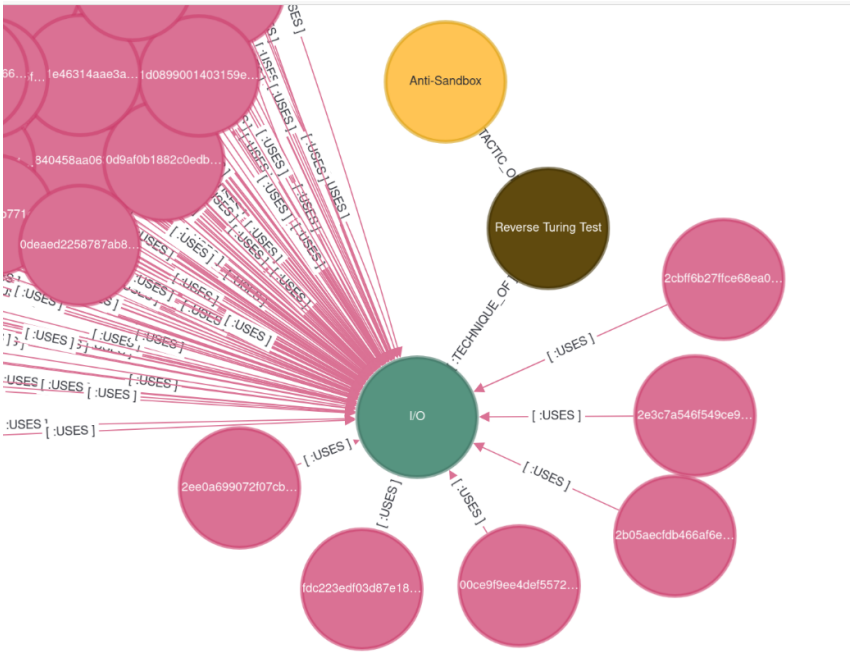


Figure 4.4: Querying Malware from the Technique-level

The next abstraction level, one above the technique-level, to query from is the tactic-level.

Querying at the Tactic-level

This query allows to find all malware connected to the *Fingerprinting* tactic of the *Anti-Sandbox* evasion type. The visualization by AGEViewer is presented in Figure 4.5.

```
SELECT * FROM cypher('metx', $$
MATCH (h : Malware) - [i : USES] -
(j : Technique) - [k : TECHNIQUE_OF] -
(l : Tactic {name : 'Fingerprinting'}) - [m : TACTIC_OF] -
(n : EvasionType {name : 'Anti-Sandbox'})
```

```
RETURN h, i, j, k, l, m, n $$)
AS (h agtype, i agtype, j agtype , k agtype, l agtype, m agtype, n
↪  agtype);
```

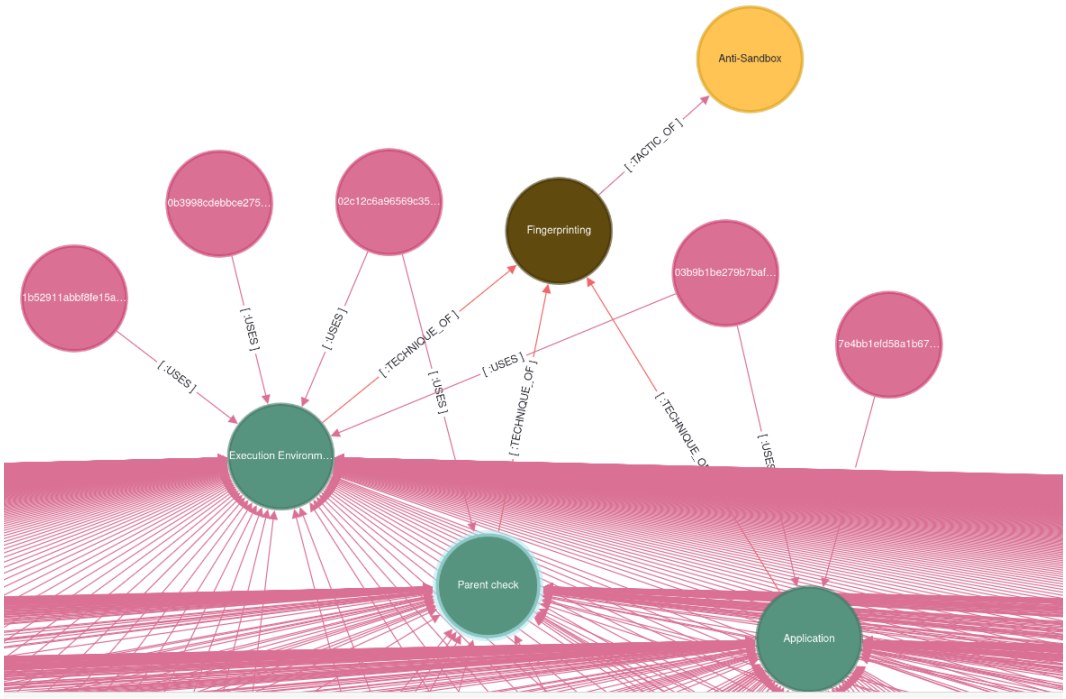


FIGURE 4.5: Querying Malware from the Tactic-level

Note that the same remarks hold about the specificity in this query related to the evasion type-level as did in the previous subsection: the user can choose to be unspecific at the evasion type-level to select more vertices that may be related.

The final, and highest, abstraction level to query at (that does not return the entire database) is that of the evasion type.

Querying at the EvasionType-level

The following query returns all malware connected to the *Anti-Sandbox* evasion type.

```
SELECT * FROM cypher('metx', $$
MATCH (h : Malware) - [i : USES] -
(j : Technique) - [k : TECHNIQUE_OF] -
(l : Tactic) - [m : TACTIC_OF] -
(n : EvasionType {name : 'Anti-Sandbox'})
```

```
RETURN h, i, j, k, l, m, n $$)
AS (h agtype, i agtype, j agtype , k agtype, l agtype, m agtype, n
↪  agtype);
```

The query returns the entire chain from the *Malware* vertex to the *EvasionType* vertex with name "Anti-Sandbox". This amounts to about 1600 rows in table-format. Here is an excerpt of one such row representing a connection from a Malware vertex to the AntiSandbox EvasionType vertex, presented in JSON format, as it is not possible to show the full visual result in a comprehensive way:


```
{"id":2814749767106661,"label":"Malware","properties":
{"name":"00e04114cba63c061e5e8ba4191bf934a7dfcf0e64b6c12f5580a58f0b1c496c",
"Implementations":"[name:antivm_checks_available_memory],
[name:queries_keyboard_layout],[name:NtYieldExecution],
[name:antidebug_setunhandledexceptionfilter],[name:stealth_timeout],
[name:antidebug_guardpages],[name:reads_self],
[name:terminates_remote_process],[name:cape_extracted_content],
[name:injection_rwx],[name:infostealer_cookies],[name:persistence_autorun],
[name:persists_dev_util],[name:procmem_yara],[name:GetLastInputInfo],
[name:binary_yara],[name:infostealer_ftp],[name:infostealer_keylog],
[name:infostealer_mail],[name:removes_zoneid_ads],
[name:cape_detected_threat]"}}
```


Vertices with the *Malware* label contain some useful information:

- **id:** A unique identifier for this vertex. This is a recurring property across all vertices.
- **label:** The name of the label the vertex may have. This is a recurring property across all vertices.
- **properties:** A list of vertex specific properties. This is a recurring property across all vertices.
  - **name:** The name of the vertex. In the case of malware the name of the file is usually a hash.
  - **Implementations**: A list of signatures that was detected for the malware.

The malware vertex is connected to the Technique vertex with the *I/O* name. The JSON representation of this vertex is shown below.


```
{"id":1688849860263945,"label":"Technique","properties":
{"id":"T00009","name":"I/O","Implementations":"
[id:i00075;name:antisandbox_foregroundwindows],
[id:i00078;name:antisandbox_mouse_hook],[id:i00539;name:get_clipboard_data],
[id:i00629;name:GetLastInputInfo]"}}
```

Vertices with the *Technique* label are very similar to *Malware* vertices. The difference is in the semantics of the *Implementations* property:

- **properties:** A list of vertex specific properties. This is a recurring property across all vertices.
  - **Implementations**: A list of signatures that indicate the use of this technique.

As is evident from this description, if the malware sample was detected as having used a technique that triggered a signature in this *Technique* node, they are connected to each other. These signatures are either existing ones from CAPE or new ones that were mapped to the evasion technique taxonomy. The connection between the vertices takes the form of a *USES* edge. The edge connecting the vertices above is shown here, in JSON representation:


```
{"id":3096224743817221,"label":"USES",
"end_id":1688849860263945,"start_id":2814749767106661,
"properties":{"Implementations":"[name:GetLastInputInfo]"}}
```


Edges with the *USES* label also have useful information contained in them:

- **id:** A unique identifier for this edge. This is a recurring property across all edges.
- **startid:** The unique identifier of the source vertex. This is a recurring property across all edges.
- **endid:** The unique identifier of the target vertex. This is a recurring property across all edges.
- **label:** The name of the label the edge must have. This is a recurring property across all vertices.
- **properties:** A list of edge specific properties. This is a recurring property across all vertices.
  - **Implementations:** The signature that warrants the connection between the source vertex and the target vertex.

In this case, the reason for the edge's existence is the *GetLastInputInfo* signature, as one of the common signatures between the *Malware* vertex and the *Technique* vertex. For flexibility, every signature gets its own personal *USES* edge.

The *Technique* vertex is then further connected to the *Tactic* vertex with the *Reverse Turing Test*, which is itself connected to the *EvasionType* vertex with the *Anti-Sandbox* name, as the query indicates.

Querying Part of the Taxonomy

The same things are possible as above, but without showing the malware nodes, making it possible to easily query the taxonomy as well. The below query illustrates this, by listing all techniques that belong to the *Anti-Sandbox* evasion type. It's result is visible in Figure 4.6

```
SELECT * FROM cypher('metx', $$
MATCH (h : EvasionType {id: 'AS1'}) - [i : TACTIC_OF] -
(j : Tactic) - [k : TECHNIQUE_OF] - (l: Technique)
RETURN h, i, j, k, l
$$) AS (h agtype, i agtype, j agtype, k agtype, l agtype);
```

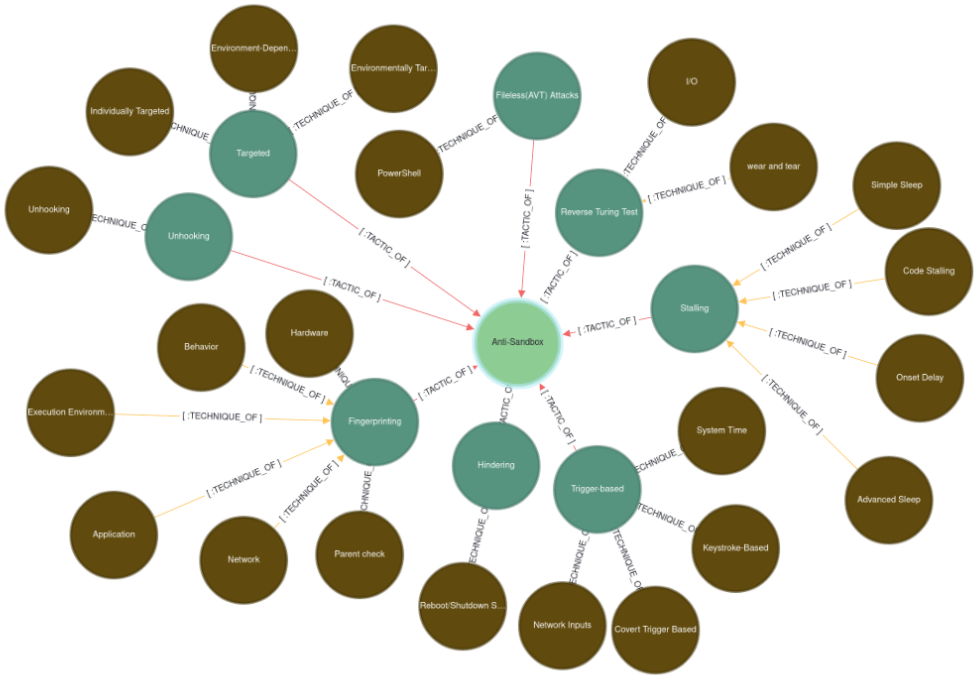


Figure 4.6: Anti-Sandbox Part of the Taxonomy

Note that in case of querying the taxonomy it may even be useful to query higher than the evasion type level. For instance, to retrieve all evasion types, the catalog level could be used in the query:

```
SELECT * FROM cypher('metx', $$
MATCH (h : Catalog) - [i : EVASIONTYPE_OF] -
(j : EvasionType)
RETURN h, i, j
$$) AS (h agtype, i agtype, j agtype);
```

Analysis of Results This graph-based approach offers considerable flexibility for querying the data. Users can query for malware vertices connected to specific technique vertices, or they can query for malware at higher hierarchical levels, such as tactics or evasion-types. Additionally, the graph model allows for querying the taxonomy itself. Beyond its flexibility, the graph format provides a clear and intuitive visual representation of the data.

4.6 Conclusion

This chapter evaluated the effectiveness of the developed signatures on both a controlled set of custom binaries and a broader distribution of real malware samples. The control set analysis confirmed that the signatures effectively detect specific evasion techniques when implemented in isolation. However, this evaluation also highlighted potential discrepancies, demonstrating the importance of scrutinizing the results to identify and omit false positives, as was evident in the analysis of the `NtSystemDebugControl` technique.

Expanding the evaluation to a general distribution of malware samples revealed the challenges inherent in applying these signatures more broadly. Only a limited number of signatures triggered somewhat frequently, indicating a potential problem with specificity or that the dataset. The results also underscored the limitations of the current scope, acknowledging that some techniques may go undetected due to variations in implementation or the focus on CAPE-specific evasion tactics. Despite these limitations, the signatures represent a valuable step towards improving the detection of evasive techniques in automated malware analysis environments.

Afterwards the evaluation shifted from the new signatures to the graph database. For the graph-database the flexibility of the querying was evaluated through example queries, representing practical use cases, from different abstraction levels of the underlying taxonomy. Visual inspection of correctness and the practical example queries illustrated that querying at different levels of abstraction is feasible for both querying malware, as querying the taxonomy itself.

Chapter 5

Discussion

In total 39 new signatures were added and 159 existing signatures were mapped to the taxonomy. The signatures present represent 52 distinct techniques from the taxonomy.

5.1 Limitations

5.1.1 Limitations of New Signatures

As the control set offers minimal interference between samples, there is the possibility that problems for the signatures will not be visible through the evaluation on this set. This is because the signature could be detecting too broad on certain edge cases that are not apparent from the set of minimally interfering control samples. A possible solution would be to evaluate the signatures on real-world evasive malware samples making use of many evasion techniques at once, such as by evaluating on a large set of samples much like the experiment on the general distribution set. However, without the knowledge that the sample is not using the technique, one cannot be certain that something is a false positive. This highlights the need for a ground truth of evasive malware samples.

Given that reverse engineering the malware samples is beyond the scope of this work, it is acknowledged that some samples may include variations of techniques not detected by the provided signatures. Conversely, some signatures might have triggered on samples that did not actually use the intended technique. These potential false positives are acknowledged, and a more detailed manual analysis to verify the techniques used by the malware is suggested for future work.

5.1.2 Limitations of the Database

The evaluation of the database in this thesis was limited to the flexibility of the queries, however, a database has many more properties than just flexibility. Important properties for consideration include scalability, storage, performance, and query expressivity. The evaluation of these properties for the graph-database is left up to future work, but some possible limitations are detailed below.

At this point, the database does not include any indexing. With approximately 1000 malware samples, the querying is still relatively fast. However, were the database to scale significantly the query performance would likely decline without indexing.

Graphs are complex structures, requiring storage for nodes, edges and all their associated properties. This means a graph-database such as this may require much more storage space than a traditional relational database.

5.2 Comparison to Related Work

5.2.1 Malware Classification

Or-Meir et al. present various methods for classifying malware, including by type, malicious behavior, and privilege level. These classifications are not relevant to the evasion technique taxonomy and are not considered.

Although the work by Afianian at al. [23] has significantly influenced the taxonomy used in this thesis, this thesis expands on their classification by including anti-instrumentation techniques targeting binary instrumentation tools like Intel Pin (see Section 2.4.4). Additional evasion types, such as anti-reverse engineering and anti-AV techniques, were also incorporated. These additions were informed by the work of [53].

The work by Chen et al. categorized fingerprinting evasion techniques targeting systems based on evasion and debugger characteristics [31]. Their subclasses consisting of hardware, environment, application and behavior were taken over and used as techniques of several fingerprinting tactics within the provided taxonomy in this thesis.

A taxonomy of evasion techniques used by advanced persistent threat (APT) malware was offered by Sharma et al [54]. Their evasion techniques are categorized into stealth mechanisms, anti-analysis mechanisms, and covert communication. Stealth mechanisms aim to maintain a prolonged foothold on the target by concealing the malware, anti-analysis mechanisms detect the analysis environment to avoid detection and covert communication techniques enabled the malware to communicate stealthily with command and control servers. In this thesis, stealth mechanisms are categorized under anti-reverse engineering and anti-AV, while covert communication is included under anti-reverse engineering. This categorization reflects the view that uncovering command and control communication is a crucial aspect of reverse engineering, and that stealth techniques are primarily designed to evade antivirus detection and reverse engineering efforts. The anti-analysis techniques are similarly organised to the evasion types in the taxonomy of this thesis: anti-sandbox, anti-virtualization, anti-emulation and anti-debugging & anti-reverse engineering. In this thesis, anti-sandbox and anti-virtualization are combined in anti-sandbox / anti-VM, while anti-debugging & anti-reverse engineering are handled as separate evasion types, namely anti-debugging and anti-reverse engineering. The reasoning behind combining anti-sandbox and anti-VM is due to the significant overlap between

sandboxes and virtual machines, mostly because virtual machines are often used for sandbox purposes.

Evasion techniques are divided into transformation-based, concealment- based, attack-based, and combined strategies in the work of Geng et al [35]. In this work, transformation-based techniques - which are obfuscation-related - are classified under anti-AV and anti-reverse engineering, as they primarily target static analysis, which is a common method used by both antivirus software and security researchers (See Sections 2.1.4 and 2.2.1). Concealment-based techniques aim to evade detection by analyzing the environment and attack-based techniques focus on exploiting vulnerabilities or attacking detection mechanisms. Both are dependent on the environment, accordingly, these techniques are categorized separately within the provided taxonomy in this thesis, depending on the specific environments they target. Although Geng et al. note the concept of combined strategies, it is not treated as a distinct category here; instead, the provided taxonomy allows for the classification of malware samples under multiple evasion techniques. Galloro et al. presented a taxonomy as a result of a comparative analysis of previous taxonomies [34]. Notable about their taxonomy is the fact that it groups anti-VM and anti-sandbox together as anti-VM, and anti-debugging and anti-instrumentation as anti-debugging. The grouping for anti-sandbox and anti-VM is also present in the taxonomy of this work, as mentioned in the comparison for the work of Sharma et al. However, as debuggers and instrumentation tools are very different in their functioning, this taxonomy opted for separate evasion types for these.

5.2.2 Evasion Detection

The approach taken in [43] consists of a transparent bare-metal system with no in-guest analysis component, one emulation-based system, and two virtualization-based analysis systems. The most prominent component of the system, called Barecloud, is its transparent bare-metal system. By combining different analysis systems, the malware can be tracked very thoroughly, as even though it may be able to evade one of the analysis systems it can be tracked by another. This system requires significant setup, namely installing and configuring the four analysis systems used. The primary goal of this work was to develop, in light of static-analysis techniques being easily evaded, a robust analysis system that was able to perform better evasion detection than systems using these static-analysis based approaches. The focus was less on being able to identify many evasion techniques with the system, which was also evident from the dataset the authors provided including a relatively small number of five categories of evasion techniques. In contrast, by using CAPE, only one system needs to be setup in practice and in combination with the graph-database is able to classify 52 distinct techniques, while being easily extensible.

In the work of Polino et al., ARANCINO is introduced [53]. ARANCINO is a framework built on top of the dynamic instrumentation tool Intel Pin that enhances its stealth through bypassing anti-instrumentation techniques. Their detection tool is able to detect 9 techniques in total, that are classified in 4 types of anti-instrumentation evasion types: code cache artifacts, environment artifacts, just in

time compiler detection and overhead detection. This is a relatively small amount of techniques that only target one specific environment compared to the number of distinct techniques spread over different environments that the presented approach in this thesis handles.

Rather than creating a tool for evasion detection, Mills et al. created a dynamic sandbox reconfiguration tool. By rapidly reconfiguring and redeploying virtualized environments, various configurations can be tested for the triggering of evasive malware behavior. It encompasses five reconfiguration modes: Guest Additions Uninstalled, Trigger-Based Activation/Time Jump, Common Artefact Creation, Registry Edit, and Reverse Turing Test. These modes all represent a configuration in which a type of malware that tests for a relevant technique in detecting the environment can be mislead. For example, malware probing for the installation of Guest Additions[1] will be fooled by the Guest Additions Uninstalled environment, as this environment will not have Guest Additions installed on the virtual machine. This detection approach is only able to classify according to the system configuration, not to the more specific technique the malware is using. This is in contrast to the approach presented in this thesis, where detections happen at the lowest abstraction level - that of techniques - in order to allow more precise specification of the evasion techniques malware uses.

The detection approach in [37] is called PINvader and is able to instrument the target binary with function-level hooks, instruction-level hooks, and syscall-level hooks using Intel Pin. Although the work by Gozzini et al. focuses on evasion detection for benign samples, it is applicable to malware as well. While the approach boasts an impressive 113 technique detections and is able to bypass them as well, the use of Intel Pin has one limitation: it only works for Intel processors. In contrast, CAPE accommodates both Intel and AMD processors (with an AMD processor being used in the malware analysis environment in this thesis: See 4.2).

Kim et al. develop EvDetector in [42], a tool that also detects evasive malware by hooking APIs using Intel Pin. Different from the work by Gozzini et al., EvDetector uses a compiled list of APIs commonly employed for evasion used to check API calls against. Relevant API calls have hooking code inserted to trace the use of dynamically-allocated buffers of interest. By analyzing how these buffers influence conditional branches affecting the execution flow, such as is the case when evasive malware decides to execute its malicious behavior or not, the tool has a can have a certain guarantee that the API call had an influence in the execution flow and might have been intended for evasion. The advantage this approach has over the approach presented in this work it the evidence of the buffer serving as a filter for API techniques that are not able to influence the execution flow of the program, and thus are likely to be benign. The tool has some limitations as well: it only works for Intel processors (as it's using Intel Pin), it is not able to track API call chains with arguments passing between them, and it currently focuses on anti-sandbox and anti-VM techniques. The approach using CAPE in this thesis does work for

[1] A collection of device drivers and system programs that enable better performance and usability within a VirtualBox virtual machine.

AMD processors, is able to track API call chains with arguments passing between them using its signatures, and currently focuses on more than just anti-sandbox and anti-BM techniques, including anti-instrumentation, anti-emulation, anti-AV and anti-reverse engineering.

5.3 Threats to Validity

This section outlines the potential flaws and limitations that may affect the validity and integrity of this work. It serves as a cautionary note for readers to consider when interpreting, citing, or applying the results presented in this thesis.

5.3.1 Absence of Ground Truth

A significant challenge in this field is the absence of a definitive ground truth, which is also a key issue addressed by this work. However, this lack of ground truth poses a challenge for evaluating the results of the research itself. Creating a comprehensive ground truth manually for a large dataset is impractical, and relying on existing academic datasets - such as those provided by the authors of BareCloud - may not resolve this issue, as these datasets may also lack verified ground truths. This threat to validity implies that the database of evasive malware could be inaccurate, either by omitting certain evasion techniques or including techniques not used by the samples. Consequently, the quality of the approach is difficult to verify. A potential resolution involves corroborating the dataset through manual analysis, but this would require extensive man-hours and likely necessitate a collaborative effort from the research community.

5.3.2 Quality of the Signatures

The use of signatures, while beneficial in many respects, presents its own set of challenges. As discussed in the relevant section, a rule-based approach requires careful fine-tuning of signatures to ensure that they are neither too narrow nor too broad. Signatures that are too narrow may miss detections for certain techniques, as they may only capture specific implementations. Conversely, overly broad signatures can lead to numerous false positives, detecting patterns that do not accurately represent the intended technique. Achieving the right balance is crucial; therefore, the quality of the signatures represents a significant threat to validity of this work. If the signatures are of insufficient quality, the evasive malware dataset may contain inaccuracies as previously discussed, with no means to identify or correct these issues due to the lack of ground truth.

5.3.3 Dynamically Loaded API functions

A significant challenge in hooking functions with capemon's hooking engine involves dynamically loaded API functions. Unlike statically exported functions, dynamically loaded functions are not directly listed in the DLL's export table, making it more

difficult to obtain their addresses. While capemon's hooking engine does support hooking these functions, it requires additional effort. Helper functions such as `get_section_bounds`, `find_string_in_bounds`, and `find_first_lea_of_target` are available to assist with this task. Some dynamically loaded API functions have already been successfully hooked, such as `JsEval` from `Jscript.dll`.

In the undetected techniques, one notable example of a dynamically loaded API function is `ExecQuery` from `fastprox.dll`. This function executes the WMI queries and receives the unobfuscated query string, which is crucial for developing high-quality WMI signatures. However, due to time constraints and limited reverse engineering experience, `ExecQuery` could not be hooked.

While `ExecQuery` is the only dynamically loaded API function that needed to be hooked for the undetected techniques, its omission represents a potential threat to the validity of the WMI signatures. Consequently, WMI techniques used by malware may be underrepresented in the final results. Future work aimed at improving signature quality and addressing these dynamically loaded functions could help alleviate this issue, but it should be kept in mind when interpreting the results presented in this thesis.

Dataset Bias Towards Certain Techniques

Malware is often influenced by the time period in which it was developed, as highlighted by [42]. Consequently, the dataset used to construct the evasive malware database may exhibit a bias towards certain techniques. This potential bias could arise because the dataset might over-represent techniques prevalent in specific time periods while under-representing others. Although a considerable number of samples were analyzed to mitigate this issue, the dataset may still reflect a skewed distribution of techniques due to the lack of preselection aimed at achieving a balanced representation across different time frames. One possible way to mitigate this issue could be through ongoing addition of new samples over time, which may help to achieve a more balanced and representative dataset. This approach will be considered as part of future efforts to address this potential bias.

5.4 Future Work

5.4.1 Manual Verification of Samples

The absence of a ground truth poses a significant challenge to validating the results presented in this work. To address this, manual analysis of the samples in the database could provide a more thorough verification of the identified evasion techniques. By examining each sample and confirming the techniques used, the accuracy and reliability of the findings can be better assessed. However, this process is labor-intensive and would require a dedicated team of researchers or a substantial community effort, along with a considerable investment of time. Such manual verification is a complex task but is essential for enhancing the robustness of the results and ensuring the credibility of the dataset.

5.4.2 Improving Signatures

The quality of the signatures used for detection is a critical factor impacting the validity of the results. To enhance this aspect, future work should focus on several key areas. First, expanding the scope of the signatures by incorporating additional variations of techniques or refining existing ones for more precise detection will improve their effectiveness. An important part of this process is addressing dynamically loaded API functions, such as `ExecQuery` from `fastprox.dll`, which are crucial for accurate detection. Additionally, transitioning the signatures to an *evented* format - while the thesis currently employs the older signature format from CAPE - will significantly improve processing speed. This enhancement will become increasingly important as the number of signatures grows. By focusing on these areas, the detection system can be made more robust and efficient, thereby enhancing the overall quality and reliability of the analysis.

5.4.3 Integration in CAPE

Although scripts and modifiable components have been used to automate much of the process for creating and populating the graph database, full automation has not yet been achieved. Integrating the database directly into CAPE would offer users the ability to build their own collections of evasive malware categorized by the signatures they have employed. This integration would enable users to visually inspect their analysis results and enhance their understanding of the evasive techniques present. Alternatively, implementing a decentralized version of the database developed in this work could allow for a larger-scale, collaborative effort. This decentralized system would enable multiple contributors to build and maintain a comprehensive evasive malware database, fostering broader access and contributions from the malware analysis community. By leveraging a collaborative approach, this system could expand the database's coverage and enhance the collective understanding and detection of evasive malware techniques.

5.4.4 Expanding the Dataset

As indicated in the evaluation of the new signatures, one possible reason for many of the signatures not triggering at all is due to dataset bias towards certain techniques. This is a very real possibility, as [42] has shown that malware authors drop techniques that are ineffective over time, meaning the techniques used in evasive malware are dependent on the time period the malware was written in. This problem can be avoided by incorporating more samples from distinctly different time periods into the dataset. The selection and addition of such samples is left up to future work. The expectation is that the dataset can be improved over time, increasing the number of distinct evasion techniques contained in it, as well as improving its overall usefulness to security researchers.

Furthermore, additional information can be added to the properties of malware vertices. For instance, the particular family of the malware, its capabilities (logging keystrokes, accessing camera, steal browser credentials, etc.). Both examples lend

itself well to the use of CAPE as an analysis tool, as they can both be detected using signatures and YARA-rules using CAPE.

5.4.5 Graphical User Interface for the Database

Although AGE Viewer is a useful open-source tool for visualizing the database, the user experience could be significantly enhanced with a custom solution. The Cypher query language used by Apache AGE can be quite verbose, requiring users to write complex queries for searching or modifying database items. Developing a graphical user interface (GUI) that abstracts away the complexity of the Cypher query language would enhance the database's accessibility and user-friendliness, making it easier and more efficient for a broader audience to use through automated query capabilities and a more intuitive interaction model.

5.4.6 Evasive Malware Community Detection

One common application of graphs is community detection. In community detection, the vertices of a graph are grouped in clusters with the goal of finding clusters with more internal than external connections. To illustrate, this is commonly used by social media applications to map out social groups (the clusters in question) in order to more effectively advertise and predict behavioral patterns. The process of community detection can reveal very interesting information about a graph. In the context of evasive malware it could be used to find certain dynamics within the graph, malware samples that were particularly influential, a view of the processes that shape the graph and patterns that were not visual before. In combination with adding malware families to the properties of malware vertices, this could reveal interesting information about the relations between these malware families, such as the most influential samples in these families in terms of evasion techniques, if some of the samples potentially get inspiration for new evasion techniques from different malware families, etc.

Chapter 6

Conclusion

Starting with a specific focus on identifying and addressing the shortcomings of existing analysis tools, particularly CAPEv2, this thesis explored the evolving landscape of evasive malware detection. Through a combination of literature review, development of new detection mechanisms, and experimental validation, this work has provided a practical methodology for improving detection capabilities within modern malware analysis environments.

By applying the practical methodology, detection gaps were identified in CAPEv2 and new signatures were developed to fill these gaps. Another key contribution of this thesis is the development and application of an up-to-date taxonomy of evasion techniques. The results from the experimental analysis indicate that while these new signatures show promise, their performance varies depending on the specific evasion technique, illustrating the complexity of creating new detections and incorporating the right malware samples from the evolving malware landscape.

The new signatures, as well as CAPE's already existing ones, were mapped to the taxonomy. This mapping allowed to automatically populate a database of evasive malware, based on the analysis reports provided by CAPE. The use of a graph database for the purpose of cataloging a set of evasive malware according to their evasion techniques represents another significant contribution of this thesis. In handling the relationships between malware samples, their position in the taxonomy of evasion techniques, and the signatures used for detection, the flexibility of the graph database has proven valuable by allowing users to write queries ranging over different abstraction levels of the inherently modeled taxonomy. However, the limitations of the graph database for the presented use case, such as scalability challenges and the verbosity of the query language, suggest areas for future research and development.

Overall, this thesis has advanced the field of dynamic malware analysis by providing a concrete means of improving detection tools and by providing a framework for future work. The methodologies and findings presented here lay the groundwork for continued research into more robust malware detection strategies, which will be necessary as threats continue to evolve. Future work may involve further refinement of signatures, expanding the dataset of malware samples and corroborating the techniques within them, and integrating the evasive database within CAPE.

Appendices

Appendix A

Undetected Techniques Source Code

A.1 Pirated Windows

```
/*
Checks whether the specified application is a genuine Windows installation.

*/

#define WINDOWS_SLID                                                    \
            { 0x55c92734,                                               \
              0xd682,                                                   \
              0x4d71,                                                   \
              { 0x98, 0x3e, 0xd6, 0xec, 0x3f, 0x16, 0x05, 0x9f }        \
            }

BOOL pirated_windows()
{
        CONST SLID AppId = WINDOWS_SLID;
        SL_GENUINE_STATE GenuineState;
        HRESULT hResult;

        hResult = SLIsGenuineLocal(&AppId, &GenuineState, NULL);

        if (hResult == S_OK) {
                if (GenuineState != SL_GEN_STATE_IS_GENUINE) {
                        return TRUE;
                }
        }
        return FALSE;
}
```

Appendix B

Capemon Log API

The Log API (LOQ) takes a format string and accordingly parses the extra arguments.

B.1 Format Specifiers

The following format specifiers are available for the Log API:

- s → (`char *`) → zero-terminated string
- S → (`int`, `char *`) → string with length
- f → (`char *`) → zero-terminated ascii filename string (to be normalized)
- F → (`wchar_t *`) zero-terminated unicode filename string (to be normalized)
- u → (`wchar_t *`) → zero-terminated unicode string
- U → (`int`, `wchar_t *`) → unicode string with length
- b → (`int`, `void *`) → memory with a given size (alias for S)
- B → (`int *`, `void *`) → memory with a given size (value at integer)
- i → (`int`) → integer
- l → (`long`) → long integer
- L → (`long *`) → pointer to a long integer
- p → (`void *`) → pointer (alias for l)
- P → (`void **`) → pointer to a handle (alias for L)
- o → (`UNICODE_STRING *`) → unicode string
- O → (`OBJECT_ATTRIBUTES *`) → wrapper around a unicode string for filenames

- K → (`OBJECT_ATTRIBUTES *`) → wrapper around a unicode string for registry keys
- a → (`int`, `char **`) → array of string
- A → (`int`, `wchar_t **`) → array of unicode strings
- r → (`Type`, `int`, `char *`) type as defined for Registry operations
- R → (`Type`, `int`, `wchar_t *`) type as defined for Registry operations
- type r is for ascii functions, R for unicode (Nt* are unicode)
- e → (`HKEY`, `char *`) → key/ascii key/value pair (to be normalized)
- E → (`HKEY`, `wchar_t *`) → key/unicode key/value pair (to be normalized)
- k → (`HKEY`, `UNICODE_STRING *`) → unicode string for registry values (to be normalized)
- v → (`HKEY`, `char *`) → key/ascii value pair for registry values (to be normalized)
- V → (`HKEY`, `wchar_t`) → key/ascii value pair for registry values (to be normalized)

These can be found in capemon's github repository [9].

Appendix C

Control Set Evaluation Results

(a) NtYieldExecution

(b) ParentProcess (1)

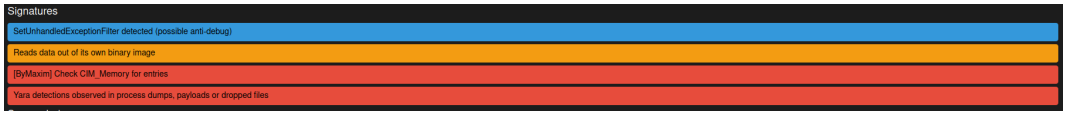
(c) ProcessJob

(d) MemoryWalk_GMI

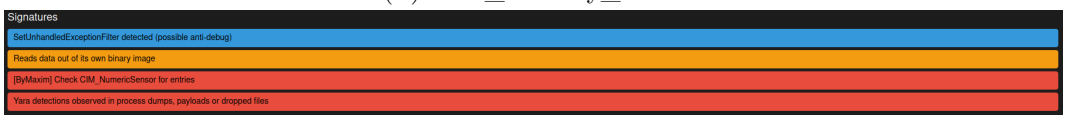
(e) MemoryWalk_Hidden

(f) WriteWatch_IsDebuggerPresent

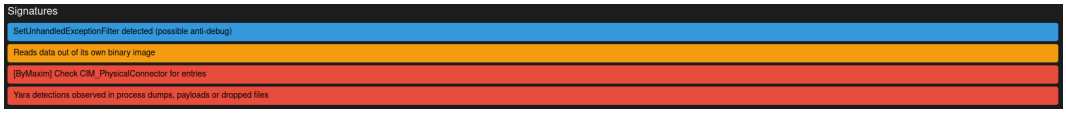
(g) CacheMemory_WMI



(h) CIM_Memory_WMI

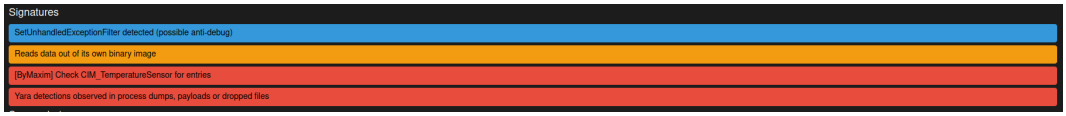


(i) CIM_Numeric_Sensor



(A) CIM_PhysicalConnector_WMI

(B) CIM_Sensor_WMI

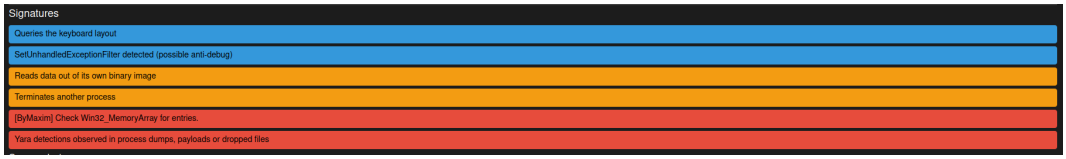


(C) CIM_TemperatureSensor_WMI

(D) CIM_VoltageSensor_WMI

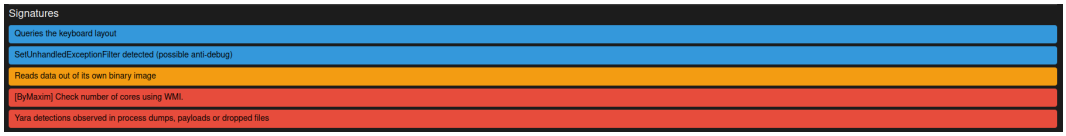
(E) CPU_Fan_WMI

(F) Current_Temperature_ACPI_WMI

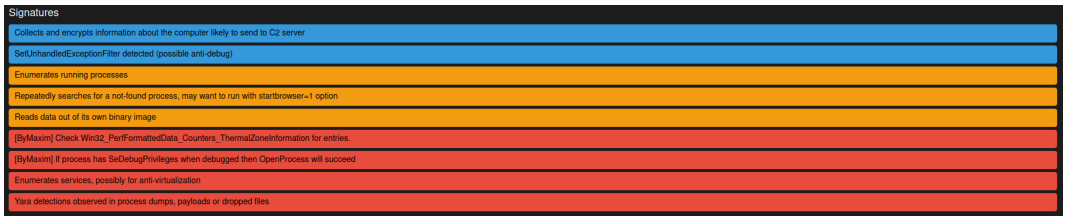


(G) MemoryArray_WMI

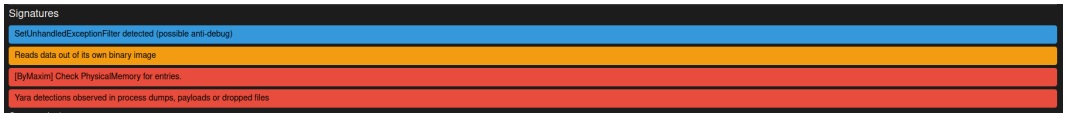
(H) MemoryDevice_WMI



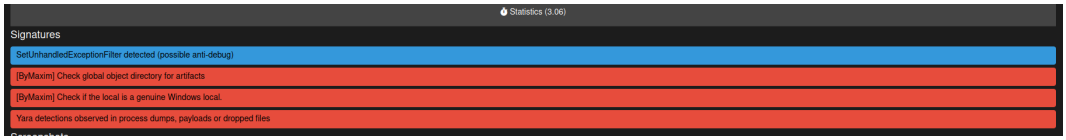
(I) NumberOfCores_WMI



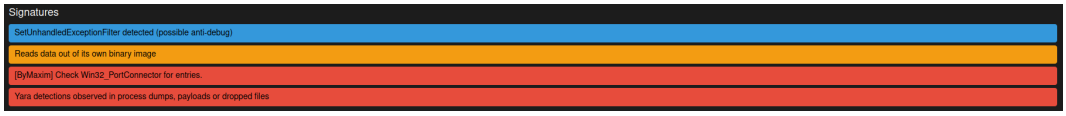
(A) Perfctr_ThermalZoneInfo_WMI



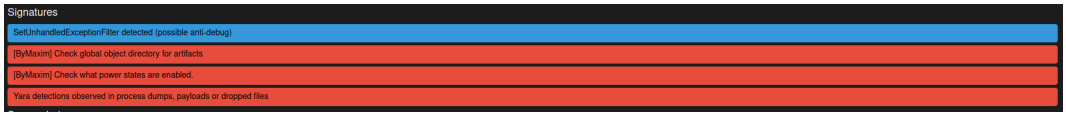
(B) PhysicalMemory_WMI



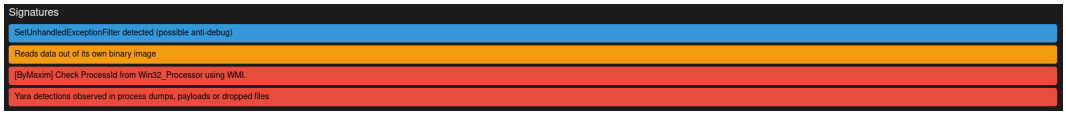
(C) PiratedWindows



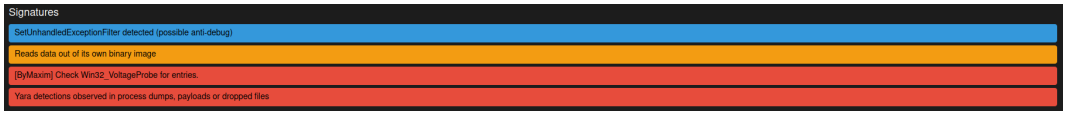
(D) PortConnector_WMI



(E) Power_Capabilities



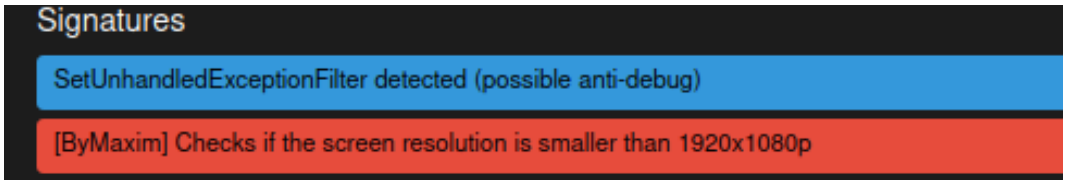
(F) Process_Id_Processor_WMI



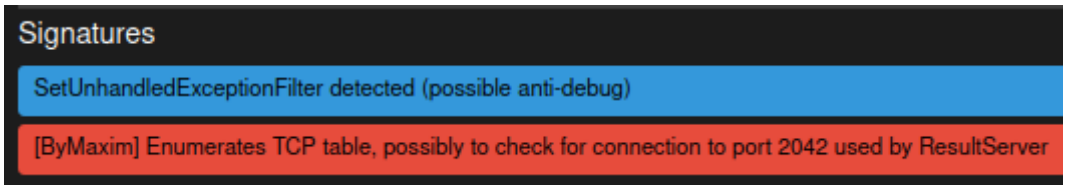
(G) VoltageProbe_WMI

(H) HyperVGlobal

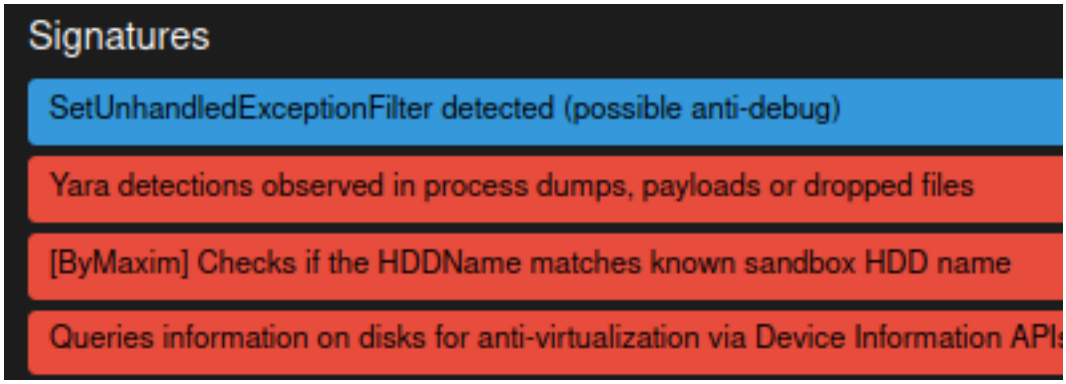
(I) QemuACPI



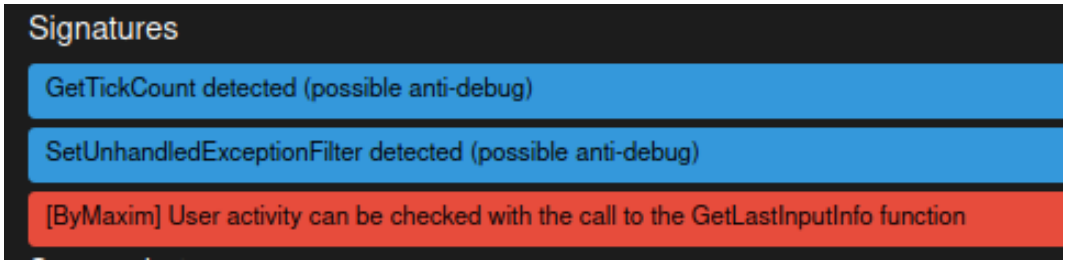
(A) ScreenResolution



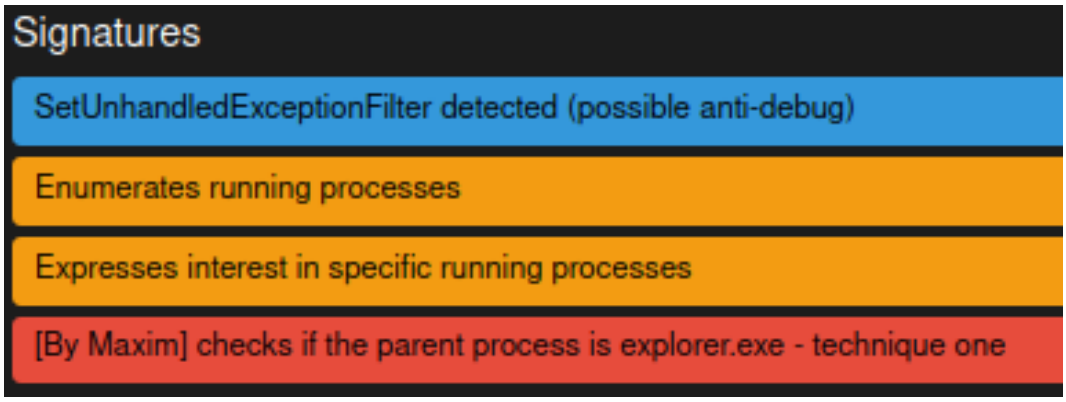
(B) CuckooTCP



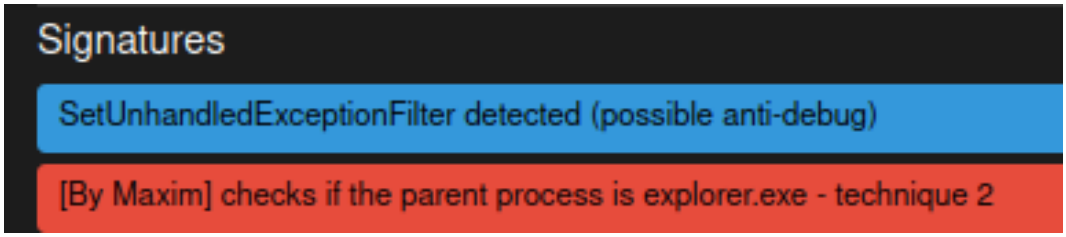
(C) HDDName



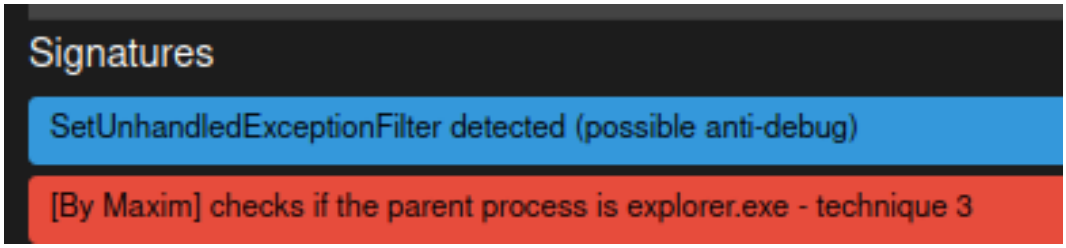
(D) GetLastInputInfo



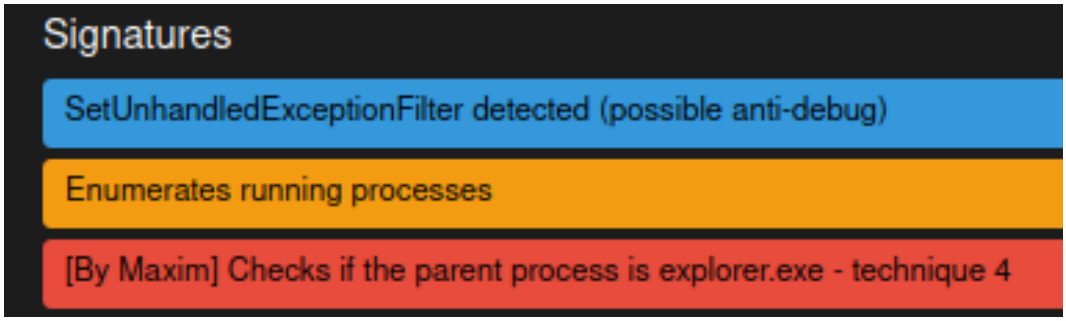
(A) ParentProcess (2)



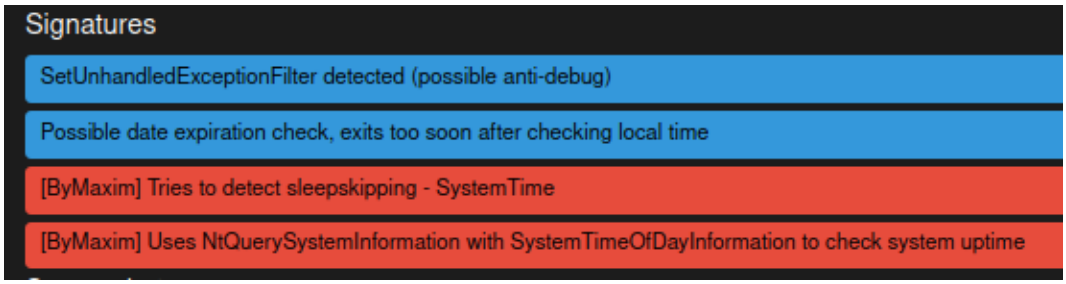
(B) ParentProcess (3)



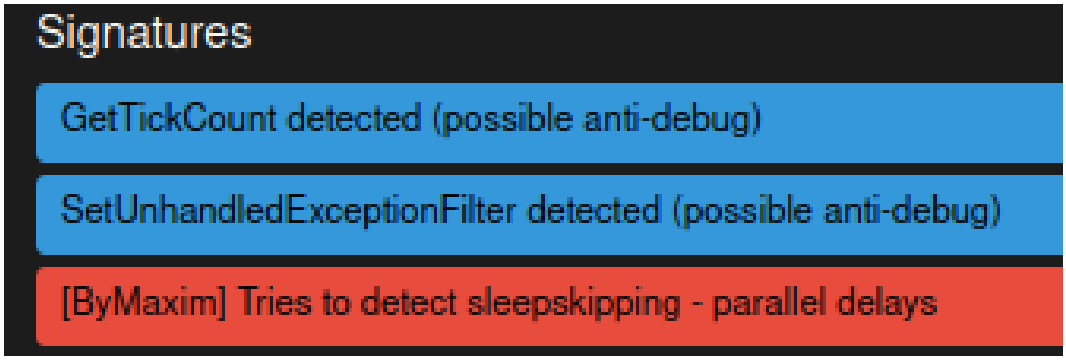
(C) ParentProcess (4)



(D) ParentProcess (5)



(A) SleepSkippingDetection (SystemTime)



(B) SleepSkippingDetection (ParallelDelays)

Appendix D

Existing CAPE Signatures Trigger Rate

Signature	count	]0, 10[	[10, 20[	[20, 30[	[30, 40[	[40, 50[	[50, 60[	[60, 70[	[70, 80[	[80, 90[	[90, 100[	[90, 100[
queries_keyboard_layout	649							X				
antidebug_setunhandledexceptionfilter	523						X					
reads_self	492					X						
antivm_generic_system	280			X								
antidebug_gettickcount	265			X								
enumerates_running_processes	225			X								
antivm_checks_available_memory	204			X								
dll_load_uncommon_file_types	105		X									
deletes_executed_files	101		X									
antivm_network_adapters	70	X										
antisandbox_unhook	65	X										
antidebug_guardpages	62	X										
antisandbox_sleep	57	X										
recon_fingerprint	44	X										
recon_programs	42	X										
accesses_recyclebin	40	X										
antivm_generic_disk	31	X										
antiav_detectreg	19	X										
antiemu_wine_reg	19	X										
removes_zoneid_ads	19	X										
antivm_generic_diskreg	16	X										
stealth_hiddenreg	15	X										
antivm_vbox_keys	13	X										
antidebug_ntsetinformationthread	12	X										
antisandbox_sboxie_libs	11	X										
antivm_generic_bios	11	X										
mimics_agent	11	X										
antidebug_windows	10	X										
disables_uac	10	X										
injection_createremotethread	10	X										

Table D.1: Existing CAPE Signatures for General Distribution Set of Malware (Part 1)

Signature	count	]0, 10[	[10, 20[	[20, 30[	[30, 40[	[40, 50[	[50, 60[	[60, 70[	[70, 80[	[80, 90[	[90, 100[	[90, 100[
injection_runpe	10	X										
stealth_hidden_extension	9	X										
antidebug_devices	8	X										
creates_largekey	8	X										
recon_systeminfo	7	X										
antiav_avast_libs	5	X										
antiav_detectfile	5	X										
antisandbox_mouse_hook	5	X										
persistence_ifeo	5	X										
antisandbox_suspend	4	X										
antivm_vbox_devices	4	X										
disables_windows_defender	4	X										
spoofs_procname	4	X										
antivm_vmware_devices	3	X										
masquerade_process_name	3	X										
antianalysis_detectfile	2	X										
antiav_servicestop	2	X										
antisandbox_sboxie_mutex	2	X										
antisandbox_sunbelt_libs	2	X										
antivm_generic_services	2	X										
antivm_vmware_keys	2	X										
antivm_vpc_mutex	2	X										
dotnet_code_compile	2	X										
windows_defender_powershell	2	X										
modify_security_center_warnings	2	X										
modify_uac_prompt	2	X										
powershell_command_suspicious	2	X										
antianalysis_detectreg	1	X										
antiav_bitdefender_libs	1	X										

TABLE D.2: Existing CAPE Signatures for General Distribution Set of Malware (Part 2)

Signature	**count**	]0, 10[	[10, 20[	[20, 30[	[30, 40[	[40, 50[	[50, 60[	[60, 70[	[70, 80[	[80, 90[	[90, 100[	[90, 100[
antivim_generic_disk_setupapi	1	X										
antivm_vbox_files	1	X										
antivm_vbox_libs	1	X										
clears_logs	1	X										
creates_nullvalue	1	X										
disables_windowsupdate	1	X										
network_fake_useragent	1	X										
network_tor	1	X										
powershell_renamed	1	X										
cmdline_process_discovery	1	X										
user_enum	1	X										

Table D.3: Existing CAPE Signatures for General Distribution Set of Malware (Part 3)

Bibliography

[1] 50 years of malware? Not really. 50 years of computer worms? That's a different story... - SANS Internet Storm Center — isc.sans.edu. https://isc.sans.edu/diary/50+years+of+malware+Not+really+50+years+of+computer+worms+Thats+a+different+story/27208/. [Accessed 20-05-2024].

[2] Antivirus | Download Antivirus | Antivirus Software — malwarebytes.com. https://www.malwarebytes.com/antivirus. [Accessed 20-05-2024].

[3] CAPE Sandbox Book. https://capev2.readthedocs.io/en/latest/. [Accessed 20-05-2024].

[4] Common malware persistence mechanisms | Infosec — infosecinstitute.com. https://www.infosecinstitute.com/resources/malware-analysis/common-malware-persistence-mechanisms/. [Accessed 20-05-2024].

[5] Creeper & Reaper — corewar.co.uk. https://corewar.co.uk/creeper.htm. [Accessed 20-05-2024].

[6] Cybersecurity threats and incidents differ by region — www2.deloitte.com. https://www2.deloitte.com/us/en/insights/topics/cyber-risk/global-cybersecurity-threat-trends.html. [Accessed 03-08-2024].

[7] Desktop Windows Version Market Share Worldwide | Statcounter Global Stats — gs.statcounter.com. https://gs.statcounter.com/windows-version-market-share/desktop/worldwide/#monthly-201701-202407. [Accessed 09-08-2024].

[8] Evasion techniques. https://evasions.checkpoint.com/. [Accessed 03-08-2024].

[9] GitHub - kevoreilly/capemon: capemon: CAPE's monitor — github.com. https://github.com/kevoreilly/capemon. [Accessed 20-05-2024].

[10] GitHub - kevoreilly/CAPEv2: Malware Configuration And Payload Extraction — github.com. https://github.com/kevoreilly/CAPEv2. [Accessed 20-05-2024].

[11] Goldoson: Privacy-invasive and Clicker Android Adware found in popular apps in South Korea | McAfee — mcafee.com. https://www.mcafee.com/blogs/other-blogs/mcafee-labs/goldoson-privacy-invasive-and-clicker-android-adware-found-in-popular-apps-in-s [Accessed 20-05-2024].

[12] Google ads push BumbleBee malware used by ransomware gangs — bleepingcomputer.com. https://www.bleepingcomputer.com/news/security/google-ads-push-bumblebee-malware-used-by-ransomware-gangs/. [Accessed 20-05-2024].

[13] Hatching - Automated malware analysis solutions — hatching.io. https://hatching.io/blog/cuckoo-sandbox-architecture/. [Accessed 20-05-2024].

[14] Malware Analysis: Steps & Examples - CrowdStrike — crowdstrike.com. https://www.crowdstrike.com/cybersecurity-101/malware/malware-analysis/. [Accessed 07-08-2024].

[15] Malware Configuration Parsers: An Essential Hunting Tool - Kraven Security — kravensecurity.com. https://kravensecurity.com/malware-configuration-parsers/. [Accessed 20-05-2024].

[16] obfuscation, n. https://www.oed.com/dictionary/obfuscation_n. [Accessed 2024-05-20].

[17] Parent Process Detection - Unprotect Project — unprotect.it. https://unprotect.it/technique/parent-process-detection/. [Accessed 04-08-2024].

[18] SolarWinds hack explained: Everything you need to know — techtarget.com. https://www.techtarget.com/whatis/feature/SolarWinds-hack-explained-Everything-you-need-to-know. [Accessed 20-05-2024].

[19] Top 10 Malware Q1 2024 — cisecurity.org. https://www.cisecurity.org/insights/blog/top-10-malware-q1-2024. [Accessed 20-05-2024].

[20] What is a malicious payload? | Cloudflare — cloudflare.com. https://www.cloudflare.com/learning/security/glossary/malicious-payload/. [Accessed 20-05-2024].

[21] What is Malware? | IBM — ibm.com. https://www.ibm.com/topics/malware. [Accessed 20-05-2024].

[22] What is Malware Builder? Understanding the Power of Malware Builders — cyberpedia.reasonlabs.com. https://cyberpedia.reasonlabs.com/EN/malware%20builder.html. [Accessed 20-05-2024].

[23] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste. Malware dynamic analysis evasion techniques: A survey. *ACM computing surveys*, 52(6):1–28, 2020.

[24] aid. Anti Debugging Protection Techniques with Examples — apriorit.com. https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software. [Accessed 04-08-2024].

[25] C. Bourne. Automating Malware Analysis using CAPE Sandbox - Endure Secure — endsec.au. https://endsec.au/blog/building-an-automated-malware-sandbox-using-cape/. [Accessed 08-08-2024].

[26] R. R. Branco, G. N. Barbosa, and P. Drimel. Scientific but not academical overview of malware anti-debugging , anti-disassembly and anti-vm technologies. 2012.

[27] A. Bulazel and B. Yener. A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium*, ROOTS, New York, NY, USA, 2017. Association for Computing Machinery.

[28] CAPE contributors. Community extensions for CAPEv2.

[29] Cert-Ee. cuckoo3: Cuckoo 3 is a python 3 open source automated malware analysis system.

[30] A. Chailytko and S. Skuratovich. Defeating sandbox evasion: how to increase the successful emulation rate in your virtual environment. In *ShmooCon 2017*, 2017.

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

[32] A. Damodaran, F. D. Troia, C. A. Visaggio, T. H. Austin, and M. Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, Dec. 2015.

[33] Echocti. echo-reports. https://github.com/echocti/ECHO-Reports/tree/main. [Accessed 03-08-2024].

[34] N. Galloro, M. Polino, M. Carminati, A. Continella, and S. Zanero. A systematical and longitudinal study of evasive behaviors in windows malware. *Computers & security*, 113:102550–, 2022.

[35] J. Geng, J. Wang, Z. Fang, Y. Zhou, D. Wu, and W. Ge. A survey of strategy-driven evasion methods for pe malware: Transformation, concealment, and attack. *Computers & security*, 137:103595–, 2024.

[36] gewarren. Introduction to .NET - .NET — learn.microsoft.com. https://learn.microsoft.com/en-us/dotnet/core/introduction?WT.mc_id=dotnet-35129-website. [Accessed 20-05-2024].

[37] S. Gozzini. Pinvader: a dynamic analysis tool for evasive techniques detection and bypass in 64-bit windows binaries. 2022.

[38] GrantMeStrength. Sl_genuine_state (slpublic.h) - win32 apps. https://learn.microsoft.com/en-us/windows/win32/api/slpublic/ne-slpublic-sl_genuine_state, Feb. 2024. [Accessed 03-08-2024].

[39] GrantMeStrength. Slisgenuinelocal function (slpublic.h) - win32 apps. https://learn.microsoft.com/en-us/windows/win32/api/slpublic/nf-slpublic-slisgenuinelocal, Feb. 2024. [Accessed 03-08-2024].

[40] Intezer. What MITRE D3FEND™ Techniques Does Intezer Analyze Implement? https://intezer.com/blog/malware-analysis/how-intezer-analyze-implements-mitre-d3fend-techniques/. [Accessed 08-08-2024].

[41] A. Khalimov, S. Benahmed, R. Hussain, S. Kazmi, A. Oracevic, F. Hussain, F. Ahmad, and C. Kerrache. Container-based sandboxes for malware analysis: A compromise worth considering. pages 219–227, 12 2019.

[42] M. Kim, H. Cho, and J. H. Yi. Large-scale analysis on anti-analysis techniques in real-world malware. *IEEE access*, 10:75802–75815, 2022.

[43] D. Kirat, G. Vigna, and C. Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In *PROCEEDINGS OF THE 23RD USENIX SECURITY SYMPOSIUM*, pages 287–301, SUITE 215, 2560 NINTH ST, BERKELEY, CA 94710 USA, 2014. USENIX Assoc; Google; NSF; Microsoft Res; Cryptog Res; Facebook; Qualcomm; IBM Res, USENIX ASSOC. 23rd USENIX Security Symposium, San Diego, CA, AUG 20-22, 2014.

[44] J.-P. Lesueur and T. Roccia. Unprotect project. https://unprotect.it/. [Accessed 03-08-2024].

[45] LordNoteworthy. Al-khaser.

[46] A. Mills and P. Legg. Investigating anti-evasion malware triggers using automated sandbox reconfiguration techniques. *Journal of cybersecurity and privacy*, 1(1):19–39, 2021.

[47] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, 2007.

[48] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach. Dynamic malware analysis in the modern era-a state of the art survey. *ACM computing surveys*, 52(5):1–48, 2019.

[49] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Computing Surveys*, 52:1–48, 09 2019.

[50] K. O'Reilly and A. Brukhovetskyy. CAPE: Malware Configuration And Payload Extraction.

[51] A. Ortega. Pafish.

[52] Y. Oyama. Trends of anti-analysis operations of malwares observed in api call logs. *Journal of Computer Virology and Hacking Techniques*, 14(1):69–85, 2018.

[53] M. Polino, A. Continella, S. Mariani, S. D'Alessio, L. Fontana, F. Gritti, and S. Zanero. Measuring and defeating anti-instrumentation-equipped malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 10327 of *Lecture Notes in Computer Science*, pages 73–96. Springer International Publishing, Cham, 2017.

[54] A. Sharma, B. B. Gupta, A. K. Singh, and V. Saraswat. Orchestration of apt malware evasive manoeuvers employed for eluding anti-virus and sandbox defense. *Computers & security*, 115:102627–, 2022.

[55] R. Sihwail, K. Omar, and K. A. Zainol Ariffin. A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis. 8:1662, 09 2018.

[56] S. Skuratovich. Invizzzible.

[57] Wikipedia contributors. 2022 costa rican ransomware attack — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=2022_Costa_Rican_ransomware_attack&oldid=1231413904, 2024. [Online; accessed 3-August-2024].

[58] Wikipedia contributors. Boolean satisfiability problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Boolean_satisfiability_problem&oldid=1219369085, 2024. [Online; accessed 20-May-2024].

[59] Wikipedia contributors. Computer worm — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Computer_worm&oldid=1222038341, 2024. [Online; accessed 20-May-2024].

[60] Wikipedia contributors. Creeper and reaper — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Creeper_and_Reaper&oldid=1210076917, 2024. [Online; accessed 20-May-2024].

[61] Wikipedia contributors. Hypervisor — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Hypervisor&oldid=1218948761, 2024. [Online; accessed 20-May-2024].

[62] Wikipedia contributors. Malware — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Malware&oldid=1223843083, 2024. [Online; accessed 20-May-2024].

[63] Wikipedia contributors. .net — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=.NET&oldid=1219304924, 2024. [Online; accessed 20-May-2024].

[64] Wikipedia contributors. Np-completeness — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=NP-completeness&oldid=1222934019, 2024. [Online; accessed 20-May-2024].

[65] Wikipedia contributors. Reverse engineering — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Reverse_engineering&oldid=1221145420, 2024. [Online; accessed 20-May-2024].

[66] Wikipedia contributors. Side-channel attack — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Side-channel_attack&oldid=1194717603, 2024. [Online; accessed 20-May-2024].

[67] Wikipedia contributors. Watering hole attack — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Watering_hole_attack&oldid=1221434286, 2024. [Online; accessed 20-May-2024].

[68] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 297–300, 2010.