Behavior Analysis for Vulnerability and Malware Detection

Razvan Raducu Teodorescu

Ph.D. Dissertation

Department of Computer Science and Systems Engineering

Universidad de Zaragoza

Advisors:
Prof. Dr. Pedro Álvarez
Prof. Dr. Ricardo J. Rodríguez

July 2025

Exploit code, not people.

"Yes, I am a criminal. **My crime is that of curiosity.** My crime is that of judging people by what they say and think, not what they look like."

LOYD «THE MENTOR» BLANKENSHIP
The Conscience of a Hacker (also known as *The Hacker Manifesto*)
Phrack, Volume One, Issue 7, Phile 3 of 10
8 January 1986

"The important thing is not to stop questioning. Curiosity has its own reason for existence. One cannot help but be in awe when he contemplates the mysteries of eternity, of life, of the marvelous structure of reality."

ALBERT EINSTEIN
Old Man's Advice to Youth: "Never Lose a Holy Curiosity"
LIFE Magazine, p. 64
2 May 1955

Acknowledgments

The process of developing a doctoral thesis often becomes an emotional roller coaster where you find yourself asking: *Is it really worth it?* Since this is the most personal section of this document, I would like to take this opportunity to thank all the people who have helped me get to this point.

I want to thank my parents for the tremendous efforts they have made so that I could become the first person in my close family to pursue higher education.

I want to thank my partner, Tatiana, who was drawn into this journey and affected by its ups and downs, but who has stayed by my side throughout it all.

I want to thank my friends (both those I have known for years and those I have met during my academic journey), who have supported me in one way or another.

I am also grateful to Dr. Apostolis Zarras for welcoming me during my stay, as well as to the reviewers who made every effort to help improve this document.

Finally, I want to thank my two supervisors, Pedro and Ricardo, for giving me freedom in several ways. I was allowed to explore different topics at my own pace, which has made this journey significantly better. I am especially grateful to them for not imposing strict schedules on me and allowing me to manage my own time. I also appreciate the opportunity they gave me to participate in several projects. Also, I would like to thank their patience when reviewing my work and their efforts to show what first-level research really involves to someone like me, who does not always have a clear understanding of what science *is* or *is not* according to modern publishing standards, and who understands even less about many and sometimes counterproductive practices of the current publish or perish system. We all know academia can be tough (particularly in environments where chasing publications takes priority over intellectual growth and meaningful progress, although this depends on individual's goals and standards) and, unfortunately, I have met many people that did not make it, abandoning and leaving behind all their work. Thanks to Pedro and Ricardo it was not my case.

I would also like to give special recognition to the cybersecurity community and all those who create high-quality content and share it openly in the spirit of free knowledge. Too often, I had the impression that these works are not credited enough, and, sometimes, even unfairly discredited, simply because they are not published in prestigious journals. This type of content (in any of its forms like videos, write-ups, books, reports, courses, or articles) makes knowledge accessible to everyone and for many of us out there, it has been and will be a primary source

of learning. After all, “if I have seen further it is by standing on the shoulders of Giants”.

This work may be open to criticism for many reasons, and perhaps one day I will look back at these lines and wonder: *What was my younger self thinking?* But what I will always be proud of is that this work has been honest. In the meantime, all I can do is strive to improve each day, hoping that when I revisit these words, the answer to the question at the beginning will be a convincing yes.

Abstract

The escalating frequency and sophistication of cyberattacks pose a growing threat to the security of information systems worldwide. Whether via vulnerability exploitation or covert malware operations, breaches evade traditional detection mechanisms, resulting in severe financial, operational, and privacy consequences. As adversaries evolve and develop increasingly sophisticated tactics, there is a pressing need for robust detection and mitigation systems. This dissertation addresses vulnerability assessment at the development stage and dynamic behavior analysis, focusing on the Windows operating system.

Initially, this work focuses on the Time-of-Check to Time-of-Use (TOCTOU) race condition vulnerability, by conducting a systematic literature review to categorize existing techniques to detect and also to exploit it. This study reveals that the vulnerability is unlikely to be removed and highlights the limitations of the proposed defenses at the time of this writing. Recognizing these limitations, our research pivots to dynamic analysis, focusing on the observation of program behavior during execution. To obtain the runtime behavior, programs can be executed in controlled environments that capture their actions and generate executions traces, which will be further analyzed.

In this sense, a comprehensive evaluation of modern malware sandboxes is conducted to identify optimal environments for dynamic behavior analysis. This leads to the development of MALVADA, a framework for generating enriched execution traces, resulting in the creation of the Windows Malware Execution Traces (WINMET) dataset. To analyze behavior patterns within those execution traces, we develop MALGRAPHIQ, a graph-based system that models and visually represents runtime behaviors, enabling pattern recognition and behavior classification. Additionally, this dissertation also defines the Windows Behavior Catalog (WBC), a structured repository that defines behaviors as sequences of Windows API and system calls. Our experimental evaluations demonstrate that MALGRAPHIQ effectively identifies behaviors exhibited by programs at runtime, reinforcing its potential for malware analysis.

Resumen

La creciente frecuencia y sofisticación de los ciberataques representan una amenaza cada vez mayor para la seguridad de los sistemas de información a nivel mundial. Ya sea mediante la explotación de vulnerabilidades o a través de operaciones encubiertas con *malware*, las brechas de seguridad evaden los mecanismos tradicionales de detección, lo que resulta en graves consecuencias financieras, operativas y para la privacidad. A medida que los adversarios evolucionan y desarrollan tácticas cada vez más complejas, surge la necesidad de sistemas robustos de detección y mitigación. Esta tesis aborda la evaluación de vulnerabilidades en la fase de desarrollo y el análisis dinámico de comportamientos, centrándose en el sistema operativo Windows.

Inicialmente, este trabajo se enfoca en la vulnerabilidad de condición de carrera conocida como *Time-of-Check to Time-of-Use* (TOCTOU), realizando una revisión sistemática de la literatura para categorizar las técnicas existentes para detectarla y también para explotarla. Este estudio revela que es poco probable que la vulnerabilidad sea eliminada y destaca las limitaciones de las defensas propuestas en el momento de escribir esta tesis. Reconociendo estas limitaciones, nuestra investigación se redirige hacia el análisis dinámico, centrándose en el estudio de comportamientos durante la ejecución de programas. Estos comportamientos se pueden obtener ejecutando los programas en entornos controlados que permitan capturar sus acciones y generar trazas de ejecución, que serán posteriormente analizadas.

En este sentido, se lleva a cabo una evaluación exhaustiva de los entornos modernos de análisis dinámico de *malware* (también llamados *sandboxes*, en inglés) para identificar los más óptimos para el análisis dinámico de comportamientos. Esto conduce al desarrollo de MALVADA, un entorno para generar trazas de ejecución enriquecidas, que da lugar a la creación del conjunto de datos «Windows Malware Execution Traces» (WINMET). Para analizar los patrones de comportamiento de esas trazas de ejecución, se ha desarrollado MALGRAPHIQ, un sistema basado en grafos que modela y representa visualmente los comportamientos que los programas manifestaron durante su ejecución, permitiendo el reconocimiento de patrones y la clasificación de dichos comportamientos. Además, esta tesis también define el «Windows Behavior Catalog» (WBC), un repositorio estructurado que define comportamientos como secuencias de llamadas al sistema y a la API de Windows. Nuestra experimentación demuestra que MALGRAPHIQ es capaz de identificar con éxito los comportamientos de programas en tiempo de ejecución, reforzando su potencial aplicación en el análisis de malware.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

System breaches are continually increasing in both frequency and sophistication. This ongoing threat to the confidentiality, integrity, and availability of data causes financial losses, data breaches, and operational disruptions worldwide, affecting organizations of all types and sizes, including public and private institutions, non-governmental organizations, and multinational corporations. Malicious behavior is a common factor in every breach, either through the exploitation of software vulnerabilities or the compromise of operating systems via covert execution of malign applications. This situation underscore the critical importance of defensive tools designed to detect and mitigate threats to system's security. Detection methods generally fall into two categories: *static analysis*, which examines code before execution (either at the development stage or after compilation) to identify patterns indicative of malicious activity or vulnerabilities, or *dynamic analysis*, which monitors program behavior during execution. Both approaches are essential in safeguarding systems and thwart increasingly sophisticated cyber threats.

1.1 Motivation and Research Questions

The proliferation of malicious software, or *malware*, continues to pose a significant threat to modern information systems. Cyberattacks leveraging malware have grown exponentially over the past decade [1, 2, 3], with global cyberattacks increasing 44% in 2024 alone [4]. This alarming trend stresses the need for effective and reliable defensive tools capable of detecting behavior patterns indicative of security breaches or intrusion attempts. As adversaries continue to innovate [5, 6, 7], there is an increasing demand for advanced approaches to identify and mitigate vulnerabilities and malicious behavior. This task is critical both at the source code level, using static analysis, and during program execution or post-mortem analysis, using dynamic methods.

The objective of this dissertation is to explore and develop methodologies to identify behavior patterns indicative of activities that could compromise system security. These behaviors may originate from vulnerabilities within source code or from malicious activities during execution. Detecting and analyzing vulnerabilities and malicious behaviors, whether during the development lifecycle or after an

application is deployed, is essential for reducing risks and maintaining trust in information systems.

Given the ongoing challenges posed by security threats and the increasing number and complexity of tactics employed by adversaries to compromise the confidentiality, integrity, and availability of systems, in this dissertation we addressed the following questions:

Research Question 1

To what extent the detection of behavior patterns through static analysis helps preventing the malicious activity?

Initially, we focus on improving security during the software development phase through source code analysis. Specifically, we study the file-based Time-of-Check to Time-of-Use (TOCTOU) vulnerability, a race condition vulnerability that originates when two functions (one that performs the check and another one that performs some action based on the results of that check) operate on the same filesystem object. Such sequences are a clear example of behavior patterns potentially detectable through static analysis. Despite being first identified in the mid-1970s [8, 9], this vulnerability remains unresolved and is actively exploited in the wild [10, 11]. To address this RQ, we examine the vulnerability in depth and conduct a systematic literature review of file-based TOCTOU attack and defense mechanisms proposed in academic research. As part of our investigation, we develop a taxonomy of existing attack and defense techniques. Further analysis reveals that most defense approaches are either non-reproducible or rely on unreliable filesystem information as a key component of their defense approach.

Our study reveals a significant limitation: even when the TOCTOU vulnerability is detected through static analysis tools, such as code linters (tools that help identify insecure coding patterns, potential vulnerabilities, and bugs), its identification only reduces the risk of exploitation, but does not eliminate it. Addressing this vulnerability requires highly trained developers with a strong background in secure coding practices, which are complex and prone to errors. Moreover, implementing a definitive solution to mitigate it would require different aproaches that pose significant backward compatibility challenges, rendering existing software potentially obsolete.

In light of these challenges, we adopt a broader approach. Rather than focusing on a specific vulnerability, we direct our efforts toward studying malicious behavior from the perspective of the operating system. Furthermore, source-code static analysis requires access to source code, which is usually unavailable, particularly when analyzing third-party software or malware. Given these limitations, we transition to dynamic analysis, aiming to analyze and observe malware or other binaries with unknown intent, ultimately detecting malicious behaviors exhibited during program execution.

The last part of this dissertation focuses on analyzing programs running on the Windows operating system, as it remains the most widely used desktop platform, holding approximately 73% of the market share as of December 2024 [12], and is

the primary target of malware attacks [3]. This transition leads to the formulation of the following research questions:

Research Question 2

How do modern dynamic analysis environments support capturing runtime behaviors of binaries in diverse scenarios?

Dynamic analysis captures runtime behaviors, including interactions with system resources, such as registry entries, sockets or files. To effectively perform dynamic analysis, it is essential to leverage tools capable of running programs in controlled environments while capturing detailed execution traces. In this regard, we focus on sandbox environments, as they provide isolated and monitored execution settings. Sandboxes are designed to emulate real-world user environments, allowing analysts to capture the manifested behaviors during the execution without endangering actual systems. Sandboxes capture execution traces, including Application Programming Interface (API) calls, system calls, and network activity, which are some of the inputs required for subsequent behavior analysis.

In order to identify the available sandbox environments and determine those best suited for different research scenarios, we conduct a comprehensive review of modern sandboxes. This evaluation focuses on their features, capabilities, and trade-offs. We examine open source and commercial solutions, exploring their effectiveness in diverse scenarios and their challenges. Each sandbox is assessed based on its support for automation, API availability, supported operating systems, ability to provide detailed behavioral insights, configurability, usability, privacy implications, and resilience against evasion techniques. These factors are essential when choosing a sandbox. Understanding the strengths and limitations of existing solutions not only facilitates informed tool selection, but also identifies opportunities for improving the effectiveness of dynamic analysis systems in malware research.

After evaluating the available tools for capturing runtime behaviors, CAPEv2 [13] emerges as the most suitable option for our needs due to its configurability, detailed reporting capabilities, and suitability for diverse malware analysis scenarios. Following the local deployment of the CAPE sandbox, we begin analyzing malware samples and generating our own dataset. Simultaneously, we analyze other collections of malware behavior information and assess their comprehensiveness. In this regard, we addressed the following research question:

Research Question 3

To what extent the available execution trace datasets (if any) are comprehensive and suitable for behavioral analysis?

While dynamic analysis tools like CAPEv2 excel at generating execution traces, the suitability of existing datasets for behavioral analysis remains a critical question. Installing and configuring a sandbox environment is a tedious task that not everyone can undertake due to constraints such as time, financial resources, or hardware availability. As a result, existing collections of execution traces can be

highly valuable for studying malware behavior. However, effective behavioral analysis requires datasets that are comprehensive, context-rich, and representative of diverse malware behaviors.

In this context, we analyze existing execution trace datasets while simultaneously develop our own. Unfortunately, our study reveals that many publicly available datasets lack relevant details. These datasets often prioritize efficiency for specific analysis techniques, such as artificial intelligence models, and simplify their content thus omitting critical contextual information, such as API parameters, return values, or resource interactions. This compromises their utility for in-depth behavioral studies.

To address this gap, we develop MALVADA, a framework designed to generate high-quality datasets of malware execution traces and, leveraging it, we create the Windows Malware Execution Traces (WINMET) dataset. We compare existing datasets with WINMET to evaluate their comprehensiveness, richness of context, and applicability to behavioral analysis. The development of MALVADA and WINMET lay the groundwork for in-depth behavioral analysis. However, capturing execution traces is only the first step. Understanding the behavior of malicious binaries requires developing methods to detect patterns indicative of malicious intent within these datasets. This leads to the last research question in this dissertation:

Research Question 4

To what extent are we able to detect malicious behavior patterns through dynamic analysis?

Dynamic analysis provides the foundation for identifying behaviors exhibited by binaries during execution, such as file modifications, network communications, or registry interactions. However, detecting malicious behavior patterns is not trivial, as modern malware employs sophisticated evasion techniques and exhibits behaviors that closely mimic benign software. To address these challenges, we develop MALGRAPHIQ, a graph-based system designed to generate visual representations of program behavior by analyzing execution traces generated in dynamic environments, thus simplifying their interpretation.

Simultaneously, we create the Windows Behavior Catalog (WBC), a structured repository of behavior patterns inspired by MITRE's Malware Behavior Catalog (MBC). The WBC defines specific behaviors in terms of API and system calls, enabling the systematic identification of actions performed by programs, whether benign or malicious. MALGRAPHIQ applies pattern-matching techniques to identify instances of specific behaviors by comparing patterns from the WBC against the execution traces. The generated visualizations allow users to quickly identify the actions performed by a program or set of programs during execution, facilitating the initial triage and analysis of unknown programs. Furthermore, we evaluate the potential of MALGRAPHIQ to uncover behavior patterns and help detecting malware in dynamic environments.

By addressing these questions, this dissertation explores the detection of vulner-

abilities through static analysis, evaluates the tools and datasets available for dynamic analysis, and develops mechanisms to detect behavior patterns during program execution.

1.2 Dissertation Outline

The remainder of this thesis is structured as follows. Chapter 2 provides the necessary background information. The subsequent chapters address each topic from this dissertation in detail. Chapter 3 presents our systematic literature review on the file-based TOCTOU vulnerability and examines the proposed defense mechanisms. Our study of available modern malware sandboxes is described in Chapter 4, describing the findings and their applicability to various scenarios. Chapter 5 details the technical aspects of MALVADA and introduces the WINMET dataset. The MALGRAPHIQ tool and the analysis of the visual representations it generates are described in Chapter 6. This chapter also includes a k-nearest neighbors (kNN) cross-validation performed on different malware families to validate MALGRAPHIQ. Each chapter provides the context of the research conducted, a review of related work, and concludes with a discussion of the findings. Finally, Chapter 7 concludes this dissertation, recapitulating the main results and discussing potential future research directions.

1.3 Scientific Contributions and Other Results

This section enumerates the results of this disseration.

Publications

Journal Articles

1. R. Raducu, R. J. Rodríguez, and P. Álvarez, "Defense and Attack Techniques Against File-Based TOCTOU Vulnerabilities: A Systematic Review," in IEEE Access, vol. 10, pp. 21742-21758, 2022, DOI: 10.1109/ACCESS.2022.3153064. Impact factor (JCR): 3.9 (3.6 without self citations), rank 73/158 (Q2; Computer Science, Information Systems). See Chapter 3.

2. R. Raducu, A. Villagrasa-Labrador, R. J. Rodríguez, and P. Álvarez, "MALVADA: A framework for generating datasets of malware execution traces," in SoftwareX, vol. 30, 2025, DOI: 10.1016/j.softx.2025.102082. Impact factor (JCR): 3.4 (3.3 without self citations), rank 38/108 (Q2; Computer Science, Software Engineering). See Chapter 5.

Under Review

1. R. Raducu, R. J. Rodríguez, and P. Álvarez, "A Graph-Based Dynamic Analysis System for Behavior Detection in Windows Applications". Currently *under review*, submitted to The Computer Journal. Impact factor (JCR): 1.5 (1.4

without self citations), rank 71/144 (Q2; Computer Science, Theory & Methods). See Chapter 6.

2. R. Raducu, A. Zarras, R. J. Rodríguez, and P. Álvarez, "The Sandbox Reloaded: A Guide to Modern Malware Sandbox". Currently *under review*, submitted to the 20th International Conference on Availability, Reliability and Security (ARES 2025). B rank (28.19% of 784 ranked venues) according to CORE2023. See Chapter 4.

Other Publications not qualifying for this dissertation

1. R. Raducu, R. J. Rodríguez, and P. Álvarez. "Resource Consumption Evaluation of C++ Cryptographic Libraries on Resource-Constrained Devices," in *Applied Cryptography in Computer and Communications (EAI AC3 2022)*. DOI: 10.1007/978-3-031-17081-2_5.

National Conferences

1. R. Raducu, R. J. Rodríguez, and P. Álvarez. "Towards a Web System for the Evaluation of Resource Consumption," in *Jornadas de Concurrencia y Sistemas Distribuidos* 2021 (JCSD 2021).
2. R. Raducu, R. J. Rodríguez, and P. Álvarez. "Model-Based Analysis of Race Condition Vulnerabilities in Source Code," in *Jornadas Nacionales de Investigación en Ciberseguridad* 2022 (JNIC'22)
3. R. Raducu, R. J. Rodríguez, and P. Álvarez. "A Review of Defense and Attack Techniques Against File-Based TOCTOU Vulnerabilities: A systematic Review," in *Jornadas Nacionales de Investigación en Ciberseguridad* 2023 (JNIC'23)

International Conferences

1. R. Raducu, R. J. Rodríguez, P. Álvarez, and A. Zarras. "Malware Sandbox Comparison (Work in Progress)," in International Conference on Digital Forensic Analysis and Exploitation 2025 (DFex 2025)

Research Internships

During this thesis, I completed two research internships, for a total of 3 months:

1. September 2023 to October 2023 (1 month) at the Systems Security Laboratory from the Department of Digital Systems, University of Piraeus, Greece. Supervised by Dr. Apostolis Zarras.
2. September 2024 to November 2024 (2 months) at the Systems Security Laboratory from the Department of Digital Systems, University of Piraeus, Greece. Supervised by Dr. Apostolis Zarras.

The Systems Security Laboratory's main objective is to conduct research, development and educational activities in various areas of systems security such as

development of secure information systems, intrusion detection systems, or web applications security. Dr. Apostolis Zarras's research trajectory aligns with this doctoral thesis, particularly in the field of malware detection and classification.

Teaching Assistance

Over the course of this thesis, I contributed to various teaching activities:

- Bachelor's Degree in Industrial Engineering Technology. Teaching assistant during the laboratory sessions of Fundamentals of Computer Science (course code: 30007). Academic years 2022/2023, 2023/2024 and 2024/2025.
- Bachelor's Degree Joint Program of Computer Science and Mathematics. Teaching assistant during the laboratory sessions of Programming I (course code: 39504). Academic year 2023/2024.
- Bachelor's Degree in Computer Science. Teaching assistant during the laboratory sessions of Programming I (course code: 30204). Academic years 2022/2023 and 2023/2024.
- Master's Degree in Computer Science. Teaching assistant during the laboratory sessions of Exploiting Software Vulnerabilities (course code: 62240). Academic years 2022/2023, 2023/2024 and 2024/2025.

Supervised Students

During this thesis, I co-supervised two undergraduate theses:

- Julia Varea Palacios. B. S. in Computer Science, February 2024. *Malware detection using machine learning techniques.* Co-supervised with Prof. Pedro Álvarez. Analyzed the accuracy of various machine learning algorithms and n-grams in detecting and classifying malware based on execution traces.
- Salomé Rea Ávila. B. S. in Computer Science, December 2024. *Detection of vulnerabilities in source code through Petri nets.* Co-supervised with Prof. Ricardo J. Rodríguez. Developed a tool that models source code as a Petri net to analyze and identify vulnerabilities.

Academic Service

Throughout this thesis, I served as a reviewer for the following academic venues:

- IEEE 26th International Conference on Emerging Technologies and Factory Automation (ETFA 2021).
- The Computer Journal, 2025.
- *Jornadas Nacionales de Investigación en Ciberseguridad* 2025 (JNIC'25).

Additionally, I was part of the following Committees:

- Organizing Committee for the Digital Forensics Research Conference Europe (DFRWS EU 2024) as a Rodeo (Capture The Flag competition) Chair.

- Technical Program Committee member for the *Jornadas Nacionales de Investigación en Ciberseguridad* 2025 (JNIC'25).

Research Projects

Throughout the course of this thesis, I actively participated in various research projects:

- *IT Consulting* (ref. 2021/0355). Funding entity: TLM Logistic Zaragoza 2014, S. L. Funds: 2,089.67€. Principal Investigator: Prof. Ricardo J. Rodríguez. From July 1, 2021 to July 31, 2021.
- *T21_20R: Distributed Computing Group (DisCo)* (ref. T21_20R) Funding entity: Aragon Government. Funds: 25,713€. Principal Investigator: Prof. Pedro Álvarez. From February 15, 2021 to December 31, 2022.
- *Malware Indicators of Compromise Enhanced by Memory Forensic Analysis (MIMFA)* (ref. TED2021-131115A-I00). Funding entity: Ministry of Science, Innovation and Universities of Spain and the European Union. Funds: 129,835.00€. Principal Investigator: Prof. Ricardo J. Rodríguez. From December 1, 2022 to November 30, 2023.
- *T21_23R: Distributed Computing Group (DisCo)* (ref. T21_23R) Funding entity: Aragon Government. Funds: 25,713€. Principal Investigators: Prof. Pedro Álvarez and Prof. Javier Fabra. From January 1, 2023 to December 31, 2025.
- *Technology Foresight in Industrial Cybersecurity* (ref. C082/2023_2). Funding entity: University of Zaragoza. Funds: 6,500€. Principal Investigator: Prof. Ricardo J. Rodríguez. From October 30, 2023 to March 31, 2024.
- *EINA UNIZAR Cybersecurity Strategic Project* (ref. 2023/2038). Funding entity: Spanish National Cybersecurity Institute (INCIBE). Funds: 648,793.79€. Principal Investigator: Prof. Ricardo J. Rodríguez. From October 30, 2023 to December 31, 2024.
- *Cybersecurity Culture Promotion (CyberCamp events)* (ref. 2023/2040). Funding entity: Spanish National Cybersecurity Institute (INCIBE). Funds: 144,576.75€. Principal Investigator: Prof. Ricardo J. Rodríguez. From November 7, 2023 to December 31, 2024.

Teaching Innovation Projects

I was also invovled in two teaching innovation projects:

- *CTFs School: learning cybersecurity through educational gamification* (ref. PRAUZ_287 - 722). Funding entity: University of Zaragoza. Funds: 300€. Principal Investigator: Prof. Ricardo J. Rodríguez. Academic year 2022/2023.
- *Learning Vulnerability Exploitation through Educational Gamification* (ref. PIIDUZ_1 - 5416). Funding entity: University of Zaragoza. Funds: 300€. Principal Investigator: Prof. Ricardo J. Rodríguez. Academic year 2024/2025.

Other Results

Other academic or research-related results:

- Winner of the Call For Flags (CFF) competition at the *VIII Jornadas Nacionales de Investigación en Ciberseguridad 2023* (JNIC'23). The competition involved designing, developing and documenting capture the flag challenges, with awards given to the most original, complex and inter-disciplinary challenges. Source: https://2023.jnic.es/call-for-flags/ (accessed on February 19, 2025).
- Co-author of the registered industrial property of software *PII-2024-0039*, titled *"MALVADA: una herramienta para la generación de trazas de ejecución de malware"*, registered through the *Research Results Transfer Office (OTRI)* of the *Universidad de Zaragoza*.

Chapter 2

Background

This chapter provides the background knowledge and definitions required to understand the topics discussed in this dissertation. It introduces key concepts related to file-based Time-of-Check to Time-of-Use (TOCTOU) vulnerabilities, malware and techniques for its analysis, Windows internals, the role of sandbox environments in dynamic analysis, execution traces, and the Malware Behavior Catalog.

2.1 Concurrency Bugs

Concurrency bugs are caused by accesses to a shared resource between threads and processes without proper synchronization. These bugs can lead to vulnerabilities that, when triggered by adversaries, can cause a much broader impact on security, such as bypassing security checks, breaking the integrity of databases [14], hijacking the vulnerable program control flow, or escalating privileges [15], among others.

A common attack especially related to concurrency bugs is the privilege escalation attack, in which a malicious user gains access to other user accounts on the target system. The number of vulnerabilities related to privilege escalation has been increasing in recent years. For instance, in 2020 this type of vulnerability comprised 44% of all Microsoft vulnerabilities [16], and the trend continues as it was the #1 vulnerability category in 2023 [17]. There are two main types of privilege escalation: *horizontal privilege escalation* attacks, in which an attacker expands their privileges by taking over another (non-privileged) user account and abusing the legitimate privileges granted to the other user; and *vertical privilege escalation* attacks (also known as *privilege elevation*), which involve increasing privileges or perform privileged accesses beyond what a user (or an application or other asset) already has.

2.1.1 TOCTOU

Privilege elevation attacks are commonly caused by a particular type of concurrency bug, called *race condition bugs*. The root cause of these bugs is a TOCTOU

(Time-of-Check to Time-Of-Use) error, which occurs when a program checks a particular characteristic of an object (e.g., whether the file exists), and later takes some action that assumes the checked characteristic still holds [18]. The window of opportunity that the program leaves between the time of check and the time of use is then exploited by an adversary. The adversary can increase this time frame by various means, such as overloading the system or creating specific inputs for the vulnerable program. In addition, TOCTOU vulnerabilities can occur in different scenarios like memory accesses involving the kernel [19, 20] (also known as double-fetch bugs), Remote Attestation [21, 22], Trusted Computing [23, 24], or file-based I/O operation [18, 25], among others.

The first part of this dissertation focuses on *file-based TOCTOU* vulnerabilities, which are among the oldest known security flaws, dating back to the mid-70s [8, 9]. These race conditions, especially common in Unix-like systems, arise from the mapping of a filename to a unique inode and device number or ID. The inode and device ID are data structures storing information about a filesystem object and the device that contains it, respectively [26]. While the mapping of the inode and device ID to a file descriptor is race-free, the mapping of the filename to the inode and device ID is volatile. This volatility occurs because filenames and their associated inode and device ID can change with each system call invocation.

2.1.2 File-based TOCTOU Example

A well-known example of this kind of problem is `sendmail` [25], which used to look for a specific attribute of a mailbox file before adding new messages to it. Unfortunately, the verification and append operations are not an atomic unit. Consequently, if an adversary (the mailbox owner) replaces their mailbox file with a symbolic link to sensitive files (such as `/etc/passwd`, which contains information about system user accounts) between the verification and append operations, then `sendmail` will add email contents to `/etc/passwd`. As a result, the adversary can craft an email message to add a new user account with superuser privileges in the system.

Listing 2.1 Example of a file-based TOCTOU vulnerability.

```
1  // toctou.c
2  char *filename = argv[1];
3  // ...
4
5  // Check permissions
6  if(!access(filename, W_OK)){
7      // Open the file
8      file = fopen(filename, "a+");
9
10     // Write to file the user input
11     fwrite(buffer, sizeof(char), strlen(buffer), file);
12     fwrite("\n", sizeof(char), 2, file);
13     fclose(file);
14 } else
15     printf("No permission, exiting!\n");
16 }
```

Listing 2.2 Example of a file-based TOCTOU exploit.

```
1  #!/bin/bash
2  # exploit.sh
3  # (execute it as: ./exploit.sh /etc/passwd)
4  TEMPFILE="temp.file"
5  OLD_LS=`ls -l $1`
6  NEW_LS=`ls -l $1`
7
8  while [ "$OLD_LS" == "$NEW_LS" ]
9  do
10     rm -f $TEMPFILE
11     echo "From user" > $TEMPFILE
12     echo "TOCTOU success" | ./toctou $TEMPFILE > /dev/null &
13     unlink $TEMPFILE
14     ln -s $1 $TEMPFILE &
15     NEW_LS=`ls -l $1`
16 done
```

Listing 2.1 illustrates this typical security flaw. On line 6 there is a check of the write permission on a file (identified by a string) with the `access` system call. Once the verification is successful, the file is opened (line 8) and certain data is appended. If the program is owned by a user or group with high privileges (e.g., root) and is executed with the *setuid* or *setgid* permissions (i.e., users can run it with the privileges of the owner or group), the adversary can take advantage of the race window between the operations on lines 6 and 8. An example of the exploit used by an adversary is shown in Listing 2.2. Suppose the exploit is run to write to the `/etc/passwd` file, which is a protected file in UNIX-based systems. If an adversary iteratively creates a symbolic link to `/etc/passwd` (line 14) at the same time as the execution of the vulnerable program (line 12), the race condition will eventually occur and the attack will succeed, appending new

content to the protected file.

Specifically, file-based TOCTOU vulnerabilities are file-based race conditions that occur on filesystems with *weak* synchronization mechanisms (that is, they do not provide methods to ensure that filesystem objects remain unchanged between consecutive interactions with them). In the rest of this dissertation, we refer to file-based TOCTOU vulnerabilities simply as TOCTOU vulnerabilities. Given the non-deterministic nature of race conditions, the success of an attack is highly dependent on the precise and timely actions of the attacker at any given time during the execution of the vulnerable program. Furthermore, the occurrence of this type of vulnerability also depends on certain system calls being executed in a specific order, as well as environmental conditions [25, 27]. Therefore, the reproducibility of these vulnerabilities is typically very difficult.

2.2 Malware

Malware, short for *malicious software*, refers to any software intentionally designed to harm, exploit, or compromise computer systems and data. It serves as a broad term encompassing various harmful programs, including viruses, worms, and trojans, among others. The main objectives of malware are to gain unauthorized access, disrupt normal operations, steal sensitive information, or pave the ground for further malicious activity. By targeting the confidentiality, integrity, and availability of systems, malware poses significant risks to individuals, organizations, and governments. Often, malware disguises itself as legitimate software to deceive users into installing or executing it.

When suspicious software is detected, it must undergo analysis to determine its intent and whether it qualifies as malware or benign (also referred to as *goodware*). This analysis involves two interconnected processes:

- *Malware detection*, which focuses on identifying whether a given program or file is malicious or benign. This process typically serves as the first line of defense in malware analysis, aiming to determine the presence of suspicious or harmful behaviors, such as unauthorized access, file modifications, or network exploitation.
- *Malware classification*, on the other hand, goes a step further by categorizing detected malware into specific families or types based on their behavior, structure, or purpose. Classification helps security teams understand the nature of the threat, its potential impact, and how it relates to known malware.

Over the years, malware has grown more sophisticated and diverse [5, 7], resulting in a large number of types (e.g., ransomware, trojans, information stealers, and keyloggers) and families (e.g., WannaCry, NjRAT, AgentTesla, and LokiBot) [28]. The constant evolution of malware stresses the importance of malware analysis techniques to deter its impact.

2.2.1 Malware Analysis

Malware analysis is the process of dissecting suspicious software to discover its inner workings, identify malicious intent, and develop mitigation techniques [29, 30]. As cybersecurity threats grow in complexity, malware analysis plays a critical role in detecting, deterring, and mitigating attacks [31, 32].

In cybersecurity, the fight between defenders and adversaries is perpetual, and malware analysis is no exception in this evasion and pursuit game. As researchers and industry professionals develop new defense mechanisms, adversaries respond by crafting techniques to circumvent them, and vice versa. Evasive malware has caught the attention of researchers throughout the years due to its substantial security risks and analytical challenges [33, 34]. Analyzing unknown programs generally involves two main approaches [35]:

- *Static analysis*, which involves examining a program without executing it. This is usually done by the means of reverse engineering, a process that aims at inspecting the program's instructions and internal structures using tools like disassemblers, decompilers, and metadata extractors [36, 37]. Static analysis is effective for identifying known patterns, embedded strings, and suspicious code segments. However, it can be hindered by obfuscation techniques such as polymorphic or metamorphic code [38, 39], anti-disassembly [40], and code packing [41, 42, 43]. Static analysis is a heavy duty task that requires significant expertise and time to be carried out [44]. While static analysis provides valuable insight into malware's structure, it may not fully reveal how malware interacts with its environment during execution, a critical aspect addressed by dynamic analysis.
- *Dynamic analysis*, which involves running the program (ideally) in a controlled environment (such as a sandbox) to monitor its runtime behavior, including network communication, file system modifications, and registry changes. While this approach simulates a realistic execution environment, it is susceptible to a plethora of evasion tactics [45, 46, 47] aiming at thwarting the analysis. For instance, anti-debugging [48, 49, 50, 51], anti-virtualization [52, 53, 54], or anti-dynamic binary instrumentation [55] are just examples of malware methods to evade controlled environments and hide malicious actions. One of the primary applications of dynamic analysis is behavioral analysis, which analyzes the exhibited actions and interactions of malware during its execution, offering insights into system-level changes, API calls, and network activity. Behavioral analysis thus extends dynamic analysis by focusing on patterns in malware's runtime behavior, enabling detection and classification even in the presence of advanced obfuscation.

The ingenuity and adaptability of malware authors require continuous innovation in malware analysis methodologies. Consequently, combining static and dynamic analysis, known as *hybrid analysis* [56], is often necessary to achieve a comprehensive understanding of malware behavior.

2.3 Windows Internals and API

The Windows operating system is designed with a layered architecture that operates at two privilege levels [57]: *user mode*, where regular applications run with restricted access to system resources, ensuring that individual processes operate independently without directly modifying critical system components; and *kernel mode*, where core system components and drivers operate unrestricted, allowing core operating system functions to execute with elevated privileges. This separation ensures system stability and security by preventing user-level processes from directly accessing critical system functionalities [58].

To interact with operating system resources, user-mode applications use the Windows API [59] (formerly known as "Win32 API", a term still widely used), a well-documented collection of functions, procedures, data structures, and protocols that offer higher-level abstractions for managing hardware, internal system components, and to perform essential operations such as process management, memory allocation, file handling, and inter-process communication. The Windows API is implemented in dynamic-link libraries (DLLs) such as `kernel32.dll`, `kernelbase.dll`, or `advapi32.dll`, among others [60, 61]. These functions pass control to the kernel, which performs the requested operations after necessary security checks, with the goal of ensuring a controlled interaction between user-mode applications and system resources. The Windows API serves as a user-friendly interface for developers, providing a convenient way to access system functionality.

In addition to the Windows API, Windows provides the Native API (NT API), a lower-level interface that offers more direct control over system resources. NT API functions, typically prefixed with `Nt` or `Zw`, allow user-mode applications to access kernel-level functionality, bringing them conceptually closer to the kernel [57]. The NT API is exposed to the user through `ntdll.dll` and implemented in the kernel image, `ntoskrnl.exe` [60, 61]. While these system calls offer granular control, their use is generally discouraged because, unlike the Windows API, the NT API remains largely undocumented [61, 62, 63], inconsistent across Windows versions [64], and subject to changes without notice. Despite these challenges, malware developers frequently employ the NT API and direct system calls to evade security mechanisms, bypass API monitoring, and manipulate system behavior at a low level [65, 66, 67, 68].

2.4 Sandbox Environments

A *sandbox* is a controlled, restricted, and isolated environment designed for running and analyzing files or programs. The term was first introduced in 1993 in the context of software fault isolation [69]. While sandboxes have applications in many domains, in this dissertation we focus on their role in computer security, where the concept was introduced in 1996 [70]. Specifically, we examine sandboxes that run and monitor program behavior in controlled environments [71], with an emphasis on dynamic malware analysis.

2.4.1 Malware Sandbox

A *malware sandbox* is a specialized type of sandbox designed to run and analyze potentially malicious software, generating reports on the sample's behavior, capabilities, and properties [33]. Whether virtualized or not, these environments replicate real operating systems and user configurations so that the program or file (referred to as a *sample*) behaves as it would in a real-world scenario. This setup allows researchers to safely observe a sample's actions at runtime without compromising the integrity of the host system or network [29, 31]. Most sandboxes log the sample's operations and generate execution traces, capturing data such as API calls, syscalls, file operations, registry interactions, and network communications. Some also produce memory dumps. In essence, sandboxes are among the most critical tools for dynamic malware analysis, particularly for (i) behavioral analysis, (ii) malware detection, and (iii) malware classification [32].

2.4.2 Types of Malware Sandboxes

Malware sandboxes can be classified into two main categories based on their architecture and characteristics: virtualized sandboxes and physical (also known as *bare-metal*) sandboxes. Virtualized sandboxes use one or more Virtual Machines (VMs) running real operating systems, configured to match specific analysis requirements. This isolation ensures safe analysis, and provides flexibility by supporting multiple operating systems. However, virtualized sandboxes are resource intensive, can be slow, and can be difficult to configure and deploy. In addition, malware can use defense evasion tactics to detect virtualization artifacts and adjust its behavior in accordingly [72].

Virtualized sandboxes can be divided into two notable subtypes: agent-based sandboxes and hypervisor-based (also known as *agentless*) sandboxes. Agent-based sandboxes embed software agents within VMs to monitor malware execution and transmit behavioral data to the sandbox engine. While flexible, they are more susceptible to detection by malware with anti-virtualization capabilities. In contrast, hypervisor-based sandboxes operate at a deeper level, leveraging virtualization technology to observe malware activity and environmental changes without requiring an agent on the VM. This approach reduces the risk of detection, but requires hardware with specific virtualization extensions and typically involves more complex configurations and performance overheads.

Bare-metal sandboxes, on the other hand, run directly on the physical hardware, eliminating virtualization layers. This approach provides a high degree of realism, making them more resistant to detection as there are no virtualization-specific artifacts. Bare-metal sandboxes are particularly useful for analyzing malware that attacks hardware or firmware. However, they are generally more expensive, less flexible, and harder to maintain compared to virtualized sandboxes.

Finally, sandboxes can be deployed on-premises or as cloud-based services following the software-as-a-service (SaaS) model. On-premises sandboxes offer a great deal of configurability but require substantial hardware resources. In contrast, SaaS sandboxes eliminate the need for hardware installation or configuration,

making them easy to use. However, they can raise privacy concerns, offer limited customization, and rely on external infrastructure, which may become unavailable at times.

2.5 Execution Traces

In the context of this dissertation, an *execution trace* is a detailed record of a program's behavior during runtime, capturing the sequence of actions performed by a process or group of processes on a system. In the context of dynamic malware analysis, execution traces typically consist of system and API calls invoked by a program, enriched with additional contextual information such as process hierarchies, call parameters, return values, and accessed system resources (e.g., files, registry entries, or mutexes).

Execution traces are generated by running programs and monitoring the execution, for example in a sandbox. They reflect the actual behavior exhibited by the program, which makes them particularly useful for behavior-based malware analysis as they can help identifying behavior patterns unique to specific malware families or types, or distinguish benign from malicious software.

2.6 Malware Behavior Catalog

To standardize malware analysis and facilitate cross comparisons, MITRE is developing the Malware Behavior Catalog (MBC) [73], a comprehensive reference that lists and defines malware behaviors to support the malware analysis process [30]. Its primary goals are to standardize malware reporting, correlate analysis results, and help identify common behaviors.

The MBC defines malware objectives, behaviors, and methods, each tagged with a unique identifier to facilitate mapping and reporting. Objectives represent high-level actions that define the malware's intentions, while behaviors represent the specific actions that contribute to these objectives. Each behavior can be further refined using methods, which specify how the behavior is implemented. For example, the Anti-Behavioral Analysis objective (ID: OB0001)[1] includes the "Debugger Detection" behavior (ID: B0001), which can use the "API Hook Detection" method (ID: B0001.001) or invoke Windows-specific API calls such as `CheckRemoteDebuggerPresent` (ID: B0001.002).

MBC also defines micro-objectives, which are low-level actions fundamental to any running program, regardless of whether it is malicious. These micro-objectives contain micro-behaviors, which can be further refined using methods. For example, the Operating System micro-objective (ID: OC0008) includes the "Registry" micro-behavior (ID: C0036), which specifies methods such as "Create Registry Key" (ID: C0036.004)[2].

[1] See https://github.com/MBCProject/mbc-markdown/blob/main/anti-behavioral-analysis/README.md.

[2] See https://github.com/MBCProject/mbc-markdown/blob/main/micro-behaviors/

operating-system/registry.md.

Part I

Static Approach

Introduction

In the early phases of software development, vulnerabilities should ideally be identified before deployment. Static analysis offers the ability to inspect source code without executing it, enabling early detection of programming patterns that may lead to security flaws. This chapter explores the file-based time-of-check to time-of-use (TOCTOU) vulnerability, examining it from both defensive and offensive perspectives. This specific vulnerability was chosen as it exemplifies a behavioral pattern that can potentially be identified through source code-level analysis.

Chapter 3

Static Analysis of Behavior Patterns: Race Condition Use Case

As an initial step in our static analysis approach, we examine a race condition source-code vulnerability, as it represents a clear example of a malicious or vulnerable behavior pattern. This vulnerability arises when specific functions are executed in sequence and reference the same filesystem object. To gain a comprehensive understanding of the vulnerability, its characteristics, and existing research on this topic, we first analyze the vulnerability and review defense and attack techniques previously proposed in the academic literature. This chapter presents the systematic literature review we conducted, the defense and attack mechanisms identified, a proposed taxonomy of defense and attack strategies, and discusses future research trends and directions.

Related Work. Several different TOCTOU vulnerabilities are mentioned in other literature reviews or surveys. The survey in [74] focuses on double-fetch vulnerabilities, which is a vulnerability that occurs when data consistency between the kernel and the user space is violated in a race condition. Vulnerabilities in remote attestation in wireless sensor networks are discussed in [75]. TOCTOU vulnerabilities can also occur in this context, as attesting a node occurs at a particular point in time and does not guarantee that the node was not temporarily compromised before or that it will not be compromised right after the attestation. TOCTOU vulnerabilities due to naming collusion in Android are explored in [76], which provides a systematic review of permission-based Android security.

Unlike these works, our work focuses exclusively on file-based TOCTOU vulnerabilities. Furthermore, the previous works do not provide an in-depth analysis of how this vulnerability is exploited or of existing defense and offensive techniques. To the best of our knowledge, we present the first systematic literature review of file-based TOCTOU vulnerabilities.

Table 3.1 Common Weakness Enumerations related to TOCTOU.

CWE ID	Vulnerability
CWE-59	*Improper Link Resolution Before File Access ('Link Following')*
CWE-61	*UNIX Symbolic Link (Symlink) Following*.
CWE-62	*UNIX Hard Link*
CWE-362	*Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')*
CWE-363	*Race Condition Enabling Link Following*
CWE-367	*Time-of-check Time-of-use (TOCTOU) Race Condition* (not only file-based TOCTOU)
CWE-386	*Symbolic Name not Mapping to Correct Object*
CWE-706	*Use of Incorrectly-Resolved Name or Reference*

3.1 Context

Today, many applications are deployed on large-scale distributed systems and multi-core processors, which perform multiple tasks concurrently while sharing common resources such as memory, disk, or network. The intrinsic characteristics of the simultaneous execution of programs make them very difficult to write, test, and debug [77, 78], which facilitates the existence of concurrency bugs. One of the most critical, nefarious and prevalent concurrency bugs are the file-based race conditions known as TOCTOU (see Section 2.1).

Despite the age of this security flaw [8, 9], numerous vulnerabilities are still reported each year related to TOCTOU vulnerabilities. For example, at the time of writing, a query to find TOCTOU-related vulnerabilities returns hundreds [10] or even thousands [11] of results in major vulnerability databases, with the newest reports being only a few days old. This clearly shows that it is still a significant security problem and that the CVE release for TOCTOU vulnerabilities is common in the software industry. Furthermore, this vulnerability affects projects of any size, such as open source projects [79, 80], and major software vendors [81, 82, 83, 84]. *The proof of the pudding is in the eating*: as shown in Table 3.1, there are several Common Weakness Enumeration (CWE) entries related to TOCTOU. CWEs represent a common language for discussing, finding, and addressing the causes of software security vulnerabilities, currently maintained by the MITRE Corporation. Each individual CWE represents only one type of vulnerability.

In this dissertation we aim to systematically review the scientific literature in order to find techniques to mitigate TOCTOU vulnerabilities, as well as techniques to exploit these vulnerabilities. Specifically, we review the literature to find out what techniques have been proposed, how they are implemented, how they detect TOCTOU vulnerabilities, which operating system they target, and whether any source code or software tool is available to reproduce the experimental results.

In summary, our contributions are the following:

- We conduct a comprehensive review of the literature on defense and attack

solutions against TOCTOU vulnerabilities. In particular, we found 37 articles proposing some kind of defense solution and only 4 articles proposing attacks against TOCTOU.

- We propose a taxonomy for TOCTOU defenses and attacks, according to when they perform the vulnerability detection/exploitation and at what level they operate. Furthermore, we classify TOCTOU attacks based on the attack vector they exploit.
- We highlight future research trends and directions regarding defense solutions for TOCTOU vulnerabilities. Our proposals cover modifying current operating system calls to make them race-free and security focused, modifying the kernel to avoid the use of filenames, and the use of transactional filesystems. We provide more details on this matter in Section 3.4.2.

3.2 Systematic Literature Review

We conduct a systematic review of the literature following the recommendations given in [85] to find detection, prevention, avoidance or exploitation techniques that are related to TOCTOU vulnerabilities. Systematic literature reviews are methodical, complete, transparent, and replicable studies that allow the compilation of results following reproducible and bias-free research carried out by consulting the main scientific and academic search engines [86].

Next, we explain in detail the methodology that we have followed. We first state the research questions and the search strategy used. We then present the criteria used to select studies for quantitative analysis. Finally, we summarize the number of articles obtained in each execution phase of our review protocol.

3.2.1 Research Questions

The main objective of this research is to review the literature in the field of prevention, detection, and mitigation mechanisms for TOCTOU vulnerabilities, as well as related exploitation techniques. In particular, we want to know the underlying principles behind mitigating and attacking file-based race conditions, how they affect the host operating system, whether they are located at the user or kernel-space level, and whether any tool or source code exists to replicate the experimental results. More formally, we formulate the following research questions (RQ):

RQ1.- How do defensive and offensive techniques of file-based TOCTOU vulnerabilities work?

RQ2.- In which regions of the memory do they reside?

RQ3.- When is the vulnerability detected or exploited?

RQ4.- In which operating system is the technique implemented?

RQ5.- Is there any tool or source code available to validate or replicate the experimental results?

3.2.2 Search Strategy

We consulted various scientific databases that allow the results to be exported for later analysis. In particular, we considered IEEE Xplore, ScienceDirect, Scopus, and ACM since together they cover the main journals and conferences in the field of interest.

The advanced search relied on keywords that were carefully selected and modified throughout the review process to improve the results and fulfill the purpose of our review. Specifically, we started from scratch by conducting a preliminary search with the term "*TOCTOU*" and adding those terms that help us narrow down the results (for instance, synonyms have also been contemplated). The final search string is:

> (*TOCTOU* **OR** *TOCTTOU* **OR** "*time of check to time of use*") **AND** *file* **AND** (attack* **OR** exploit* **OR** abus***OR** defen* **OR** mitigat* **OR** fix*)

We looked for items until the year 2021, without setting any initial year.

As we are only interested in scientific/academic works that have been published in peer-reviewed scientific journals and conferences, other works such as gray literature, books, standards, or patents are discarded from our results. In addition, we carried out a complementary manual search by reviewing the title of the works presented in the Tier-1 and Tier-2 conferences of computer security, according to [87]. This search consisted of checking whether the titles of the publications contained at least one of the following keywords: *file*, *race*, *time* or *toc**. A total of 470 conferences (216 from Tier-1 and 264 from Tier-2) have been verified and all editions of each conference have been reviewed. For example, regarding the *IEEE Symposium on Security and Privacy*, 25 editions have been reviewed, from 1995 to 2020. In particular, the following conferences have been consulted: *IEEE Symposium on Security and Privacy*, *ACM Conference on Computer and Communications Security*, *USENIX Security Symposium*, *Network and Distributed System Security Symposium*, *Annual International Cryptology Conference*, *International Conference on the Theory and Application of Cryptographic Techniques*, *European Symposium on Research in Computer Security*, *International Symposium on Recent Advances in Intrusion Detection*, *Annual Computer Security Applications Conference*, *Dependable Systems and Networks*, *ACM Internet Measurement Conference*, *ACM Asia Conference on Computer and Communications Security*, *International Symposium on Privacy Enhancing Technologies*, *IEEE European Symposium on Security and Privacy*, *IEEE Computer Security Foundations Symposium*, *International Conference on Theory and Application of Cryptology and Information Security*, *Theory of Cryptography Conference*, and *Workshop on Cryptographic Hardware and Embedded Systems*.

After thoroughly reviewing the proceedings of these conferences, 13 articles were selected according to their titles for further study. In addition, we also carried out snowballing (i.e., reference inspection) on all selected articles. This process allowed us to find 11 additional relevant articles. We provide more details on the number of articles selected during the review process in Section 3.3.

Table 3.2 Inclusion (IC) and exclusion (EC) criteria.

Type	Criterion
IC1	The article focuses on concurrency attacks.
IC2	The main contribution of the article is a defense against TOCTOU or an attack to exploit it.
EC1	The article is a short introductory paper, early access, or conference abstract.
EC2	The article is not written in English.
EC3	The article is duplicated.
EC4	The article focuses on other types of TOCTOU rather than file-based TOCTOU.
EC5	The article is not available for downloading or reading.

3.2.3 Study Selection Criteria

After finding these initial articles, we used the `StArt` tool to better perform the research and article selection processes. `StArt` [88, 89] is a tool that helps researchers define and execute the systematic review protocol. This tool automatically detects duplicate results, rates them based on predefined keywords, and provides visualizations of the current review status, among other features.

The scoring metric provided by `StArt` was used as the first filter. The `StArt` scoring system allows the user to rate each article based on the appearance of certain keywords in its title, list of keywords, or abstract. The rating system we have used is simple, but allows us to really focus on the relevant articles. For each term in the keyword bag, the score value is obtained as follows:

- Add 5 points if the term appears in the article title.
- Add 3 points if the term appears in the article abstract.
- Add 2 points if the term appears in the article's keywords.

The keyword bag comprises the following main terms, as well as their synonyms and plurals: *TOCTOU*, *attack*, *concurrency*, *defense*, *exploit*, *filesystem*, *interference*, *mitigation*, *race condition* and *data race*. We also consider variations of these terms. The term TOCTOU, for example, has different forms throughout the literature such as *TOCTTOU*, *time-of-check-time-of-use*, or *time of check to time of use*. We set a minimum score of 15 to select an article. Note that articles relevant to our research should easily exceed that minimum value, given the scoring scheme described above.

All articles above the threshold are considered for further inspection and selected or excluded based on the criteria defined in Table 3.2. These criteria help us select and focus on the most relevant articles in relation to the proposed RQs. After applying the criteria filter, the selected articles are studied in depth to answer each RQ indicated in Section 3.2.1.

3.2.4 Articles Collected and Reviewed

After running the search protocol, we collected 563 articles. 66 of them have been discarded for being duplicates. After applying the scoring threshold, only 126 remained, which were further analyzed to apply the selection and exclusion criteria. In addition, 13 articles were collected after manually reviewing the Tier-1 and Tier-2 computer security conferences, according to [87]. 6 of them were also discarded because they were duplicates with regard to the previously considered corpus. These manually-included articles have not undergone the scoring system, but were reviewed immediately. Again, 6 of them were discarded because they did not focus on file-based race conditions. This process resulted in a total of 41 articles that we considered for our quantitative synthesis analysis.

This information is combined and summarized in Figure 3.1. The PRISMA diagram [90] summarizes the execution phases of our review protocol (*identification*, *screening*, *eligibility*, and *inclusion*) and the articles obtained in each phase.

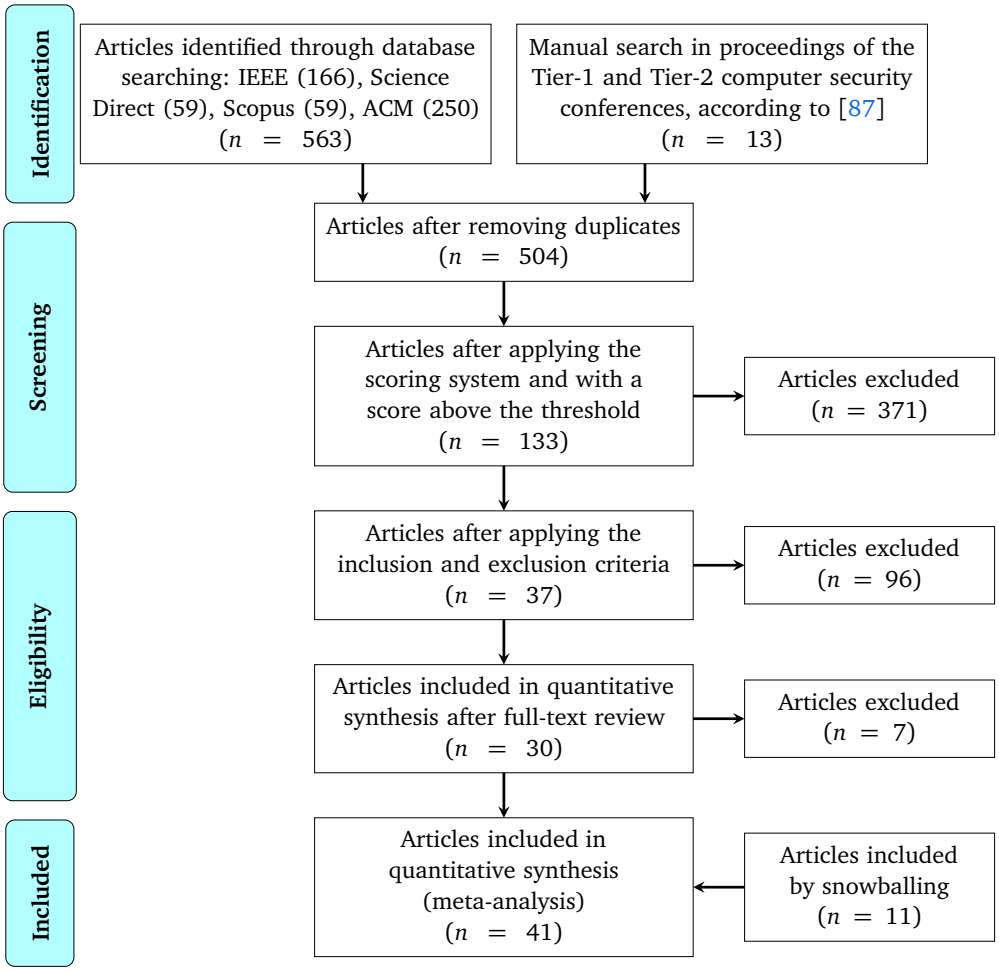


Figure 3.1 PRISMA diagram of our review protocol.

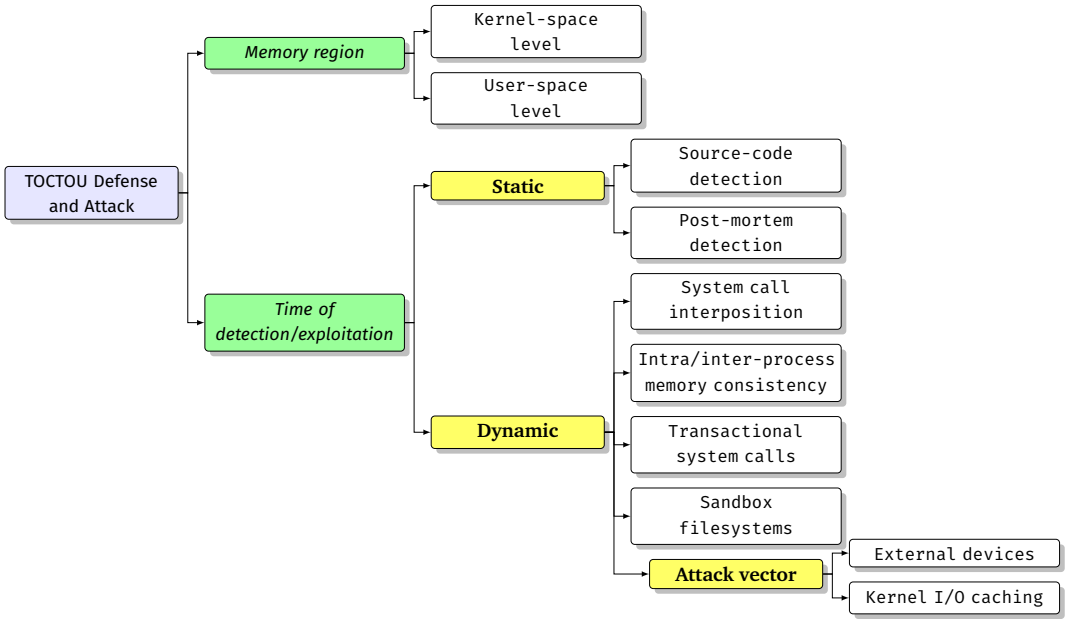


Figure 3.2 Proposed taxonomy for TOCTOU defense and attack mechanisms.

3.3 Analysis of Results

This section presents the results of the systematic review of the literature. We first propose a taxonomy for current TOCTOU defense and attack mechanisms that collects the main insights drawn after the systematic review of the literature and responds to the research questions established in Section 3.2.1. We then explain the different categories into which TOCTOU defenses can be classified, and then we explain the articles in each category in more detail. Finally, we follow the same narrative to explain the studies found on attack methods against TOCTOU vulnerabilities.

3.3.1 Towards a Taxonomy for TOCTOU Defense and Attack Mechanisms

Figure 3.2 illustrates the classification of TOCTOU defense and attack mechanisms resulting from responding to the research questions established in Section 3.2.1.

A TOCTOU defense or attack mechanism can be categorized considering two aspects: *memory region*, which indicates at which level the TOCTOU defense or exploitation occurs; and *time of detection/exploitation*, which means the time when the TOCTOU vulnerability is detected or exploited. Memory regions can be divided into *user-space level*, which includes solutions that run in the same memory area as the vulnerable application (and some drivers), and *kernel-space level*, which comprises solutions that run in the same memory area where the operating system kernel runs (as well as kernel extensions and most device drivers). Detection/exploitation times can be divided into *static*, which comprises defense solutions in which the vulnerability is detected without the vulnerable application running or after it has been run (in other words, the detection of the vulnerability occurs before or after the execution); and *dynamic*, which includes proposals capable of detecting or exploiting the vulnerability when the vulnerable

program is running.

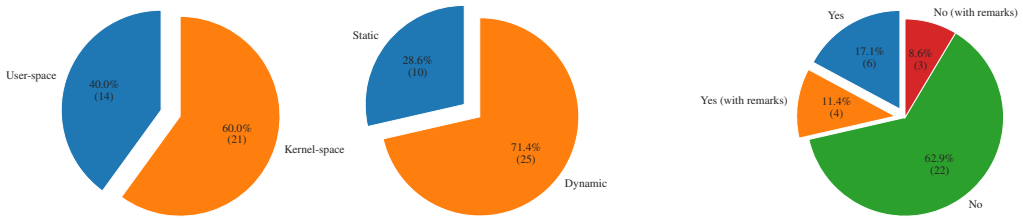
Regarding memory regions, as shown in Figure 3.3a, 60% of the defense solutions are located in kernel-space, while 40% are located in user-space. As for offensive techniques, all of these are attacks from the user-space level. Recall that this vulnerability allows the attacker to gain system privileges. If the attacker is able to execute the attack from the kernel-space level, there is no real gain in exploiting the vulnerability. We give a more detailed discussion on this matter below in Section 3.4.1.

Regarding time of detection/exploitation, static proposals are exclusively defense approaches, and can be further divided into *source code detection* approaches, which analyze the source code of the vulnerable program [18, 91, 92], and *post-mortem detection* approaches, which detect the TOCTOU vulnerability after the exploitation attempt has already occurred [93, 94, 95, 96, 97, 98, 99, 100]. Unlike post-mortem detection approaches, source code detection approaches find the TOCTOU vulnerability before it is exploited, which is often preferable in certain systems such as critical infrastructures or systems with highly sensitive information.

Dynamic proposals are more diverse, based on a multitude of runtime analysis techniques. Some defense approaches use *system call interposition*, monitoring the behavior of the programs by intercepting their system calls. This monitoring can occur at either the user-space level [101, 102, 103, 104, 105, 106] or the kernel-space level [25, 27, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119]. A few defense approaches propose *intra-process* or *inter-process techniques for memory consistency* (both at kernel-space level [120, 121]) to guarantee the consistency of variables shared across threads. Other kernel-level defense approaches propose *transactional system calls* [122] as an alternative to traditional filesystems to prevent race conditions from occurring in system resources, while others propose *sandbox filesystems* [123, 124] to protect against unauthorized file modifications caused by file-based race condition vulnerabilities.

In particular, as shown in Figure 3.3b, 71.4% of the defense solutions are dynamic, while only 28.6% are static. Note that we use the same terms as in program binary analysis (static and dynamic), but we refer to the time of detection rather than how the program is analyzed (not running or while running).

The attack techniques are all dynamic since the vulnerable program must be running to exploit it. Attack mechanisms can be further classified according to the *attack vector*, which defines the path or means that an attacker takes to exploit a vulnerability. Regarding TOCTOU vulnerabilities, we have found two different attack vectors. The first is the *external devices* vector, which consists of abusing the trust that the system places in external devices (i.e., USB sticks or SD cards) during the installation process of a given application. During the installation process, the system can use external devices to store sensitive data that can be altered or manipulated by an attacker, since the external device is under their control. This attack vector is used in [125, 126]. The second is *kernel Input/Output (I/O) caching*, which involves attacks that abuse the kernel's I/O caching mechanism to deliberately increase the window of the vulnerability. If the attacker can tamper



(a) Detection location **(b)** Detection time **(c)** Reproducibility

Figure 3.3 Graphical summary of defense solutions (detection location, detection time, and reproducible).

with the kernel cache, they will force I/O operations so that the kernel resolves the specified pathnames. These I/O operations take time to complete, which broadens the vulnerability window and facilitates exploitation. This attack vector is used in [127, 128].

3.3.2 On TOCTOU Defenses

Figure 3.4 shows a timeline of the articles studied in this dissertation focused on defense solutions against TOCTOU vulnerabilities. Although the first references to TOCTOU vulnerabilities are approximately 50 years old [8, 9, 129], the first defense solution was not proposed until 1994 [93]. Defense solutions then extend over the years until 2019, the date of the last solution we found.

Figure 3.3a and Figure 3.3b show a graphical summary of defense solutions according to *memory region* and *time of detection*, respectively. Regarding detection location, there is no clear or predominant choice among the proposed solutions analyzed in this systematic literature review, although kernel-level solutions represent slightly more than half. As for the moment when TOCTOU is detected, almost three-quarters of the defense solutions are dynamic (specifically, 71.4%).

Finally, it is worth mentioning the trend of detection techniques chosen by the proposed defenses and their level of execution. The timeline in Figure 3.4 clearly indicates that the first solutions were based on static user-space detection, beginning in 1994. In the early 2000s, the first solutions based on dynamic kernel-space detection emerged. Dynamic detection at the user-space level began in 2006. Additionally, 19 dynamic kernel-space solutions were published between 2001 and 2014, thus averaging more than one publication per year. In contrast, during our research we found only one defense mechanism that relies on a static kernel-based solution.

Table 3.3 summarizes our findings on defensive techniques, answering research questions RQ2 through RQ5. A detailed discussion answering these questions is provided in Section 3.4. For each study, we indicate in the table the detection location and the detection type. We also indicate the operating system on which it is evaluated (the specific operating system and version if indicated in the publication, or otherwise the generic operating system) and if the proposed solution is reproducible (that is, if a prototype tool or source code is provided for download). In this regard, we consider reproducibility to be an important issue in terms

of scientific rigor and its contribution to open science. Figure 3.3c summarizes the reproducibility of the techniques proposed by the works analyzed in this systematic literature review. In addition, we indicate the items each technique uses to identify (uniquely) a filesystem object and detect external manipulation. Figure 3.6 shows in a bar graph how many defense techniques use each item identified in the literature review. We describe each of these works in more detail below to answer RQ1. The studies, which are presented in chronological order, have been grouped according to the detection location (user-space level versus kernel-space level) and the detection time (static versus dynamic).

Based on Static User-Space Detection

The first work we found is [93]. The authors proposed a defense solution that detects TOCTOU exploitation attempts after the vulnerable program has been executed. The detection process is mainly based on the analysis of execution traces. We refer to this type of static detection as *post-mortem detection*, since the detection is made *after* the exploitation attempt has been carried out and the vulnerable program has finished its execution. The authors' solution monitors the execution of privileged programs, auditing certain sequences of unwanted actions and then checking them against expressions described by the logic of predicates and regular expressions. The authors also presented a software prototype that runs on the Sun Solaris operating system and is capable of detecting TOCTOU vulnerabilities in three widely used programs (specifically, `fingerd`, `rdist`, and `sendmail`). However, no reference to the source code or to the tool itself is provided to facilitate reproduction of the experiments.

File-based race conditions on Unix-like operating systems were first discussed in detail in [91], concluding that kernel modifications are required to eliminate file-based race conditions. In addition, a lexical source code scanner is proposed to detect the vulnerabilities related to file access. Although the presented prototype successfully discovered new instances of TOCTOU, the tool or its source code is not available.

In a later work, Bishop and Dilger [18] demonstrated that privilege escalation attacks that exploit TOCTOU vulnerabilities only occur when filesystem objects are referenced by their names and not by file descriptors. Again, a software prototype is developed to (lexically) parse C source code files and detect file-based race conditions. Detection is based on pattern matching techniques and dependency and data-flow graph analysis. Unfortunately, the prototype is not accessible.

Goyal et al. [94] proposed an algorithm that is evaluated on a Unix-like system to detect TOCTOU attacks based on the analysis of execution traces (that is, it performs a post-mortem detection). A set of predefined rules is verified against execution traces to detect successful exploitation of TOCTOU vulnerabilities. As the authors stated, this solution is incomplete as the attack patterns must be known beforehand. No reference is provided to the availability of the prototype that implements the algorithm.

A probabilistic solution was proposed in [92], in which the source code of the

Table 3.3 Overview of TOCTOU defenses, sorted by publication year.

Publication	Detection location	Detection time	Operating System	Reproducible	Items used to identify file objects
[93]	User-space	Static	Sun Solaris	✗	File path, permission mode, owner ID, GID
[91]	User-space	Static	Unix-like	✗	(n/a)
[18]	User-space	Static	SunOS and Solaris	✗	(n/a)
[107]	Kernel-space	Dynamic	Linux	✗†	Filename
[108]	Kernel-space	Dynamic	Red Hat Linux 6.2	✗	(n/a)
[94]	User-space	Static	Unix-like	✗	(n/a)
[109]	Kernel-space	Dynamic	OpenBSD	✗	File path, PID, current time, file operation, inode
[92]	User-space	Static	Linux 2.4.18, FreeBSD 4.7, Solaris 8, SunOS 4.1.4	✓	Inode, device ID, generation number (if available)
[110]	Kernel-space	Dynamic	Linux 2.4.20	✗	Filename, inode
[111]	Kernel-space	Dynamic	Red Hat Linux 7.3 (kernel 2.4.18)	✗	Filename, inode
[27]	Kernel-space	Dynamic	Red Hat Linux 9 (kernel 2.4.20)	✗	File path, arguments, PID, filename, UID, GID, EUID, EGID
[112]	Kernel-space	Dynamic	Red Hat Linux 7.3	✗	File path, PID, inode, # of processes accessing the file
[101]	User-space	Dynamic	Linux	✓	Filename
[120]	Kernel-space	Dynamic	Linux 2.4.28	✗	File path, # of processes accessing the file, UID
[95]	User-space	Static	Red Hat Linux 7.3	✗	File path, UID, inode
[102, 103]	User-space	Dynamic	Solaris 8, AIX 5.3 and Linux 2.4.26, 2.6.20 and 2.6.22	✓	Filename, inode
[113]	Kernel-space	Dynamic	POSIX	✓	Device ID, inode
[122]	Kernel-space	Dynamic	Linux 2.6.22	✓*	Inode
[104]	User-space	Dynamic	POSIX	✓⋆	File path, UID
[25]	Kernel-space	Dynamic	Linux 2.4.28	✗	File path, logical disk block
[114]	Kernel-space	Dynamic	Linux	✗	(n/a)
[96, 97]	Kernel-space	Static	Linux 2.6.35	✓†	Inode, file handlers, UID, PID, PPID
[115]	Kernel-space	Dynamic	Linux 2.6.35	✗	Inode
[105]	User-space	Dynamic	Unix-like	✗†	Device ID, inode, parent directory
[116]	Kernel-space	Dynamic	Linux 3.2.0	✗	Inode
[98]	User-space	Static	Linux 2.6.15	✗	File descriptors, inode, PID
[123]	Kernel-space	Dynamic	Linux 3.2.0-36 and 3.8.10	✓	(n/a)
[117]	Kernel-space	Dynamic	Linux 2.6.35 and 3.2.0	✓‡	Device ID, inode
[118]	Kernel-space	Dynamic	Linux 3.2	✗	Inode
[121]	Kernel-space	Dynamic	Linux	✗	(n/a)
[119]	Kernel-space	Dynamic	Linux	✗	Device ID, inode,
[106]	User-space	Dynamic	POSIX.1-2008 compliant	✗†	Inode
[99]	User-space	Static	Any on Simics	✗	PID, File descriptors, inode
[124]	Kernel-space	Dynamic	Linux 4.10	✓	File path, inode
[100]	User-space	Static	(n/a)	✗	(n/a)

(n/a): Not available; †: No longer available; ‡: Found in the related article material such as the conference presentation; ∗: Found by searching the Internet; ⋆: Lack of details to fully replicate it.

vulnerable program is modified to reduce the probability of success of an attack. This solution replicates an arbitrary number of times the execution of the original sequence of potential vulnerable actions, verifying afterwards if the accessed file is changed. Since the authors provided examples on how to modify the source code, we consider this to be a reproducible work.

The solution proposed by Bhatkar et al. [95] was also based on post-mortem detection as it parses execution traces to build a control and data-flow graph which is then verified against a set of learning temporal properties representing TOCTOU vulnerabilities. The solution is implemented in a software tool for Red Hat Linux 7.3 that is not available.

Yu et al. [98] proposed a virtualization-based solution dubbed `SimRacer`, which tests the occurrence of certain types of race conditions by replaying event traces of the executions of the program. The authors test it against a set of vulnerable programs, successfully detecting all TOCTOU vulnerabilities. By construction, `SimRacer` is compatible with any operating system that runs on top of the full-system `Simics` simulator. Unfortunately, neither the tool nor its source code is available.

Yu et al. [99] introduced `SIMEXPLORER`, an improved version of `SimRacer`. This solution extends the detection algorithms of `SimRacer` to consider hardware interruptions and signal handlers. Tested with 24 programs, it detected 36 out of 41 previously known vulnerabilities. Like `SimRacer`, `SIMEXPLORER` can run on any OS that runs on top of Simics and is also not available.

The latest static user-space solution also relies on post-mortem detection. Capobianco et al. [100] provided a solution for detecting TOCTOU vulnerabilities by calculating the attack graph of the vulnerable program and analyzing it to detect sequences of events that may end up exploiting the vulnerability. These attack graphs allow the user to find the attack surface used by the adversaries and how they effectively elevate privileges. While the authors conduct detection statically, they explore how it can be applied at runtime and discuss the research challenges, as their primary goal is to improve the capabilities of intrusion detection systems. Unfortunately, neither the prototype nor the source code is available.

Based on Dynamic User-Space Detection

Aggarwal and Jalote [101] proposed the first solution based on dynamic user-space detection. In particular, the solution relies on a software agent integrated in the vulnerable program to control its execution while detecting common vulnerabilities. System calls are monitored by the agent and sent to another process in charge of real-time analysis of the behavior of the vulnerable program. A software prototype for Linux is provided and evaluated, which succeeds in stopping file-based TOCTOU exploits.

A new standard function was provided in [102, 103] to avoid the TOCTOU vulnerability window between the system call sequence `access` and `open`. This new feature is an enhancement of a previous version introduced in [92] to defend against complex attacks such as filesystem mazes [127]. More details on this type

of attacks are given in Section 3.3.3. Although the solution provides good results, it still has some drawbacks, such as the difficulty of deployment in production, defending against circular symbolic links, or multi-threaded applications, among others. Source code is provided by the authors. Since [103] is the full report of [102], we consider them as a single solution.

Chari et al. [104] proposed a set of secure calls for POSIX-compliant operating systems to prevent privilege escalation attacks based on TOCTOU. These secure calls overlap actual system calls, monitoring invocations of certain file-based system calls for unwanted inputs. However, the solution does not work with statically-linked programs since it is provided as a software library. Unfortunately, the work in [104] only shows a subset of the proposed secure calls, leaving the reader without full knowledge to fully reproduce their work.

Likewise, Payer and Gross [105] also proposed a Unix-based software library dubbed `DynaRace`. This binary-instrumentation solution is based on the state-machine formalism: it maintains a state machine for each file used by the vulnerable program, updated upon a sequence of certain file-based system calls, to detect unwanted behavior. When detected, it issues a warning and aborts the vulnerable program. The tool was available on the author's website.

A software library solution that detects file-based race conditions is also provided in [106]. This solution, though, puts all the responsibility on the team of software developers, as they must use the secure system calls provided by the software library rather than those of the operating system. In addition, it can also generate false positives, and leaves the vulnerable program in an unknown state after detecting an exploitation attempt. The source code was available on the website of the author's research group. Unfortunately, it is no longer online.

Based on Static Kernel-Space Detection

In [96, 97], the authors introduced a system dubbed `RacePro` capable of detecting different types of race conditions, including TOCTOU. The system monitors program executions and audits system calls that access shared kernel objects. These audit records are then verified against benign and harmful race models, which are known in advance. `RacePro` was tested on Linux Kernel version 2.6.35 and found 4 unknown bugs in common Linux tools such as `make` and `locate`. The source code for `RacePro` is freely available at [130], although it is not explicitly mentioned in the paper. Since [96] is the preliminary work of [97], we consider them a single solution.

Based on Dynamic Kernel-Space Detection

`RaceGuard` is a Linux kernel modification proposed in [107] to detect race conditions when creating temporary files. Internally, it keeps track of filenames created through certain system calls, which are then checked for race conditions. However, this solution is incomplete as it only monitors the creation of new files, regardless of existing ones. Although the source code was originally available as a kernel patch for Immunix, this commercial operating system was discontinued in 2003.

Similarly, Ko and Redmond proposed in [108] a kernel module for Red Hat Linux 6.2 that monitors system calls made during the execution of a privileged program, deliberately performing them ahead of the execution of system calls in non-privileged programs. However, the availability of the prototype is not mentioned.

In [109], Tsyrklevich and Yee also proposed a kernel module for OpenBSD to detect sequences of system calls that can lead to race conditions. The module removes the sharing property of some file objects, making their accesses mutually exclusive. The solution is successfully evaluated in four attack scenarios, detecting and stopping all exploitation attempts. However, as the authors admit, this solution is not free of race conditions, as the interception of system calls implicitly generates another race condition vulnerability window. Although the authors state that their proposal is portable to other operating systems (not only Unix-like), the source code is not provided.

`Race-attack Prevention System` is a system proposed by Park et al. [110] that also intercepts system calls and checks the consistency between them. Built on top of `RaceGuard` [107], it verifies if shared file objects are manipulated from a transactional point of view to avoid race conditions. This solution is implemented for Linux kernel 2.4.20, but unfortunately there is no mention of where or how it can be obtained.

Lhee and Chapin [111] proposed another software library that intercepts system calls to detect TOCTOU vulnerabilities, detecting inconsistencies in file objects by means of their binding information (specifically, the inode and the filename). This solution is implemented as a kernel module for Red Hat Linux 7.3 running on top of Linux kernel version 2.4.18. Although the authors implemented, tested, and evaluated a simplified prototype of their defense proposal, it is not available.

Uppuluri et al. [112] defined a set of security policies, specified using a behavior modeling specification language, that can be compiled and integrated into different detection engines. A prototype engine is implemented as a kernel module for the Red Hat 7.3 operating system. As before, this solution is not free of race conditions, as it relies on the interception of system calls. In addition, the prototype is not available either.

Wei and Pu [27] proposed a model for TOCTOU vulnerabilities in Unix filesystems, called *CUU model*. This model consists of pairs of system calls that can lead to a TOCTOU vulnerability. In addition, they propose different tools that are based on this model to monitor and detect TOCTOU vulnerabilities in Linux systems at the kernel level. These tools were successfully tested in version 2.4.20 of the Red Hat Linux 9 kernel, in approximately 130 utility programs. This solution, though, is only suited for single core processors. However, none of the tools are publicly available.

A defense solution called *Event Driven Guarding of Invariants* (EDGI), based on the CUU model, is presented in [120]. Vulnerable pairs of system calls are translated into invariants, which are used as sophisticated locks with a time-out mechanism. This solution is implemented in version 2.4.28 of the Linux kernel, but neither

the tool nor its source code is available.

Kupsch and Miller proposed in [113] a set of functions to create and manipulate files in a secure way, replacing standard C functions such as `creat`, `open`, or `fopen`, to name a few, to eliminate TOCTOU race conditions. A working implementation of these functions is publicly available at [131].

Porter et al. [122] introduced a variant of Linux 2.6.22, dubbed `TxOS`, which incorporates system call transactions, allowing software developers to perform operations on system resources guaranteeing ACID properties (atomicity, consistency, isolation and durability) of the underlying system calls. Furthermore, the vulnerable program is blocked when an exploitation attempt is detected. `TxOS` is open source and publicly available at [132].

Wei and Pu extend their CUU model in [25] by proposing the *Stateful TOCTOU Enumeration Model*. This model lists all the pairs of system calls that can lead to a TOCTOU vulnerability on a Linux and POSIX system (224 and 285 pairs, respectively). To the best of our knowledge, this study is the most comprehensive characterization of the system calls leading to the TOCTOU vulnerability to date. EDGI is also extended to incorporate this model in version 2.4.28 of the Linux kernel.

Rouzaud-Cornabas et al. [114] formalized the concept of race conditions and provided a framework for defining security properties to prevent them. These properties are specified and integrated into a Linux kernel module, implemented on top of SELinux. It is based on information flow graphs to represent the temporal relationships between processes and system resources. This solution was tested in production systems for six months with successful results. However, it is not publicly available.

Vijayakumar et al. [115] introduced a software prototype that stops attacks targeting vulnerabilities based on name resolution (such as TOCTOU) by combining four incomplete defense techniques (specifically, system resource restrictions, capabilities, namespace management, and program resource restrictions) to build a complete solution. It is implemented as a SELinux module in version 2.6.35 of the Linux kernel. Neither the source code nor the tool is available.

Vijayakumar et al. [116] presented a software engine, dubbed `STING`, that prevents name resolution attacks. In particular, it analyzes system calls at runtime and creates test cases that are then used to replicate an adversary's behavior and thereby detect exploitation attempts. `STING` is implemented as a Linux security module in Linux kernel 3.20, and tested with different operating systems, discovering 26 race condition vulnerabilities (21 of them were previously unknown). However, as the authors warn, it can produce false positives under certain running conditions. Unfortunately, the tool is not available for public use.

Different policies are given in [118] to determine which files can be retrieved using the name resolution process in system calls. These policies control file access at run-time in the context of a system call. A prototype that enforces these policies is deployed on top of the SELinux access control module and tested on the Ubuntu 12.04 operating system. The experimental results show that all

exploitation attempts were successfully stopped. However, the policy enforcement prototype is not available.

Vijayakumar et al. [117] also proposed `Process Firewall`, a Linux security module that analyzes system calls and restricts access to resources depending on the current state of the process. These constraints are modeled as Linux IPTable rules and are interpreted by a rule processing mechanism designed for system calls. Tested on Ubuntu 10.04, nine resource attacks (including TOCTOU-based attacks) were detected and blocked successfully. Although it is not mentioned in the article, `Process Firewall` is publicly available at [133].

Kim and Zeldovich [123] introduced a new Linux-based sandboxing mechanism called `Mbox` that interposes on system calls. `Mbox` creates a layered sandbox filesystem on top of the host filesystem where all the operations take place, preventing the latter from being manipulated. The user can then browse the sandbox filesystem, committing the modifications to the host filesystem or discarding them accordingly. The interposition of system calls is carried out using the `seccomp/BPF` facility. `Mbox` is open source and available at [134].

Zhou et al. [121] proposed `SHIELD`, a software that uses deterministic multithreading techniques to guarantee that variables shared across threads are consistent. To do this, when it detects a memory modification, it executes a memory propagation mechanism that extends the modification to the virtual memory of other threads within the program. Like other solutions, `SHIELD` is not available.

Vijayakumar et al. [119] presented `Jigsaw`, a defense mechanism against resource access attacks based on the interception of system calls. It works in two phases: first, it parses the program using graph-based formalisms to find system calls on shared resources; second, it uses `Process Firewall` [117] to enforce the invariants that avoid the vulnerability. This solution is implemented as a kernel module and tested on Ubuntu 10.04, identifying two unknown vulnerabilities. Unlike `Process Firewall`, neither the program application nor its source code is publicly available.

A lightweight Unix-based filesystem sandboxing mechanism is proposed in [124]. The mechanism is dubbed `SandFS` and is designed as an extensible kernel filesystem that intercepts all filesystem requests. It works with low-level kernel objects and provides a C-like API for the developers to implement their own security extensions. Acting as an interposing layer between the filesystem and the user-defined security extensions, it does not perform any filesystem operations, but instead compares them to the extensions. Allowed operations are tracked to the filesystem, whereas denied operations are canceled with the corresponding error number. `SandFS` leverages the `eBPF` framework to achieve safety guarantees and is publicly available at [135].

3.3.3 On TOCTOU Attacks

Figure 3.5 shows the timeline of the articles studied in this dissertation that focus on attacks that try to exploit TOCTOU vulnerabilities. At a glance, the literature on attacks is very scarce. We have found only 4 works that propose new ways

to abuse TOCTOU vulnerabilities, in addition to the seminal work of [93] that first introduced specific examples on how to exploit TOCTOU vulnerabilities and gain elevated privileges. Regarding these offensive works, the oldest and newest of them date from 2005 and 2017, respectively. The remaining are dated from 2009 and 2012.

As commented above, all of them are attacks from the user-space level and dynamic. In summary, we conclude that there is not much innovation in the ways of abusing TOCTOU vulnerabilities and few authors are interested in it. Regarding the attack vectors, the offensive techniques that we found take advantage of: the *trust in external devices* and the *I/O caching mechanism of the system kernel* (explained in Section 3.3.1).

Below we classify the articles found during our systematic review of the literature that contribute to new ways of abusing TOCTOU according to the attack vector they exploit. We describe them according to the research questions set out in Section 3.2.1. As before, they are presented in chronological order. Let us remark that none of these works provide source code or software tools, but instead they provide a detailed explanation of how the attacks work.

Attack Vector Based on External Devices

Mulliner and Michéle proposed in [125] another novel attack called *Read It Twice*, focused on consumer electronics and embedded devices. This attack takes advantage of the installation and update processes of these devices, which normally depend on external devices and are carried out in two steps (not atomic): one to verify and the other to install/update. This attack has been successfully tested on Linux-based Samsung TVs. In addition, the authors develop a hardware board to determine whether a device is vulnerable to these attacks.

Similarly, Lee at al. studied the installation process of of Android applications in [126], finding TOCTOU vulnerabilities in all its phases. As a result, the authors present a novel attack called *Ghost Installer Attack* (GIA), as well as defense solutions against it. We have categorized this paper exclusively as TOCTOU attack because the proposed defense solutions are designed for the GIA.

Attack Vector Based on the Kernel I/O Caching

Borisov et al. presented in [127] a novel technique to exploit race conditions when the defense solution proposed by [92] is working. This attack, carried out by means of three software tools, relies on a deliberate increase in input/output filesystem operations, as they are likely to force the preemption of the running thread due to memory cache buffering issues.

Subsequently, in [128] Cai et al. presented a novel attack that defeats the solutions proposed by [92] and by [102]. This attack is based on collision attacks targeting the kernel's filename resolution algorithm. As a result, the filesystem operations of the vulnerable program are slowed down and thus the window of vulnerability increases.

3.4 Synthesis

In this section we first present a detailed analysis of the results of our systematic literature review to answer research questions RQ2 through RQ5. We then highlight the future research trends and directions that we envision and finally discuss the limitations of our work.

3.4.1 Discussion of Results

Below, we address and answer each of the research questions established in Section 3.2.1.

RQ2: Memory Regions of Defensive and Offensive Techniques

After performing the systematic literature review, we have found that defense mechanisms reside either in user-space or in kernel-space. As shown in Figure 3.3a, 60% of the defense techniques reside in kernel-space.

User-space techniques can be applied to a wide variety of programming languages, compilers, and interpreters and are easier to debug. However, they cannot access kernel-level information or mechanisms such as the system's cache, the scheduler, hardware, input/output (I/O), or the inode generation algorithm, which is a major limitation in terms of the scope of the solution.

Alternatively, kernel-space solutions have access to all system components and information. Solutions that work in kernel-space benefit from lower latency when performing certain operations, such as system calls. Despite all this, the kernel-space approaches may require modifying the kernel or adding modules, which is not always a viable option. If the kernel can be modified, serious backward compatibility issues can arise, even rendering older software or kernel versions unusable. Furthermore, implementing and debugging solutions in kernel-space is not only more difficult than its user-space counterpart, but is also limited to the kernel language. Moreover, kernel-level errors are likely to crash the entire system.

As for offensive techniques, all of them are attacks from the user-space level. This result would be expected, because otherwise, if the attacker can already execute code in the kernel, there is no motivation to exploit a file-based TOCTOU vulnerability.

RQ3: Time of Vulnerability Detection or Exploitation

According to the moment at which the vulnerability detection is carried out, the defense solutions are broadly divided into static and dynamic techniques. Static detection comprises solutions in which the vulnerability is detected without the vulnerable application running or after it has been run (that is, the detection happens before or after the execution), while dynamic detection includes techniques capable of detecting the vulnerability when the vulnerable program is running.

As shown in Figure 3.3b, there is a clear predominance of dynamic techniques, with 71.4%.

Static techniques have proven useful in detecting and correcting the vulnerability before it occurs (for instance, during the development phase of the software system). Some of these techniques require the source code of the program to be executed, which is unlikely to happen in most cases. Their main advantages include the fact that they are simple to implement, they do not require modification of the runtime environment, and that the overall system performance is not affected as a result of code analysis. Unfortunately, these techniques only propose solutions to known attacks (that is, to specific vulnerable instruction sequences) which limits their effectiveness. Furthermore, even when the attack is detected, it is not always easy to figure out how to modify the source code of the vulnerable program to avoid the vulnerability, especially in multi-threaded programs.

Other static detection approaches rely on log analysis or execution traces. Therefore, in this case the vulnerability is detected when it has already occurred. In addition, these detection techniques are often unsound because there is a wide variety of factors that influence the exploitability of the vulnerability, such as environment variables or system load.

On the other hand, dynamic defenses protect systems in real time and can thwart exploitation attempts as they occur. The defense is typically performed by an external agent that is integrated into the runtime environment, and hence these solutions tend to incur performance overheads for the system. Alternatively, other solutions require running the vulnerable application in a controlled environment to perform the analysis. Regardless of the type of approach, dynamic techniques are useful for detecting well-known attacks, although they have the ability to store information about the program execution that could be used to identify new attacks. The main advantage of these techniques is that it is not necessary to have or modify the source code of the program, facilitating their adoption in a more general way.

As for the offensive techniques, they are all dynamic since the vulnerable program must be running to exploit it. These techniques use two different attack vectors: external devices (such as USB sticks or SD cards) and the I/O caching mechanism of the system kernel. External devices are abused during the installation process of the vulnerable application, as the attacker can alter or manipulate the sensitive data that can be stored on these devices. In contrast, abuse of the system kernel I/O caching mechanism is done by third-party programs, which force the kernel to perform more I/O operations and thus increase the vulnerability window, which facilitates the occurrence of race conditions.

RQ4: Operating System used by the Techniques

All the defense solutions focus on Unix-like operating systems. In addition, 3 out of the 4 attack techniques are also directed at these operating systems (the remaining attack proposal focuses on Android, which uses Linux as its kernel). The prevalence of the Unix-like operating system is due to the fact that other

operating systems, such as Windows, manage references to files through internal structures similar to file descriptors (for instance, via *handles* in Windows [61]). This means that these operating systems are free of file-based TOCTOU vulnerabilities, although other types of race conditions are still possible (which are beyond the scope of this dissertation).

RQ5: Reproducibility of Source Code or Tools

Figure 3.3c shows the reproducibility of defense techniques. Almost three-quarters of the articles studied are not reproducible, either because no source code or tool is provided or because details are lacking to fully replicate them. In the spirit of open science, experimentation on any proposal should be reproducible to allow others to evaluate, compare, and improve the proposal.

Most defense solutions use some kind of history tracking of filesystem objects' metadata to detect any changes. When they are detected, the solutions apply their logic to decide whether the modification is legitimate or corresponds to an attempted exploitation. Although none of the articles studied specify what underlying filesystem they used to test their proposed defenses, commonly used metadata includes inode, device ID, filename, or file path, among others. Figure 3.6 shows the prevalence of each metadata used by the defense solutions analyzed in this systematic literature review. However, the accuracy of these metadata for detecting file-based TOCTOU vulnerabilities is highly dependent on the underlying filesystem.

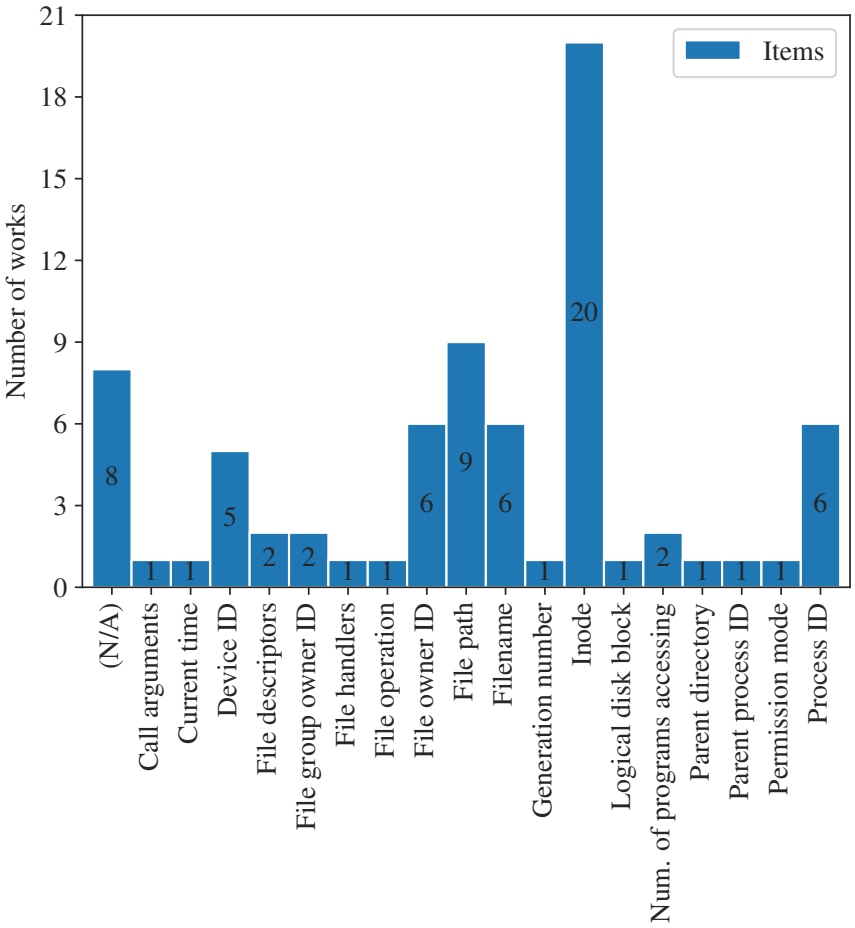


Figure 3.6 Prevalence of filesystem objects' metadata to detect any changes.

For instance, a common metadata is the inode, used by 20 of the 35 defense solutions. An inode (*index node*) is a data structure that defines a file or a directory on a Unix-style filesystem and is stored in the directory entry. To determine if inodes are unique per file, we have studied the behavior of inodes on the main filesystems in the Unix universe [136]. In particular, we have empirically tested if, when an inode is freed (that is, without hard or soft links that point to it), the filesystem eliminates it and never uses it again or if it is free to reuse it when necessary. We have found that the uniqueness of an inode depends to a large extent on the underlying filesystem and, therefore, inodes cannot be assumed to be an item for single distinction. The results of our tests are shown in Table 3.4.

Turning to offensive techniques, we consider two of them are partially reproducible and the other two are no longer reproducible. Based on their level of detail, we consider that [125] and [126] are partially reproducible because, although they do not provide any source code or tool to perform the attack, both works are detailed enough to be replicated. On the other hand, we consider both [127] and [128] as no longer reproducible, as both articles link to their respective source code repositories which are unfortunately no longer available.

Table 3.4 Reutilization of inode according to the filesystem.

Filesystem	Reutilization of inode
BTRFS	No
EXT2	Yes
EXT3	Yes
EXT4	Yes
FAT16	No
FAT32	No
NTFS	No
HFS+	No
JFS	No
NILFS2	Yes
REISERFS	Yes
XFS	Yes
RAMFS	No
TMPFS	No

3.4.2 Future Research Trends and Directions

Most defense solutions protect against specific cases of TOCTOU vulnerabilities, but incur large impacts on performance or make strong assumptions about the behavior of the underlying filesystem. In summary, no defense solution is universal, as reflected in the fact that, until now, no solution has been officially adopted to prevent TOCTOU vulnerabilities.

In our opinion, it is unlikely that a universal solution will be found, given the non-determinism of the TOCTOU vulnerability and the influence of external factors (such as the environmental variables, among others). In any case, the best solution we envision is a mixture of some of the approaches mentioned above. In particular:

- *A new API (or modification of the current one) to provide a race-free, security-focused API*. A good option to avoid TOCTOU vulnerabilities is to use an API based on file descriptors rather than filenames. However, legacy software would still be vulnerable. In addition, the burden falls on software developers, who must know and use this race-free, security-focused version of the API.
- *Modification of the kernel to always work with file descriptors*. Modifying the kernel to work exclusively with file descriptors instead of filenames can also be a good solution. However, this solution implies a drastic modification of the kernel and therefore it is likely to cause serious backwards compatibility issues.
- *Transactional filesystems*. Another good option can be to use transactional filesystems. A transactional filesystem allows the files and directories to be created, modified, renamed, and deleted atomically, protecting the consistency of their filesystem structure. A good solution can rely on this type of filesystem to verify that the file objects do not change between pairs of TOCTOU vulnerable

system calls.

3.4.3 Limitations

Like any other systematic review of the literature, we have defined a search protocol that is reproducible and its results are free of bias. Note that we have considered articles written in English, without considering articles potentially relevant written in any other language. In addition, as we have used the scoring system of the `StArt` tool, our results are tied to that particular scoring system.

Our results are also limited to the keyword bag that we have defined to perform the search. We can also improve these terms to expand our search results. Finally, we have excluded gray literature (e.g., blogs or repositories) as we are exclusively interested in scientific contributions. However, gray literature is an important source of knowledge about issues related to security. For instance, the Openwall kernel patch [137] is a collection of security hardening patches for various versions of the Linux kernel posted on a website.

3.5 Conclusions

Although file-based TOCTOU vulnerabilities were first mentioned in the mid-1970s, they began to be studied in more detail twenty years later. Despite this vulnerability being almost 50 years old, it remains unresolved. In this dissertation, we have presented a systematic review of the literature up to 2021 on defense solutions, as well as related attack techniques, against this type of race condition vulnerability. In particular, we found 41 articles of interest in different scientific databases (in particular, IEEE Xplore, ScienceDirect, Scopus, and ACM).

Our results indicate that a large majority of research efforts have been directed towards defense mechanisms (37 out of 41), whereas a small fraction of works focuses on offensive techniques (the remaining 4). The defense solutions proposed in the literature can be classified into *source code detection*, *post-mortem detection*, *system call interposition*, *memory consistency*, *transactional system calls*, and *sandbox filesystems*. As for the offensive solutions, half of them deliberately force more time-consuming input/output operations, while the rest focus on exploiting the installation of programs from external storage devices.

We found defense solutions that reside in the kernel (slightly above half, 21 out of 35) and at the user-space level (the remaining 14). However, all the attack techniques are carried out from the user-space level. Most of the defense solutions proposed are dynamic (25 out of 35), while the others are static solutions. Static solutions detect TOCTOU vulnerabilities in source code or at the binary level, while dynamic solutions execute, monitor, and verify execution at runtime or after program execution, analyzing logs and audit trials. All the defense techniques are developed for Unix-like operating systems. Similarly, 3 out of the 4 attack solutions focus on Unix-like systems, while the remaining attack focuses on Android.

Finally, we discovered that almost all the software tools developed to defend or exploit TOCTOU vulnerabilities are not available. Few give access to the source

code or the tool itself, or give enough details to code it ourselves, making it difficult to replicate experiments later. For the sake of open science and reproducibility, any contribution that introduces new software tools or new methods should be accessible to the public and other scientific researchers.

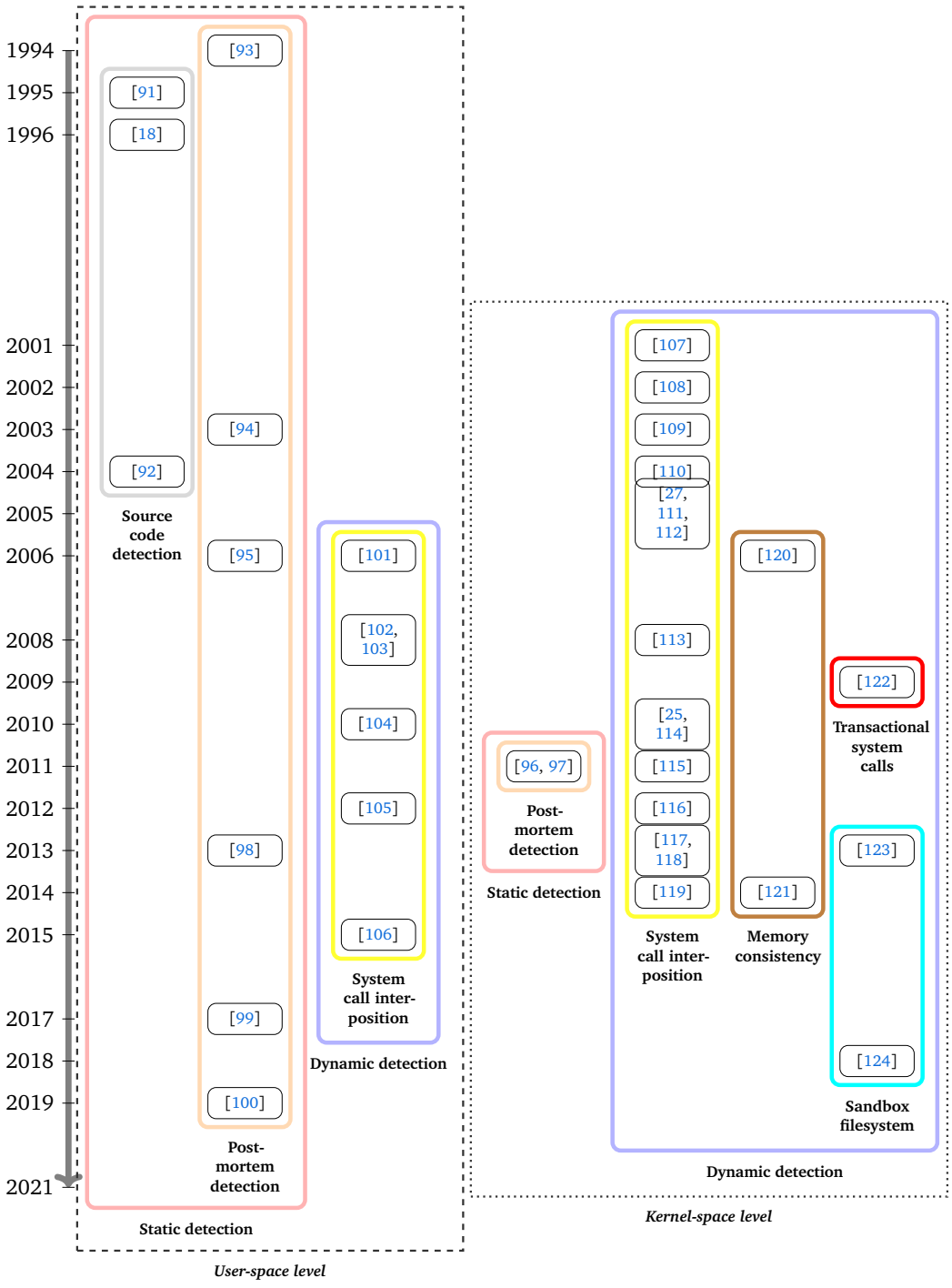


Figure 3.4 Evolution of TOCTOU defenses over the years.

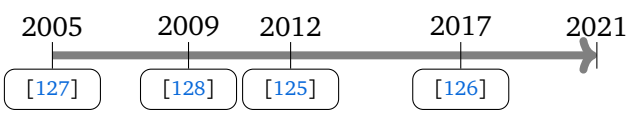


Figure 3.5 Evolution of TOCTOU attacks over the years.

Part II

Dynamic Approach

Introduction

Static analysis is a fundamental technique in software security, used to detect vulnerabilities and assess program structures without execution. By inspecting source code or disassembled binaries, static analysis allows for early detection of potential weaknesses before deployment.

Initially, our research focused on studying a source code vulnerability, specifically, TOCTOU, with the objective of understanding its exploitation techniques and defense strategies, to ultimately develop a mitigation mechanism. The approach we first envisioned involved applying static analysis techniques to detect, assess, and potentially mitigate the vulnerability at the source code level. However, as a result of our initial research, we identified fundamental limitations that made us reconsider our approach. Apart from the constraints inherent to source code analysis, we found out that the complete mitigation of the vulnerability is not feasible (see Section 3.4.1). As a result, we adopted a broader and more flexible approach, ultimately shifting toward dynamic analysis.

Dynamic analysis allows for a more comprehensive assessment of potentially malicious activities by analyzing how software interacts with its execution environment. With this approach, our ultimate goal is detecting patterns indicative of malicious activity in the behavior exhibited by the sample, rather than in the source code. Additionally, this enables us to study a large variety of malicious behaviors instead of specific vulnerabilities.

By adopting dynamic analysis, we address the shortcomings of static approaches and gain deeper insights into malicious software behavior. Observing program execution in real time allows for more effective behavior analysis.

The subsequent chapters of this dissertation examine our research on dynamic analysis environments, the generation of execution traces through sample analysis, and our method for detecting behaviors based on these traces.

Chapter 4

Sandbox Environments

As a first step in our transition from static source-code analysis to dynamic analysis, we conduct an Internet survey to identify tools for analyzing binaries of unknown intent at runtime. We evaluate their features, limitations, cost, and requirements to determine the most suitable option for our research and also for different scenarios other users, researchers, and industry practitioners may face. We focus on sandbox environments, as they are one of the main dynamic analysis tools. This chapter outlines the methodology used for the survey, the sandboxes identified, our evaluation of their features and limitations, and our assessment of their suitability for different scenarios.

4.1 Context

In today's digital landscape, the increasing sophistication of malware has significantly amplified threats to systems and data, making robust detection and mitigation strategies essential. Malware is specifically engineered to compromise confidentiality, integrity, and availability, posing severe risks to individuals, organizations, and governments [1, 2, 3, 4]. The growing frequency and complexity of malware attacks underscores the urgent need for advanced tools to analyze, detect, and neutralize these threats.

Malware analysis—the process of examining a software sample to determine whether it is benign or malicious—relies primarily on *static* and *dynamic* approaches (see Section 2.2.1).

Sandboxes (isolated, controlled environments) are central to dynamic analysis, allowing threat analysts to safely execute and monitor malware behavior (see Section 2.4). These environments restrict harmful interactions with the host system while providing invaluable insights into malware activity. Modern sandboxes leverage advanced technologies such as virtualization, memory forensics, and network monitoring to improve analysis and detection.

In this dissertation we survey and compare ten modern malware sandboxes, spanning both open source and commercial solutions. By evaluating key attributes such as configurability, capabilities, usability, and privacy, we provide potential

users with the information needed to choose the most suitable sandbox for their needs. Focusing exclusively on dedicated malware analysis platforms, we exclude broader security solutions that incorporate sandboxing as secondary element in their analysis workflows [138]. Whether for academic research, enterprise security, or independent analysis, this work serves as a valuable resource for navigating the changing landscape of malware sandbox technologies.

In summary, our contributions are the following:

- We conduct a detailed analysis of ten modern malware sandboxes, encompassing open source and commercial solutions.
- We compare key features such as configurability, usability, capabilities, and privacy to assist in technology selection.
- We offer tailored recommendations based on diverse user needs, including budget, technical expertise, and privacy requirements.
- We highlight limitations like the absence of iOS-specific tools and suggest improvements in transparency and platform support.

4.2 Methodology

To identify modern malware sandboxes, we conducted a systematic literature review [85], incorporating peer-reviewed scientific publications, general-purpose web search engines, and code repositories as sources of grey literature [139, 140]. Following established best practices for systematic reviews [86], we emphasized transparency, replicability, and scientific rigor throughout our investigation. Since many malware analysis tools originate from the commercial industry, similar trends among sandboxes were expected. This section details the methodology, including research questions, search terms, and inclusion and exclusion criteria.

4.2.1 Research Questions

The main objective of this research is to identify modern malware sandboxes by examining both academic and grey literature. Furthermore, the study assesses their accessibility to a broad user base and evaluates their ability to meet the needs of contemporary malware analysts [30, 141]. To achieve this, the following research questions are addressed:

RQ1.- What modern malware sandboxes are currently available, and how can they be identified through academic, grey literature, and general internet searches?

RQ2.- What are the key features, capabilities, limitations, and privacy implications the modern malware sandboxes currently available?

RQ3.- What malware sandboxes are best suited to address the limitations and various scenarios faced by researchers and industry practitioners?

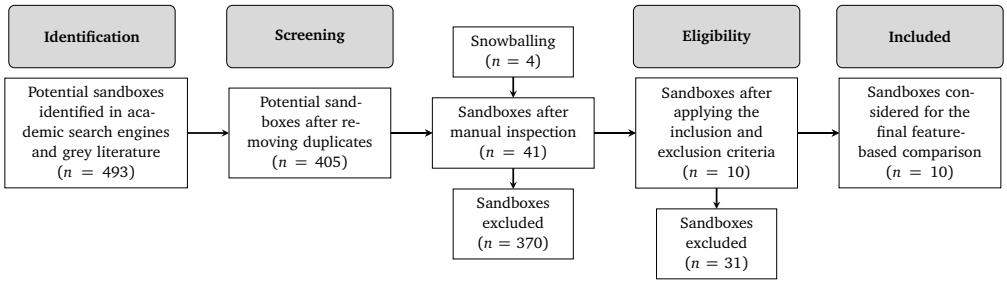


Figure 4.1 PRISMA diagram of our review protocol (horizontal layout).

4.2.2 Search Strategy

In this survey, we aimed to identify all available malware sandboxes by exhaustively searching multiple scientific databases, including the ACM Digital Library, IEEE Xplore, ScienceDirect, and Scopus. To broaden the scope, we also queried major general-purpose search engines such as Yandex, Google, DuckDuckGo, Brave, Bing, and AOL.

The final search query was developed by iterative refinement, starting with basic terms such as (`malware sandbox`) and systematically expanding them to improve relevance. The resulting query is:

> (malware **OR** virus **OR** sample) **AND** (sandbox* **OR** ((virtualized **OR** isolated) **AND** environment) **OR** framework) **AND** ("execution trace" **OR** report **OR** behavior **OR** interactive **OR** sample **OR** scoring **OR** ioc **OR** system **OR** guest **OR** host **OR** config*)

No date constraints were applied; however, non-peer-reviewed scientific results, extended abstracts, patents, and book chapters were excluded from consideration.

This query generated thousands of results on academic platforms such as the ACM Digital Library. To filter these, titles and keywords were manually reviewed, selecting records containing at least three query terms, resulting in an initial corpus of 493 results. After removing duplicates, 405 unique records remained. We then examined each record to determine whether it proposed a sandbox or focused on related methodologies or implementations, which involved a review of abstracts and a brief reading of the text to clearly identify the main contributions. A similar approach was applied to entries from websites, industry articles, and reports, and ultimately 41 sandboxes were identified. After applying the inclusion and exclusion criteria (Section 4.2.3), 10 sandboxes were selected for detailed analysis. Figure 4.1 presents a PRISMA [90] diagram that summarizes the selection process.

4.2.3 Selection Criteria

A wide range of sandboxes emerged from our review. To determine which ones qualified for inclusion in this work, we applied the selection criteria summarized in Table 4.1.

Table 4.1 Inclusion (IC) and exclusion (EC) criteria.

Type	Criterion
IC1	The article or paper presents a sandbox.
IC2	The sandbox analyzes at least Windows-based samples, as Windows remains the predominant target (approximately 90% of malware targets it [3]).
EC1	The sandbox is discontinued†.
EC2	It requires a financial transaction for access.
EC3	Free version is only a time-limited trial.
EC4	The article or result presents a comprehensive framework in which the sandbox just one of many built-in features.
EC5	There is no English version.

†A sandbox was deemed discontinued if its primary website was no longer accessible, the tool was unavailable, or the last update or commit occurred more than two years ago (as of October 2024).

4.3 Survey Results

This section presents the outcomes of our comprehensive survey on modern malware sandboxes, addressing the research questions outlined in Section 4.2.1. The selected sandboxes are identified through a systematic review and categorized into open source and commercial solutions. A detailed feature-based comparison follows, assessing their capabilities, limitations, and suitability for various use cases. These insights provide a clear perspective on the current malware sandbox landscape, guiding users in selecting the most suitable tools for their specific needs.

4.3.1 Identifying and Cataloging Modern Malware Sandboxes

To identify modern malware sandboxing solutions, we conducted a comprehensive survey using academic and general-purpose search engines. Additionally, we manually examined code repository such as GitHub, GitLab, and SourceForge. Through this approach, we aimed to replicate the steps of an uninformed user searching for malware analysis environments by exploring both online resources and academic literature. While this survey provides a broad overview, it is not exhaustive, and the sandboxes analyzed here represent only a subset of the many options available..

After compiling the results from our initial search process, we the inclusion and exclusion criteria specified in Section 4.2.3. This filtering yielded the following sandboxes: `CAPEv2` [13], `Cuckoo3` [142], `DRAKVUF` [143] (including `DRAKVUF Sandbox` [144]), `Noriben` [145], `ANY.RUN` [146], `Hybrid Analysis` [147], `Joe Sandbox` [148], `Tria.ge` [149], `Filescan.IO` [150], and `Threat.Zone` [151]. For clarity, we categorize these sandboxes into open source and commercial solutions. In what follows, we provide a detailed description of each sandbox category.

Open Source Sandboxes

CAPEv2 (often referred to as CAPE) is a widely recognized open source sandbox designed primarily for Windows malware analysis, with limited support for Linux samples. It features API hooking, network traffic analysis, and memory dumping, and handles various file types (e.g., PE files, PDFs, and URLs). While highly customizable and modular, CAPEv2 requires extensive configuration and is best deployed on Ubuntu LTS. It supports interactive sessions, performs malware detection and classification, and includes anti-evasion mechanisms such as sleep deactivation. It is licensed under the GNU GPL v3 and is actively maintained by its developers and community.

Cuckoo3, developed by CERT Estonia, is an open source sandbox designed for malware analysis on Windows 7 and Windows 10. It supports common file types such as PDFs and PE files, but relies on a proprietary monitor, which may limit extensibility. Installation and configuration require Ubuntu 22.04 or higher with Python 3.10, and an online submission service is also available. Malware detection and classification rely partially on third-party anti-malware solutions. Cuckoo3 is licensed under the EUPL v1.2 and has an active community contributing to its continued development.

DRAKVUF is an open source hypervisor-level sandbox designed to evade anti-analysis techniques. It supports sample analysis on Windows 7, Windows 10, and Linux, and offers features such as syscall tracing, kernel heap monitoring, and file access logging via plugins. Installation requires Intel CPUs with VT-x/EPT or ARM CPUs, and it is licensed under the GNU GPL v2. DRAKVUF is regularly updated by a committed developer community.

DRAKVUF Sandbox, developed by CERT Polska, improves on DRAKVUF by introducing a web-based GUI and streamlined installation tools for easier setup. It also features a REST API for broader integration. The sandbox only supports Intel CPUs and is compatible with Windows 7 and Windows 10, running on Debian 11 or Ubuntu 20.04. Like its base project, DRAKVUF Sandbox remains open source and supported by the community.

Noriben is a lightweight, open source sandbox that leverages Microsoft's Sys-Internals Process Monitor for Windows malware analysis. It monitors process, registry, file, and network activity, filtering out extraneous data to improve usability. Designed for a hands-on approach, it encourages user interaction during analysis. Licensed under the Apache License 2.0, Noriben prioritizes simplicity and rapid deployment.

Commercial Sandboxes

ANY.RUN is an interactive malware analysis sandbox offered as a SaaS solution with freemium tiers. It supports a wide range of file types (e.g., executables, Office documents, URLs) and operating systems (Windows 7, 8.1, 10, 11, and Ubuntu 22.04). The free plan limits scans to one minute and 16 MB file size, and results are publicly accessible. Paid plans offer private scans, API integration, and browser extensions. Users can interact with the virtual environment during the

analysis process, and all data is accessible through a web interface.

`Hybrid Analysis`, built with Crowdstrike's Falcon Sandbox, offers static and dynamic analysis for a wide variety of file types and platforms, including Windows, macOS, and Linux. Free users can submit up to 20 files (with a total size limit of 100 MB) for analysis, though interactions are limited to predefined scripts. Paid plans provide access to custom scripts and private analysis. Reports and samples submitted with the free plan are publicly disclosed.

`Joe Sandbox` is a freemium SaaS sandbox compatible with Windows, macOS, and Linux. The free plan allows for limited analysis of up to 15 samples per month, each with a file size limit of 100 MB. Detection and classification are based on behavior, and results are made publicly available. Live interaction and API access are restricted to paid plans, which also offer private analysis. The tool requires user registration and manual approval; however, our registration request (submitted on October 17, 2024) received no response or approval. Despite this, `Joe Sandbox` was included in our study due to its well-known popularity among malware analysts.

`Tria.ge` is a proprietary online sandbox that supports Windows, Linux, Android, and macOS. The free plan offers both static and dynamic analysis, with configurable options such as language settings, Internet connectivity, and timeouts. Users can interact live during the analysis, and generated reports include detection and classification scores. However, results and samples for the free plan are publicly disclosed, while private analysis is only available on paid plans.

`Filescan.IO` is a free, standalone sandbox service powered by MetaDefender. It supports Windows, Linux, and Android and scans multiple file types, including Office documents, URLs, with a file size limit of 100 MB. It also allows zipped files, offers automated API integration, and incorporates community-contributed YARA rules. Free submissions are publicly accessible, though users have the option to hide the sample and keep the analysis report public.

`Threat.Zone` is a hypervisor-level SaaS sandbox that offers a free tier that allows up to 10 scans per day, each limited to 180 seconds and 16 MB, exclusively for Windows platforms. Users can interact with the virtual machine during the analysis; however, private submissions and support for additional platforms (e.g., Android, Linux, macOS) are only available through paid plans. Free tier reports and submissions are publicly disclosed.

Discontinued Sandboxes

Several sandboxes were excluded from this study due to their discontinuation. Nonetheless, these tools have remained influential, often appearing in search engine results and contributing significantly to the evolution of dynamic malware analysis. Although they are no longer actively maintained, some remain accessible and may still be useful for specific use cases.

`Norman Sandbox` was a pioneering malware analysis environment in the early 2000s and continues to appear prominently in search results. However, its identity

has been diluted by integration into various products and companies [152]. Similarly, CWSandbox (later renamed GFISandbox) was one of the first commercial sandbox technologies, which had a notable impact on the industry and influenced many subsequent developments [153].

The original Cuckoo Sandbox [154], introduced in 2011, remains one of the most influential modern open source sandboxes. Its final release (version 2.0.7) was published in 2017, although an online instance maintained by CERT Estonia is still operational[1]. Anubis [155], originally known as TTAnalyze [156], was discontinued in 2016, but was highly regarded during its active years.

Zero Wine was a malware sandbox for Windows, the last version of which was released in 2009 [157]. It was later forked and updated under the name Zero Wine Tryouts, but this version received its final update in 2013 [158].

Several Linux-oriented sandboxes have also been discontinued. Detux [159], designed for Linux malware and network IOC, ceased development in 2018, followed by its fork DetuxNG, which ended in 2022. Other notable Linux-focused sandboxes, including HaboMalHunter [160], Limon [161], and LiSa [162], were discontinued between 2015 and 2022. Similarly, Tamer [163], which specializes in IoT malware and ARM binary analysis, received its last update in 2022.

Finally, Android-focused sandboxes such as DroidBox [164] and DroidHook [165], designed for dynamic analysis of Android applications, stopped being updated in 2014 and 2021, respectively.

Although no longer maintained, these sandboxes played an important role in the advancement of malware analysis techniques and remain historically important in the evolution of sandbox technologies.

Other Discarded Sandboxes

Throughout our survey, we identified several technologies that were overlooked due to the lack of a free version, the absence of Windows analysis capabilities, or because their functionality extends beyond the sandbox. Some of these solutions serve as the primary engine for their freemium SaaS counterparts, such as Falcon Sandbox [166], which underpins Hybrid Analysis, and MetaDefender Sandbox [167], which powers Filescan.IO.

We also found several commercial products that incorporate sandboxing as part of a broader cybersecurity suite. Some examples include: Malcore [168], VMRay [169], Kaspersky Sandbox [170], Deep Discovery [171], Zscaler Sandbox [172], FortiGuard/FortiSandbox [173], Bitdefender Sandbox [174], Spectra Analyze [175], Advanced Wildfire [176], Sophos Sandstorm [177], ThreatAnalyzer [178], Symantec Sandboxing [179], SecondWrite DeepView [180], Trellix Malware Analysis [181] (formerly FireEye AX), and Intezer [182].

Some industry offerings, such as Valkyrie [183], offer a free plan, but lack suffi-

[1] https://cuckoo.cert.ee/

Table 4.2 Evaluated sandbox features.

Feature	Description
	INSTALLATION AND CONFIGURATION
Local deployment:	Indicates whether the sandbox can be installed and operated on user-managed systems.
Online service:	Determines if the sandbox is accessible as a cloud-based SaaS solution.
Configuration required:	Assesses whether extensive setup is needed or if the sandbox is ready to use out-of-the-box.
Payment tiers:	Identifies whether the sandbox is free or freemium.
Automation:	Checks for automated workflows, such as queuing multiple analyses.
API availability:	Evaluates if the sandbox provides APIs for programmatic interaction and integration.
Configurable:	Specifies whether the sandbox can be customized.
Documentation:	Indicates the availability of official documentation.
	CAPABILITIES
Supported operating systems:	Lists supported platforms (e.g., Windows, Linux, macOS, Android).
Interactive sessions:	Indicates whether the sandbox supports user interaction during analysis.
Behavioral trace generation:	Determines if execution traces (e.g., system calls, API invocations) are produced.
Agent type:	Specifies whether the sandbox employs an in-VM agent, distinguishing between user-space and kernel-space implementations, or operates at the hypervisor level (*agentless*).
URL analysis:	Evaluates if the sandbox supports analyzing URLs.
Anti-evasion mechanisms:	Assesses whether the sandbox includes measures to counter evasion strategies.
	IMPLEMENTATION
Distribution model:	Specifies whether the sandbox is open source or commercial.
Backing:	Evaluates support from community, industry, or government stakeholders.
Licensing:	Identifies the sandbox's licensing model (e.g., GPL, proprietary).
	PRIVACY
Privacy:	Indicates whether submitted samples and reports are disclosed or kept private.

cient technical documentation to determine how dynamic analysis is performed. While dynamic analysis is briefly mentioned, its precise function remains unclear. Finally, we identified `esfriend` [184], a lightweight malware analysis sandbox for macOS that is no longer actively maintained.

4.3.2 Key Features, Capabilities, Limitations, and Privacy Implications

We address RQ2 by evaluating each selected sandbox (see Section 4.3.1) against the features listed in Table 4.2. However, complete information is not always available, as certain details remain undisclosed and it can be difficult to infer answers from the source code (when it is accessible). Table 4.3 summarizes our findings, where (✓) indicates that a feature is implemented and (?) denotes insufficient information, often due to trade secrecy. For example, anti-evasion features are rarely disclosed, but are likely implemented in most modern sandboxes to remain competitive.

Selecting the most suitable sandbox for a given context can be challenging. To support this decision-making process, we evaluated each sandbox based on four main criteria: (i) installation and configuration, (ii) capabilities, (iii) implementation, and (iv) privacy. The evaluation was performed by examining the official websites and, where available, code repositories of the selected sandboxes. The identified features are summarized in Table 4.2.

Installation and configuration are critical factors, as some sandboxes require substantial setup. We evaluate whether a sandbox is deployed locally or offered as a remote SaaS service, the complexity of its setup, and whether it operates on a free or freemium plan. Additionally, we consider automation capabilities,

Table 4.3 Features of open source and commercial sandboxes.

	Open Source				Commercial					
	CAPE	Cuckoo3	DRAKVUF	Noriben	ANY.run	Hybrid Analysis	Joe Sandbox	Tria.ge	Filescan.IO	Threat.Zone
Local deployment	✓	✓	✓	✓		✓[$]	✓[$]		✓[$]	
Online service		✓			✓	✓	✓	✓	✓	✓
Config. required	✓	✓	✓	✓						
Free	✓	✓	✓	✓						
Freemium					✓	✓	✓	✓	✓	✓
Automation	✓	✓			✓[$]	✓	✓	✓	✓	
API	✓	✓	✓[•]		✓[$]	✓	✓	✓[‡]	✓	✓[$]
Configurable	✓	✓	✓	✓	✓	✓		✓		
Documentation	✓	✓	✓		✓			✓		
Windows	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Linux	✓		✓[°]		✓	✓	✓	✓	✓	✓[$]
macOS						✓	✓[$]	✓		✓[$]
Android					✓	✓	✓[$]	✓	✓	✓[$]
iOS										
Other OS										
Interactive	✓			✓	✓	✓[$]	✓	✓		✓
URL Analysis	✓	✓			✓	✓	✓	✓	✓	✓
Behavioral trace	✓		✓	✓		✓	✓	✓		
User space	✓	✓		✓	?	?	?	?	?	
Kernel space					?	✓	?	?	?	
Hypervisor level			✓		?		✓[$]	?	?	✓
Anti-evasion	✓	?	✓	✓	?	✓	?	?	✓	?
Detection	✓	✓			✓	✓	✓	✓	✓	✓
Classification	✓	✓			✓	✓	✓	✓	✓	✓
Community-backed	✓	✓	✓	✓					✓	
Industry-backed					✓	✓	✓	✓	✓	✓
Government-backed		✓	✓[•]							
Privacy[⋆]	U	U[†]	U	U	D	D	D	D	D	D

[⋆]D: Disclosed, U: Undisclosed; [†]Undisclosed if deployed locally, otherwise disclosed; [°]DRAKVUF; [•]DRAKVUF Sandbox; [$]Only paid plans; [‡]Free upon approval (may not be eligible); ? Information not available

API availability, and environment customization, which is particularly valuable for samples that require specific conditions to trigger malicious behavior. Finally, we evaluate the availability of documentation, as it is essential for facilitating installation, configuration, and initial use.

Capabilities are also crucial in determining the suitability of a sandbox. Key factors include supported operating systems, types of files that can be analyzed, and

Table 4.4 Suitability of open source and commercial and sandboxes.

Factor	Open Source Sandboxes	Commercial Sandboxes
Budget constraints	Cost-free, ideal for small budgets; supported by active communities, reducing costs.	Premium plans require financial investment; free tiers have limited features and potential privacy issues.
Time limitations	Require significant configuration and maintenance, leading to delays in urgent scenarios.	Ready to use with minimal setup and vendor support, allowing for faster deployment.
Customization requirements	Fully customizable with source code access, making them suitable for advanced workflows.	Limited to out-of-the-box functionality; unsuitable for highly specialized needs.
Technical expertise	Require advanced technical knowledge for setup and troubleshooting.	User-friendly, designed for users with limited technical skills, featuring preconfigured environments.
Privacy concerns	Privacy by default; deployed manually on user-controlled systems.	Privacy ensured only in premium plans, while free plans often disclose samples and reports.

whether the sandbox allows for interactive analysis beyond basic interaction with the VM. We also examine whether the sandbox can generate an execution trace that captures API calls and syscalls, and whether it supports URL analysis, which requires a slightly different approach. Both behavioral and URL analysis provide valuable insights for malware investigations.

Additionally, we consider whether the sandbox uses other analysis techniques and what memory region it operates in, such as user space, kernel space, or hypervisor level (rings 3, 0, and -1, respectively). Anti-evasion mechanisms are also vital, as they determine a sandbox's ability to deal with malware specifically designed to avoid detection. Lastly, we assess whether the sandbox offers built-in sample detection and classification capabilities.

In terms of *implementation*, we distinguish between open source sandboxes, which provide public access to source code, and commercial sandboxes, where access to the source code is controlled by the vendor. We also assess the presence of an active community, as this often indicates constant support and innovation. In addition, we consider the license terms, which can be significant for specific use cases.

Finally, regarding *privacy*, we examine whether submitted samples are publicly disclosed according to the policies set out in each sandbox.

4.3.3 Evaluating Sandboxes for Diverse Research and Industry Scenarios

The suitability of a malware sandbox depends on the intended use case, technical requirements, and resource availability. To address this, we analyzed realistic scenarios influenced by five key factors: (i) budget limitations, (ii) time constraints, (iii) customization requirements, (iv) technical expertise, and (v) privacy concerns. These factors play a critical role in determining whether a commercial or open-source sandbox is more appropriate, as summarized in Table 4.4.

Budget Constraints. Open source sandboxes are often preferred in resource-constrained environments, as they allow malware analysis without significant financial investment. This makes them especially attractive to small businesses, startups, academic institutions, and independent researchers. They are especially useful for education, training, and proof-of-concept experiments, as they benefit

from strong community support, which helps reduce maintenance and development costs. However, when budget is not a constraint, commercial solutions can be equally viable, depending on other factors such as ease of use, support, and advanced features.

Time Limitations. Commercial sandboxes are ideal for time-sensitive scenarios, offering rapid deployment, minimal configuration, and vendor-supported troubleshooting. Their plug-and-play design, user-friendly interfaces, and automated reporting make them ideal for enterprise environments where efficiency is a priority. In contrast, open source solutions typically require significant configuration and optimization, making commercial solutions more advantageous when operating under tight deadlines.

Customization Requirements. Open source sandboxes are often preferred by users who require extensive configurability and transparency, as their publicly accessible source code allows for custom modifications such as adjusting configurations, modifying guest OS conditions for detonation triggers, adding custom hooks, or integrating specialized workflows. This level of flexibility is essential for advanced or highly specific use cases where commercial sandboxes may lack the necessary adaptability.

Technical Expertise. The level of technical competence required is a key factor when choosing between open source and commercial sandboxes. Open source tools often require extensive configuration, deployment, and maintenance knowledge, which is challenging for less experienced users. However, if sufficient time and resources are available for training and troubleshooting, they are still viable options. In contrast, commercial sandboxes, with streamlined interfaces and preconfigured SaaS environments, are often the preferred choices for users with limited technical knowledge.

Privacy Concerns. Privacy considerations differ significantly between open-source and commercial sandboxes. Commercial solutions typically require premium plans for fully private analysis, while free tiers often publicly disclose submitted samples and results, making them unsuitable for handling sensitive data. In contrast, open source sandboxes, when installed on user-controlled systems, ensure privacy by default, preventing third parties from accessing potentially sensitive materials.

Balancing Trade-offs in Sandbox Selection. These differences highlight the diverse needs and preferences of users, as sandbox selection depends on factors such as cost, technical expertise, privacy, and operational priorities. Because each sandbox has its own advantages and limitations, identifying a single, universally *"best"* solution is infeasible. Instead, the optimal choice depends on the specific user context, reinforcing the need to assess individual requirements when selecting the most appropriate technology.

4.4 Discussion

We performed a comprehensive comparison of modern malware sandboxes, highlighting their features, capabilities, and limitations. Our findings emphasize the importance of choosing a sandbox based on specific use cases and requirements, given the trade-offs associated with each solution. This section discusses the major findings of our study and examines its limitations.

4.4.1 Major Outcomes

In the following, we examine the findings derived from the three research questions defined in Section 4.2.1.

RQ1: What modern malware sandboxes are currently available, and how can they be identified through academic, grey literature, and general internet searches?

This research identified ten modern malware sandboxes through a systematic review of academic publications, grey literature, and online repositories. The identified sandbox environments include open source solutions such as `CAPEv2`, `Cuckoo3`, and `DRAKVUF`, as well as commercial platforms such as `ANY.RUN` and `Hybrid Analysis`. The selection process involved a comprehensive search strategy, refined through inclusion and exclusion criteria, ensuring the identification of diverse and relevant tools. The findings underline the variety of solutions available and highlight the importance of structured searches to identify suitable sandbox environments for different malware analysis scenarios.

RQ2: What are the key features, capabilities, limitations, and privacy implications of the modern malware sandboxes currently available?

The evaluation revealed significant variation across sandboxes in terms of configurability, usability, and privacy. Open source solutions offer a great deal of customization but require considerable expertise for setup and maintenance. In contrast, commercial sandboxes offer ease of use but often compromise privacy in free tiers. Many sandboxes are restricted to specific operating systems or file types, limiting their applicability. While anti-evasion capabilities are implemented in most sandboxes, details about these features are often not disclosed. Privacy concerns are particularly relevant in SaaS-based models, where free-tier analyses are publicly disclosed, making these solutions less suitable for handling sensitive samples.

RQ3: What malware sandboxes are best suited to address the limitations and various scenarios faced by researchers and industry practitioners?

Selecting the most appropriate sandbox depends on budget, technical expertise, and privacy requirements. Open source tools are cost-effective and ideal for users requiring advanced customization, although they demand significant technical expertise. In contrast, commercial solutions are better suited for enterprise environments, as they offer plug-and-play deployment and vendor support to

improve efficiency. Privacy-conscious users may prefer locally deployed open source sandboxes to protect sensitive data, especially when SaaS-based models publicly disclose free-tier analytics. These findings highlight the importance of aligning sandbox selection with specific user needs and operational priorities.

4.4.2 Limitations

Although sandboxes are among the most commonly used dynamic analysis tools in modern malware research and cybersecurity, their limitations can affect their accuracy, effectiveness, and overall usability.

First, sandboxes often lack realism, as they may not fully replicate real-life user environments, including specific hardware configurations or authentic network traffic patterns (some of the factors that can indicate the presence of a sandbox to malware). Second, some samples require specific execution conditions, such as particular network services or command-line arguments, which a sandbox may not provide. Third, advanced malware frequently employs defense evasion tactics, detecting sandbox environments and either terminating execution or exhibiting deceptive behaviors to avoid detection.

Time constraints also present a significant challenge, as sandboxes typically run scans for a fixed period (2 minutes is considered enough for the vast majority of cases [185]), making it difficult to observe malicious behaviors that are triggered over extended periods. Furthermore, sandboxing methods, particularly those using virtual machines or hypervisors, require substantial CPU, memory, and storage resources, leading to scalability limitations and high computational overhead. Privacy concerns also arise with cloud-based sandboxes, as submitted samples and reports may be exposed, potentially compromising sensitive information. Finally, support for IoT and mobile platforms remains limited, as most sandbox solutions are primarily designed for desktop and server environments.

While this study presents a comprehensive overview of sandboxing technologies, it has several limitations. First, the evaluation relies on publicly available resources, which may not fully reveal the inner workings of proprietary solutions, particularly with regard to anti-evasion mechanisms. The absence of an empirical performance benchmark further limits the study, as factors such as execution speed, system resource consumption, and detection accuracy remain untested. Furthermore, the distinction between cloud-based and on-premises sandboxes was discussed, but no direct evaluation of latency, scalability, or network limitations was performed. Security concerns related to open source sandboxes were also not evaluated: many of these tools are maintained by the community, leaving them potentially vulnerable to sandbox escapes, misconfigurations, or outdated components. Similarly, the study does not assess how frequently modern malware employs sandbox detection techniques such as identifying virtual environments, detecting API hooks, or executing delayed payloads, which could impact the effectiveness of different sandboxes.

Another limitation stems from the exclusion criteria used to limit the scope of the study. Discontinued projects, tools not available in English, and time-

limited free trials were omitted, potentially excluding valuable insights from older or region-specific communities. Furthermore, the study does not incorporate feedback from industry professionals, security researchers, or enterprise users, leaving practical aspects such as ease of integration, automation capabilities, and real-world adoption challenges unexplored. Future research should incorporate empirical testing with real-world malware samples, assess the security integrity of open source solutions, and investigate how malware actively evades sandbox detection. Additionally, collecting user feedback through surveys or case studies could provide a more complete understanding of how sandboxes work in practical threat analysis and incident response workflows.

4.5 Related Work

Despite the increasing prevalence of malware and the critical need for effective detection tools such as sandboxes, research on this topic remains limited. Choosing the right sandbox is essential for time efficiency and system security, as installation and configuration can take weeks, and then risk discovering that the selected tool is not suitable. This work provides a comprehensive comparison of modern sandbox technologies while also reviewing previous research on their functionality, reporting quality, and suitability for machine learning applications.

Egele et al. [186] analyzed the automated dynamic analysis tools available at the time, including sandboxes such as `Anubis`, `CWSandbox`, `Norman Sandbox`, and `Joebox`. Their evaluation focused on analysis implementation, techniques, and network support. However, since the study was conducted in 2008, it does not cover modern malware analysis tools or address issues such as configurability and privacy.

Similarly, Neuner et al. [187] compared 16 Android malware sandboxes and evaluated features such as network activity analysis, phone activity monitoring, and GUI-based user interaction simulation. In addition, they tested 10 sandboxes using a set of malware samples, which provided valuable insights into mobile security. However, their study is limited to the Android ecosystem and does not explore cross-platform malware analysis tools.

In [188] a comparison between `DRAKVUF` and `Cuckoo` is presented, highlighting their different approaches to behavioral detection. `Cuckoo` follows an agent-based model, which requires software inside the VM, while `DRAKVUF` operates at the hypervisor level by using virtual machine introspection, eliminating the need for agents in the VM. The study evaluates features such as system call hooking, automation, scalability, and memory snapshots. In addition, [189] compares both sandboxes from an end-user perspective, analyzing configuration complexity, reporting, visualization, runtime, supported file types, and evasion prevention.

In [190] a framework for IoT-specific malware analysis is proposed, combining static and dynamic techniques tailored to IoT architectures. While valuable for IoT security, its applicability is limited to IoT malware and does not generalize to broader sandbox use cases. Similarly, the work in [191] examines Linux-based

IoT malware and compares sandboxes such as `Joe Sandbox`, `Cuckoo`, `Limon`, `Habo MalHunter`, `Detux`, and `Falcon Sandbox` based on resource constraints, supported architectures, and vulnerabilities. However, the study incorrectly categorizes `REMnux`, a reverse-engineered Linux distribution, as a sandbox. Unlike these works, our study evaluates a broader range of sandboxes across a variety of features and scenarios, providing comparative guidance for users with varying needs.

In [192], `Malwr` (based on `Cuckoo`), `Anubis`, and an unspecified commercial sandbox are tested using a malware sample that requires user interaction (e.g., mouse clicks) to detonate. The study evaluates whether these sandboxes can simulate human activity, highlighting challenges in evasion techniques. However, its focus is limited to evasion and does not provide a comprehensive assessment of sandbox capabilities or features.

Similarly, `Anubis` and `Cuckoo` are discussed in [193], where their reports are tested against various machine learning algorithms to assess their suitability for malware detection and classification. While this study offers valuable insights into detection rates, its focus is limited and lacks a broader assessment of sandbox capabilities. In contrast, our work examines a broader range of sandboxes, both commercial and open source, evaluating them across multiple criteria including compatibility, capabilities, and privacy considerations.

In [194] several publicly available sandbox tools were tested against information leakage attacks. Since many sandboxes connect to the Internet for analysis, their IP addresses can be exposed and blacklisted. In addition, specially crafted decoy samples can be used to leak sandbox activity, posing potential security risks.

Other works, such as [195, 196], offer partial comparisons of different sandboxes, but only evaluate a limited set of features rather than performing a comprehensive evaluation.

This study takes a practical approach, as it covers a broader range of sandboxes, including both commercial and open source solutions, and evaluates them based on a variety of criteria such as usability, compatibility, capabilities, and privacy. To our knowledge, it represents the most comprehensive modern comparison of malware sandboxes. Unlike previous works, our study is not limited to specific platforms or sandboxes, but instead employs a broad and systematic search to identify all available sandbox technologies.

4.6 Conclusions

This research provides insights into the current landscape of malware sandboxes by studying available platforms and comparing their features, capabilities, and limitations. While many discontinued sandboxes remain in use, a notable shortage of iOS-specific tools remains, highlighting a critical gap in the field. Open source sandboxes offer great flexibility, modularity, and extensibility, making them ideal for advanced customization. However, they require significant configuration efforts and expertise in creating realistic, stealthy environments to counter

sophisticated malware evasion techniques. In contrast, freemium SaaS solutions simplify usage by providing preconfigured environments, eliminating the need for complex installations. However, they often disclose submitted samples and reports, raising privacy concerns unless premium (i.e., paid) options are used.

Furthermore, commercial sandboxes target enterprise users, prioritizing ease of use and vendor support, but offer limited transparency about their internal implementations, in line with proprietary marketing strategies. These findings underscore the need for privacy-preserving, user-friendly, and platform-diverse sandboxing technologies to meet the evolving demands of malware analysis. Future research should focus on expanding platform coverage, improving transparency, and striking a balance between configurability and ease of use to overcome existing gaps in sandbox technology.

In this dissertation, we selected CAPEv2 [13] as the dynamic analysis environment for our research and experimentation. This choice is based on its configurability and flexibility, allowing modifications to specific components as needed. Additionally, it is an open source technology with an active community and a focus on Windows operating systems. CAPE is also widely recognized in the industry and is considered one of the most comprehensive open source sandboxes.

Chapter 5

Malware Execution Traces

After identifying available dynamic analysis sandboxes for modern malware and choosing our preferred technology, we continue our study of dynamic analysis approaches. With a focus on detecting patterns and behaviors indicative of malicious activity, our next objective is to determine the availability of execution trace datasets for analysis. Execution traces are crucial for behavioral analysis, as they capture the actual runtime behavior of programs, providing insights into how a program interacts with the system (see Section 2.5). Understanding what execution trace data is available is essential for developing and testing our approach while allowing us to study the behavior of previously labeled malicious samples. Additionally, we aim to develop our own dataset using samples analyzed in our environment. This chapter presents the development of MALVADA, a flexible framework designed to generate extensive datasets of execution traces from Windows malware, our study of existing execution trace datasets, and the dataset we created, WINMET.

5.1 Context

The rise in cyberattacks involving malware [4, 6] has driven the need for improved detection methods. Several approaches have been proposed [197], including artificial intelligence techniques [198, 199, 200], similarity algorithms [201, 202], and execution signatures [203], among others. These techniques typically rely on malware execution traces that have been previously captured, often in production systems (when a real attack occurs) or in sandbox environments. Malware execution traces are necessary to effectively train and test these methods, as execution traces reveal the actual behavior of the malware at runtime. However, most available execution trace datasets are simplified or optimized to be more efficient for techniques such as artificial intelligence. This simplification, in turn, frequently removes important contextual information, such as API or system call parameters and return values. Producing large datasets of malware execution traces remains a significant challenge due to the requirement for specialized tools, high resource costs, the risk of errors, and the need for user intervention during malware execution.

Many malware detection proposals have focused on Windows systems due to their widespread use [12] and high appeal to attackers [3]. Behavior-based detection for Windows often involves analyzing *execution traces* (see Section 2.5) to identify patterns relevant to determining the malware family or type.

To our knowledge, there are very few publicly accessible datasets on Windows malware execution traces [204, 205, 206, 207]. These datasets typically consist only of sequences of API names or numerical identifiers, which provide a basic representation of execution. They also tend to include a limited number of malware families and types, which are sometimes grouped together and treated as indistinguishable. Furthermore, this simplified representation often omits critical contextual information such as API parameters, results, processes created, synchronization objects, resources accessed, and communications established. This lack of detail hinders a comprehensive understanding of execution behavior, which is particularly important in malware analysis [30].

There are few tools for automatically generating datasets of malware executions. In [208], the authors introduced a tool that gathers information on malicious behavior from various security reports and analysis sources without executing the programs themselves, resulting in datasets based on secondary information. In contrast, the work in [209], more aligned with the approach discussed here, involves executing malware in a controlled environment to generate Windows system call traces. Specifically, the virtualization-based environment integrates tools to gather information about the system calls invoked and the files used during the execution of malware samples. This information is then stored into a relational database so that it can be translated to different output formats. Additionally, each sample is classified using two labels: one indicating the malware category and another one specifying the malware family. Unfortunately, these category labels are too generic and have a limited semantic meaning, such as “Virus”, “Trojan”, “DangerousObject”, or “Packed”. This tool was used to generate a public dataset, called `AWSCTD`, which consists only of the anonymized sequence of system calls (the name of system calls have been translated to numerical identifiers and their parameters/results removed). In contrast, MALVADA is based on CAPE’s reports, which provide a richer description of the actions involved in the execution of the samples (including information about processes, network communications, synchronization, or the usage of registry, for instance). Besides, modern versions of two labeling algorithms have been used to determine the malware family of each sample, increasing the significance and precision of their classification labels.

We developed MALVADA, a framework designed to generate Windows program execution trace datasets that relies on CAPE Sandbox [13] to execute programs and produce detailed reports. MALVADA filters and processes these reports into traces that include contextual information. Furthermore, these traces are also enriched with metadata, such as the likely malware family the program belongs to, providing a comprehensive dataset. The end result is a collection of traces in JSON, suitable for various malware analysis applications. Our framework allows users to create custom datasets or extend existing ones. Additionally, we also introduce the first version of a dataset generated by MALVADA, called *Windows*

Malware Execution Traces (WINMET), which comprises approximately 10,000 malware execution traces.

5.2 MALVADA

As shown in Figure 5.1, we use the CAPEv2 sandbox [13] (also referred to as CAPE) although our system can be extended to other sandboxes capable of generating traces of the Windows API functions and syscalls executed by a sample. CAPE is an open source sandbox that monitors the execution of Windows programs by hooking Windows-specific API functions and syscalls. It logs information such as function parameters and return values to provide detailed insights into the sample's behavior.

To improve the detection capabilities, we modified the original *CAPE monitor* [210] (dubbed `capemon`) to include additional hooks for certain Windows API functions that are necessary to detect particular behaviors in our catalog. Additionally, we developed a support tool (dubbed `CAPE Hook Generator`) to facilitate the definition of new hooks for `capemon`, which is publicly available on GitHub [211]. For instance, we added hooks for functions such as `GetVolumeInformation` (which malware typically uses to obtain characteristics of device drivers), `Process32First/Next` (which is used to iterate through running processes), or `FindFirstFile` (which is used to iterate over files on disk).

Our system intercepts functions from both the Windows API and the NT API, representing two distinct levels of hooking. This differentiation arises from the complex nature of the libraries that implement these functions in Windows (see Section 2.3 for more details). For example, one might assume that the Windows API function `DeleteFile` internally relies on the NT API function `NtDeleteFile`. However, it actually invokes `NtSetInformationFile` to delete a file. By hooking functions at both the higher-level Windows API and the lower-level NT API, we generate richer behavioral evidence.

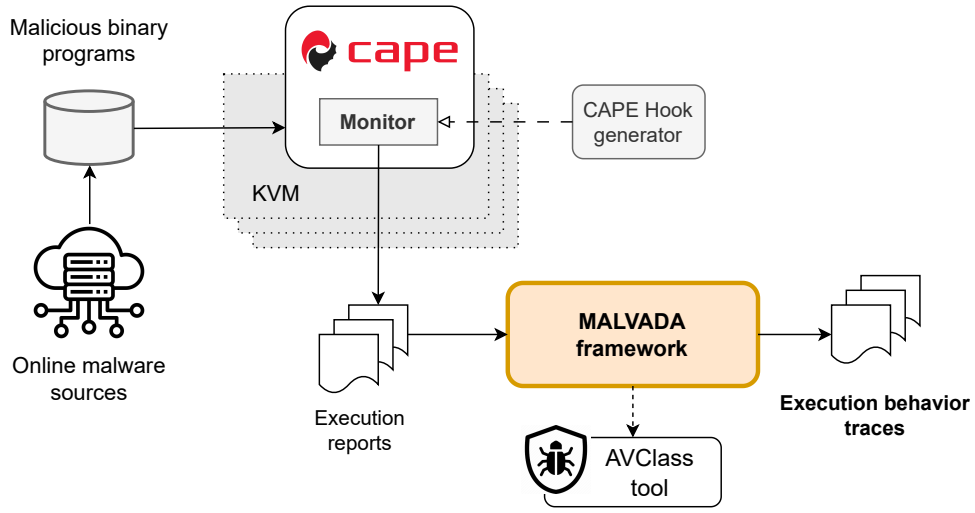


Figure 5.1 Contextual overview of the MALVADA framework.

We use Kernel-based Virtual Machine (KVM) technology to deploy virtual machines (VMs) to run malware samples with CAPE. Each VM generates a report with key events and artifacts from the dynamic analysis, which is then processed by MALVADA. For medium-sized sample collections, we recommend a multi-VM setup. In our setup, we used four VMs on an Ubuntu 22 host, each running Windows 10 x64. Using this setup, we analyzed over 20,000 samples. We discarded incorrect executions caused by errors, crashes, or connectivity issues with the sandbox environment. The remaining reports were then processed using MALVADA, resulting in the creation of the first version of WINMET.

MALVADA processes CAPE reports to extract key data for understanding malware execution behavior. It generates detailed execution traces for each report, including the process tree, API call sequences, contextual information, accessed operating system resources, and mutex synchronizations, among others. Each report contains VirusTotal labels [212]. These labels are used to assign each trace to a malware family by applying two labeling algorithms: CAPE's algorithm and AVClass [213].

5.2.1 Software Architecture

Figure 5.2 shows the architecture of MALVADA, designed as a modular pipeline for processing and generating datasets from input reports. This modular design improves its maintainability, extensibility, and adaptability. Each task in the pipeline can be executed independently, allowing users to customize phases or configure different implementations. The tasks and control algorithm are implemented in Python, and data input and output are handled in JSON format.

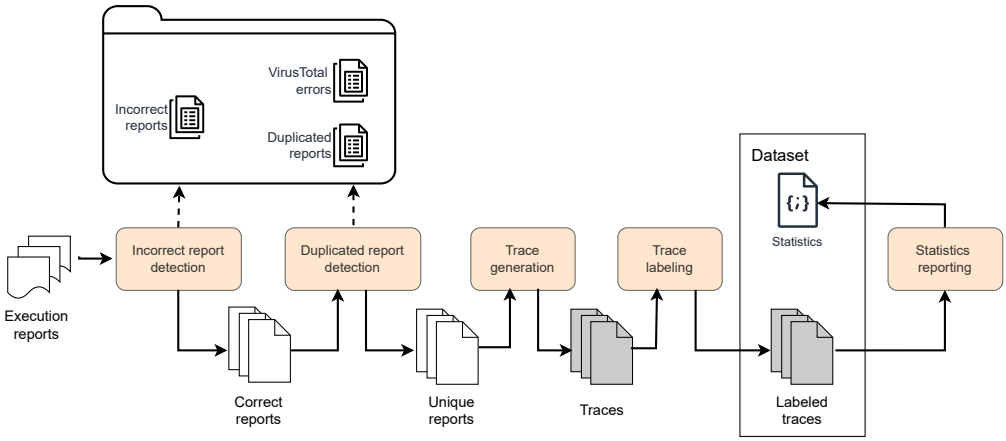


Figure 5.2 Internal architecture of MALVADA.

MALVADA automatically processes the CAPE analysis reports given as input in five steps to create a dataset of execution traces. First, it filters out reports of incomplete or failed executions (*incorrect report detection*) and removes duplicates based on sample ID hashes (*duplicate report detection*), retaining only one report per sample. It then transforms the remaining reports into execution traces by extracting and structuring relevant data about the execution behavior and context while anonymizing sensitive information (*trace generation*). The structure of these traces is described in Section 5.3.1. Each trace is then tagged with information about the malware family using results from antivirus engines and tools such as AVClass [213] to standardize classification (*trace labeling*). Finally, the system generates statistics about the processing of the reports and the composition of the final dataset (*statistics reporting*).

5.2.2 Software Functionalities

The framework offers several key functionalities. It includes a configuration functionality to set operational parameters, such as directories for output results, criteria for report duplication, and thresholds for sample classification. The report filtering functionality also records reasons for exclusion of incomplete or erroneous reports and duplicates for the user to review. Trace generation processes CAPE analysis reports into detailed execution traces in JSON format, enriched with malware behavioral characteristics. The framework also incorporates malware family detection using CAPE and AVClass [213] to provide standardized family labels. Traces, labels, and a statistical description of the processing are packaged by MALVADA into a comprehensive dataset. In addition, it provides real-time monitoring of the status and results of each task in the process.

5.2.3 Software Configuration

The source code for MALVADA is publicly available [214]. This section provides a detailed guide on the necessary steps to execute MALVADA.

Name	Type	Description
`json_dir`	string	Directory containing one or more execution reports.
`-w`	int	Number of workers. Default: `10`.
`-vt`	int	Threshold for VirusTotal positives to consider a sample malicious. Default: `10`.
`-a`	string	Replace the terms in the file provided with "[REDACTED]". Default: `terms_to_anonymize.txt`.
`-s`	bool	Silent mode. Default: `False`.

Table 5.1 Configuration parameters of MALVADA.

Folder	Contents
`src`	MALVADA's source code.
`doc`	Documentation for developers rendered in HTML.
`test_reports`	A set of 200 execution reports generated with CAPE. The purpose of these test reports is to check the execution of MALVADA. The folder also contains the expected results after the execution, in order to compare if the tool behaves as expected.
`capemon`	The compiled version of capemon we used.
`cape-hook-generator`	Version 1.0 of CAPE Hook Generator [211].
`WinMET`	Information about WINMET dataset.

Table 5.2 MALVADA's repository [214] structure.

MALVADA processes reports generated by `CAPEv2 Sandbox` [13]. Therefore, installing CAPE, the VMs for malware analysis, and other dependencies is the first recommended step [215]. Additionally, to expand the API calls that CAPE hooks and thus improve the contextual information obtained from a program execution, it is necessary to modify the original CAPE monitor (`capemon`). This involves editing and recompiling the `capemon` source code, written in C [210]. Our tool `CAPE Hook Generator` [211] simplifies this by generating hook code skeletons, which can be then integrated into the `capemon` source code. Once these steps are completed, the environment for executing malware samples and generating reports is ready.

MALVADA works with minimal user intervention. To run it, simply execute the main script (`malvada.py`) and specify the directory containing the reports to be analyzed. Users can also customize certain parameters to fine-tune the tool's behavior. Table 5.1 lists these parameters, along with their type and a brief description. More details are available in our GitHub [214], which contains all the material necessary to execute and test the tool. Table 5.2 summarizes the structure of the repository.

5.3 Dataset Generation

In this section we detail results produced by executing MALVADA. We first describe the structure of an execution trace generated by the framework, and then we provide a detailed example of WINMET, a dataset created with MALVADA.

5.3.1 Structure of an Execution Trace

The JSON document for a trace comprises several fields that collectively detail the execution behavior of the analyzed sample. Listing 5.1 displays the most relevant fields with "..." denoting additional, omitted fields.

The key fields include: sample identification using cryptographic and similarity hashes (line 4); details about the binary file type (such as whether it is a Portable Executable [216] with specific attributes such as imports, exports, and sections; lines 15–18); a process tree that records all processes initiated by the sample (line 47); a sequence of API calls made during execution, including their arguments, return values, and categories (lines 33–46); malware classification labels, as determined by CAPE (line 1) and AVClass [213] (line 53) based on the VirusTotal results (line 24); and a summary of the OS resources accessed by the sample, such as files, registry keys, mutexes, and services (line 48).

The API call sequence is the most crucial element in a trace. Each entry in the `"processes"` array (line 34) represents a process started during execution, identified by a `"process_id"` (line 35). The API calls made by each process are stored in the `"calls"` entry (line 37). For each API call, it includes the name of the API (`"api"`; line 39), the category of the call (`"category"`; line 38), the return value (`"return"`; line 30), and the arguments (`"arguments"` array; lines 41–44), among other data.

Listing 5.1 Main structure of an enhanced report.

```
{ "detections": [{...}],
  "target": {
    "file": {
      "md5": "...", "sha256": "...", "ssdeep": "...", ... // Additional hashes
      "imports": {
        "KERNEL32": {
          "dll": "KERNEL32.DLL",
          "imports": [{
            "address": "0xABADBABE",
            "name": "LoadLibraryA"
            }, ..., // Additional entries per imported function
          ]
        }, ..., // Additional entries per each imported dll
      },
      "pe":{
        "resources": [{...}],
        ... // Additional fields in the "pe" entry
      },
      "strings": [...],
      "virustotal":{
        "scan_id": "...",
        "positives": 13,
        "total": 73,
        "results":[{
          "vendor": "...",
          "sig": "..."
          }, ..., // Additional entries per vendor
        ], ..., // Additional fields in the "virustotal" entry
      }, ... // Additional fields in the "file" entry
    }, ... // Additional fields in the "target" entry
  },
  "dropped": [{...}],
  "behavior": {
    "processes":[{
      "process_id": 1337,
      "parent_id": 31337,
      "calls":[{
        "category": "filesystem",
        "api": "NtOpenFile",
        "return": "0x00000000",
        "arguments":[{
          "name": "FileHandle",
          "value": "0xDEADBEEF"
        }, ...,] // Additional entries per each argument
      }, ...,] // Additional entries per each API or syscall
    }, ...,], // Additional entries per each process
    "processtree": [ ... ],
    "summary": {
      "files": [ ... ], "read_files": [ ... ], "write_files": [ ... ], "delete_files": [ ... ],
           "keys": [ ... ], "read_keys": [ ... ], "write_keys": [ ... ], "delete_keys": [ ... ],
           "executed_commands": [ ... ], "mutexes": [ ... ],
      ..., // Additional fields in the "summary" entry
    },..., // Additional fields in the "behavior" entry
  },
  "avclass_detection": "...",
  ..., // Additional entries in the report
  }
}
```

5.3.2 WINMET

Using MALVADA, we created the WINMET dataset, which currently contains approximately 10,000 execution traces. The top five malware families represented in the dataset, according to AVClass [213], are `Reline` (22.1%), `Disabler` (7.4%), `Amadey` (5.8%), `Agenttesla` (4.8%), and `Taskun` (3.8%). According to CAPE, the top five families are `Redline` (12.4%), `Agenttesla` (10.2 %), `Crifi` (6.3%),

`Amadey` (6.13%), and `Smokeloader` (5.4%). Both labeling approaches are based on labels provided by vendors from VirusTotal. On average, there are 53 labels per report. Additionally, 7% of the samples are labeled as "(n/a)" by AVClass, compared to 26% by the CAPE labeling algorithm. This suggests that AVClass is able to assign a label in most cases. The "(n/a)" label indicates that the respective algorithm could not determine a decision on the malware family.

The WINMET dataset is publicly available at [217], and additional details are provided in our GitHub repository [214].

Creating a dataset that includes all malware families is nearly impossible due to the sheer number of families. However, MALVADA allows for continuous updates and improvements to the dataset by allowing new samples from other families to be analyzed and included, either by the original developers or by third parties. This flexibility ensures that the dataset can be expanded and refined over time.

5.4 Impact

Cyberattacks are growing exponentially and becoming increasingly sophisticated, posing a significant threat to users and organizations. A major challenge in cybersecurity is developing tools that can efficiently detect and mitigate these attacks to minimize damage. These tools often rely on learning from past data, making the availability of specialized, high-quality datasets essential. The rise of artificial intelligence as a detection method has further amplified the need for diverse, large-scale data sets, particularly since models like deep learning require extensive and varied data to perform accurately and reliably.

MALVADA addresses the scarcity of publicly available malware execution trace datasets. While traditional efforts have focused on collecting malware samples (e.g., VirusShare, VX-underground, Malware Bazaar, and MalShare malware repositories, among others), our framework enables the creation of datasets by processing the reports generated from sandbox environments such as CAPE.

MALVADA offers several advantages for researchers too. Its modular design allows tasks in the process chain to be easily modified and extended, enabling new functionalities and improved reporting processing. The framework is easy to use and requires minimal intervention, and no specialized technical knowledge, making it accessible to users from all backgrounds. Furthermore, datasets can be created incrementally and combined with others, allowing collaborative and progressive development of a complete reference dataset..

In this sense, WINMET [217] represents a significant advancement in malware research [218]. Unlike other public datasets, WINMET provides a wide range of malware behavior characteristics, including detailed process information, API calls, parameters/results, resource access, and synchronization details. This comprehensive dataset enables in-depth analysis of malware operations and interactions, making it a valuable resource for developing effective detection methods. Its size and detailed data provide a robust foundation for building widely accepted reference datasets, and its JSON format simplifies data interpretation and con-

version, facilitating its use in current detection technologies such as those based on machine learning.

Both MALVADA and WINMET are open source and publicly available [214, 217], aligning with the principles of open science and aiming to facilitate their widespread use by the research community.

5.5 Conclusions

In this chapter we presented MALVADA, a framework for generating malware execution trace datasets from sandbox reports (specifically, CAPE), enhanced with classification tools. Through its modular approach, MALVADA facilitates the generation of high-quality, large-scale datasets, addressing the gap in the availability of public malware execution data. Executing samples in CAPE integrates contextual information, including API call parameters, process trees, and resource interactions, allowing us to generate a more comprehensive analysis of malware behavior compared to existing datasets.

The increasing reliance on artificial intelligence-driven malware detection methods underscores the necessity for robust, diverse, and detailed datasets. MALVADA addresses this need and supports the development of improved detection models. Its automated processing pipeline ensures minimal user intervention while maintaining the integrity and reliability of the generated datasets. Furthermore, MALVADA's open source nature fosters collaboration within the cybersecurity research community, enabling further improvements and adaptations to emerging challenges.

We also release the WINMET dataset, aiming to provide a valuable resource for researchers to collaboratively extend and contribute to a comprehensive reference dataset for the malware research community. WINMET represents a tangible outcome of this research, providing a valuable resource for further exploration and model development.

At the time of writing, we are actively working on improving both MALVADA and WINMET. Specifically, we are exploring alternative labeling algorithms to integrate into MALVADA, or querying external services (e.g., VirusTotal) to assign a family label to each processed report, therefore producing fully labeled execution traces. Additionally, we are developing the second version of WINMET, which we estimate will include over 20,000 execution traces and provide both the malware family and the malware type labels.

Chapter 6

Detecting Behaviors at Runtime

After transitioning to dynamic analysis and studying available analysis environments, we began constructing our own execution trace dataset, which remains an ongoing effort. As the dataset evolves, our objective is to leverage it to validate our approach for behavior pattern identification. By detecting and quantifying these patterns, we aim to assist analysts in understanding program behavior and identifying potential malicious activity.

This chapter presents MALGRAPHIQ, a dynamic analysis system designed to enhance the interpretation of unknown Windows binaries by leveraging the Windows Behavior Catalog (WBC). The WBC is a new repository of behavior patterns inspired by the MBC (see Section 2.6), systematically cataloging key APIs and system calls used by Windows binaries to exhibit specific behaviors.

MALGRAPHIQ analyzes execution traces generated using the selected sandbox environment (CAPEv2) to detect and quantify behaviors in programs, regardless of whether they are benign or malicious. It also provides graph-based visual representations of these behaviors, simplifying the interpretation of program actions. To assess its effectiveness, we apply the system to multiple malware families and validate its results through cross-validation. Our evaluation demonstrates MALGRAPHIQ's ability to identify distinct actions and behavior patterns across different malware samples. The results indicate that MALGRAPHIQ effectively detects behavior patterns and distinguishes between malware types, achieving an accuracy of up to 0.96 and an F1 score of 0.92. These findings highlight the potential of our approach in malware detection and program behavior analysis.

6.1 Context

Attacks and security breaches caused by malware are relentless and continue to increase over time [3, 4]. At the same time, malware is becoming increasingly sophisticated, as noted by several industry reports [5, 7]. In this perpetual battle against malware developers and their ever-evolving attack techniques, both researchers and industry professionals must develop advanced tools to stay ahead of cybercriminals.

The main distinction between benign applications (also known as *goodware*) and malicious applications lies in the behavior they exhibit while running. Specifically, it is the actions the application performs and their impact on other components of the underlying operating system, such as the registry, other processes, or files on the disk, that differentiate them. To infer an application's behavior, several binary program analysis techniques are available [219]. These techniques include analyzing an application's assembly code through static analysis, which often involves reverse engineering, examining changes in the system state before and after the program's execution through dynamic analysis, or executing the program within a controlled environment, such as a *sandbox*. In essence, security researchers seek to determine an application's behavior by monitoring and capturing its actions [30]. The captured behavioral data is then analyzed to assess the application's potential threat to the system and its components, ultimately measuring its impact on the system's security attributes. A useful resource in this process is the MBC [73], created by MITRE, which catalogs malware objectives and behaviors to aid in malware labeling, similarity analysis, and standardized reporting.

Modern operating systems provide a set of functions, known as *application programming interfaces* (APIs) and system calls (*syscalls*), that facilitate interaction with their internal software and hardware resources. Malware often abuses specific APIs and syscalls to carry out its malicious activities on compromised systems. This dissertation focuses on the Windows Operating System (Windows, for short), which remains a predominant target for cybercriminals [3]. For instance, on Windows, a keylogger could use the `SetWindowsHookExA` [220] API to install an application-defined hook procedure on any thread running in a desktop user session. However, while APIs can be statically imported into a program and then resolved during load time, their mere presence in an application, as detected through static analysis, is not always indicative of malicious behavior. APIs can be part of infeasible execution paths, leading to false positives. Dynamic analysis, on the other hand, provides a more accurate picture by monitoring which API functions and syscalls are actively called during execution.

Our contribution in this regard is twofold. First, we introduce the *Windows Behavior Catalog* (WBC), an extension of the MBC that identifies key API functions and syscalls used to implement specific behaviors in Windows programs. Second, we develop a dynamic analysis system dubbed MALGRAPHIQ that uses well-known sandbox technologies, specifically CAPEv2 [13]. This system automatically generates graph-based visual representations of each behavior from the WBC identified during the execution of a given Windows program, providing a clearer understanding of the program's activities. These visual representations not only provide insight into the behaviors of individual programs, but also help identify common behavior patterns across different malware families. By improving the ability to visualize and analyze these behaviors, our system provides a useful tool for cybersecurity professionals in malware detection and analysis.

Table 6.1 Windows Behavior Catalog summary.

Micro-objective	Micro-behaviors
FILESYSTEM	Alter Filename Extension, Create or Open File, Copy File, Create Directory, Delete File, Get File Attributes, Read File, Write File, Move File
CRYPTOGRAPHY	Encrypt Data, Encryption Key, Cryptographic Hash, Decrypt Data
COMMUNICATION	Socket, HTTP, WinINet
MEMORY	Allocate Memory, Change Memory Protection
PROCESS	Create Process, Create Thread, Create Mutex, Check Mutex, Resume Thread, Suspend Thread, Enumerate Threads, Open Process, Open Thread, Process Enumeration
OPERATING SYSTEM	Environment Variable, Registry

6.2 The Windows Behavior Catalog

The Windows Behavior Catalog (WBC) is a central component of our analysis approach, providing a systematic method for identifying behaviors in Windows programs. The WBC consists of sequences of Windows API functions and syscalls, which together define specific *behavior patterns*. The primary goal of WBC is to identify these patterns, helping to uncover evidence of program behavior, whether benign or malicious, during the analysis of unknown programs. The WBC is based on MITRE's MBC v3.2 [73], which serves as a foundational reference for defining these behavior patterns and the hierarchy of the catalog (see Section 2.6).

In the WBC, each micro-behavior defined in the MBC is implemented using at least one associated method. These implementations are compiled using the Visual Studio compiler and then run in our dynamic analysis environment, CAPE, to generate baseline *call-graphs* that represent the corresponding behavior. A call graph is a visual representation of the relationships between functions or methods in a program, showing specifically which functions call others during execution. In the context of program analysis, a call graph illustrates the flow of control from one function to another, making it easier to understand how different parts of the program interact [221]. These call-graphs are then used to define the behavior patterns. This approach ensures that each behavior in the catalog is linked directly to executable code, providing consistent and reliable patterns for analysis.

The WBC is freely and publicly available [222], fostering reproducibility and collaboration within the cybersecurity research community. Currently, the WBC comprises 6 micro-objectives, 30 micro-behaviors, 87 methods, and 329 behavior patterns. The complexity of these behavior patterns varies and ranges from a single node up to 7 nodes. Below, we describe the construction of the WBC in detail.

6.2.1 Construction

We summarize the WBC in Table 6.1. Each micro-objective has multiple micro-behaviors, directly inspired by the MBC definitions, which we implemented using at least one method to establish a baseline for graph-pattern matching. This implementation is the simplest practical example of the associated behavior, avoiding polluting the resulting execution traces with unnecessary Windows API functions or syscalls.

The WBC maintains the same hierarchy and elements as the MBC, with methods ranging from specific Windows API functions to more generic operations. For example, the "Create or Set Registry Key" method under the "Registry" micro-behavior (from the OPERATING SYSTEM micro-objective) is implemented using different Windows API calls, such as `RegSetKeyValue`, `RegSetValue`, and `RegSetValueEx`. These methods result in behavior patterns such as `NtCreateKey` → `NtSetValueKey` → `RegCloseKey` or `RegSetValueEx` → `RegCloseKey`, where → denotes chronological transition from the left to the right node. By including all of these variations, the WBC captures the different ways a behavior can be exhibited, ensuring comprehensive detection of relevant actions.

During the generation process, we refine the extracted behavior patterns by removing those that lack operational significance. For example, in our experiments we found a communication pattern such as `WSAStartup` → `closesocket`. However, this pattern lacks operational meaning as it does not represent a complete or logical communication sequence, such as establishing, using, and closing a connection. Such incomplete patterns do not provide valuable insights into the program's actual behavior and can therefore be discarded.

While we can easily refine patterns from known implementations, analyzing unknown programs requires a more conservative approach. Consequently, when working with unknown programs, we retain all Windows API functions and syscalls, discarding only those behavior patterns that we empirically discovered to have no meaning. This conservative strategy ensures comprehensive coverage during analysis, minimizing the risk of overlooking behaviors that could be relevant to detecting malicious activity.

6.2.2 Notation and Definitions

In this section we introduce the formal definitions of behavior graph, function node, category graph, and behavior pattern.

Definition 1. A *behavior graph* $\mathcal{G}_{\mathcal{E}} = \langle F, s, E \rangle$ is a directed graph representing the invocation sequence of API or syscall during the execution of a process $\mathcal{E}$, where:

- F is the set of *function nodes*, where each node represents an API or syscall invoked during $\mathcal{E}$;
- $s \in F$ is the *starting node*, representing the first function invoked during execution. It is denoted by a double circle in the graph; and

- E is the set of directed edges, represented as ordered pairs of nodes connecting elements in F, i.e., $E \subseteq F \times F$.

The set of predecessors and successors nodes of $v \in F$ are defined, respectively, as:

- ${}^\bullet v = \{u \in F \mid (u, v) \in E\}$, which represents all nodes with edges leading to v, indicating the functions invoked prior to v during execution.
- $v^\bullet = \{u \in F \mid (v, u) \in E\}$, which represents all nodes with edges originating from v, indicating the functions invoked after v during execution.

The incidence function $\phi : E \to F \times F$ maps each edge to an ordered pair of nodes, defining the relationship between two function calls. Let $e_{i,j}$ be a directed edge connecting nodes v_i and v_j, such that $\phi(e_{i,j}) = (v_i, v_j)$. This relationship is represented as $v_i \to v_j$, meaning that function v_i is followed by function v_j during process execution.

The *category* of each function node allows similar functions and system calls to be grouped into domains (e.g., file system, networking) to facilitate structured behavior analysis. More formally:

Definition 2. A function node $n \in F$ belongs to a category $c \in \mathscr{C}$, where $\mathscr{C}$ is the set of function categories. The function $\textsc{Cat} : F \to \mathscr{C}$ maps each function node to a specific category.

A category graph $\mathscr{G}_{\mathscr{C}\mathscr{E}} = \langle V, s, E \rangle$ represents the execution of a process $\mathscr{E}$ in which all nodes share the same category, establishing its *behavioral context*. More formally:

Definition 3. A *category graph* $\mathscr{G}_{\mathscr{C}\mathscr{E}} = \langle V, s, E \rangle$ is a directed graph that represents the execution of a process $\mathscr{E}$, where:

- $V = F' \cup O$ is the disjoint set of function nodes F' such that $\forall u, v \in F', \textsc{Cat}(u) = \textsc{Cat}(v), u \neq v$, and the set of function nodes $O \cap F' = \emptyset$, where $\forall u \in F', \forall v \in O, \textsc{Cat}(u) \neq \textsc{Cat}(v)$. The category of the nodes in F' determines the category of the entire graph, establishing its *behavior context*. The set of function nodes O is graphically represented by a node labeled as *others* and referred to as the *others node*;
- $s \in F'$ is the starting node (i.e., the first function executed by the process), represented graphically by a double circle; and
- E is the set of directed edges, which are ordered pairs of nodes, i.e., $E \subseteq V \times V$. The incidence function $\phi : E \to V \times V$ maps every edge to an ordered pair of nodes.

A *category walk* in $\mathscr{G}_{\mathscr{C}\mathscr{E}}$ is a sequence of edges $(e_1, \ldots, e_{n-1})$ that corresponds to a sequence of nodes $(v_1, \ldots, v_n)$, such that $\forall i \in [1, \ldots, n-1], \phi(e_i) = \{v_i, v_{i+1}\}$, and $v_1 \in \{s\} \cup O, v_n \in O, v_i \in F', i \in [2, n-1]$. This walk is represented as $v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \ldots \xrightarrow{e_{n-1}} v_n$ or simply as $v_1 \to v_2 \to \ldots \to v_n$.

A *category path* is a category walk in which all intermediate nodes and edges are distinct, providing the basis for defining a *behavior pattern*. Let the first node of a category walk w be denoted as n^w.

Definition 4. A *behavior pattern* is either:

- A category path p where $n^p = s$ (i.e., the path starts from the initial function node s); or
- The concatenation of p with another category path q, where $n^p = s$ and $n^q \in O$ (i.e., q starts from the *others node*).

Running Example. Consider the graph shown in Figure 6.2b, which represents a category graph $\mathcal{G}_{\mathcal{CE}} = \langle V, s, E \rangle$ belonging to the *Filesystem* category. In this graph, $V = F' \cup O$, where $F' = \{\texttt{NtCreateFile}, \texttt{NtReadFile}, \texttt{NtWriteFile}, \texttt{NtQueryAttributesFile}, \texttt{NtOpenFile}\}$, $O = \texttt{others}$, and $s = \texttt{NtCreateFile}$. The `others` node represents all functions that do not belong to the current category being analyzed (in this case, *Filesystem*).

The following category paths C_{P_n} are obtained from Figure 6.2b:

$C_{P_1} = \texttt{NtCreateFile} \rightarrow \texttt{NtReadFile} \rightarrow \texttt{others}$

$C_{P_2} = \texttt{others} \rightarrow \texttt{NtWriteFile} \rightarrow \texttt{others}$

$C_{P_3} = \texttt{others} \rightarrow \texttt{NtQueryAttributesFile} \rightarrow \texttt{NtOpenFile} \rightarrow \texttt{others}$

$C_{P_4} = \texttt{others} \rightarrow \texttt{NtOpenFile} \rightarrow \texttt{others}$

Based on these category paths, the resulting behavior patterns B_{P_n} are derived as follows:

$B_{P_1} = \texttt{NtCreateFile} \rightarrow \texttt{NtReadFile} \rightarrow \texttt{others}$

$B_{P_2} = \texttt{NtCreateFile} \rightarrow \texttt{NtReadFile} \rightarrow \texttt{others} \rightarrow \texttt{NtWriteFile} \rightarrow \texttt{others}$

$B_{P_3} = \texttt{NtCreateFile} \rightarrow \texttt{NtReadFile} \rightarrow \texttt{others} \rightarrow \texttt{NtQueryAttributesFile} \rightarrow \texttt{NtOpenFile} \rightarrow \texttt{others}$

$B_{P_4} = \texttt{NtCreateFile} \rightarrow \texttt{NtReadFile} \rightarrow \texttt{others} \rightarrow \texttt{NtOpenFile} \rightarrow \texttt{others}$

The behavior pattern B_{P_1} is identical to the category path C_{P_1} since $n^{C_{P_1}} = s$. On the other hand, B_{P_2}, B_{P_3}, and B_{P_4} are formed by concatenating C_{P_1} with C_{P_2}, C_{P_3}, and C_{P_4}, respectively.

Note that while the `others` node is included in the composition of the behavior patterns B_{P_n}, it is omitted from the final patterns in the WBC. This simplification ensures that the resulting patterns focus only on category-relevant behaviors, which is particularly important for the pattern matching phase (detailed in Section 6.3.1).

Table 6.2 Main categories of our Windows API and syscalls classification (accessible at [223]).

Category	Examples
Files and I/O (Local file system)	`NtCreateFile`, `WriteFile`
Cryptography	`CryptEncrypt`, `CryptDecrypt`
CNG Cryptographic Primitive	`BCryptEncrypt`, `BCryptDecrypt`
Cryptographic Next Generation (CNG)	`BCryptCreateContext`, `BCryptDeleteContext`
Windows Sockets (Winsock)	`accept`, `WSASocket`
Network Management	`WinHttpOpen`, `NetUserEnum`
Windows Networking (WNet)	`WNetOpenEnum`, `WNetEnumResource`
Windows Internet (WinINet)	`InternetOpen`, `URLDownloadToFile`
Memory Management	`HeapCreate`, `NtProtectVirtualMemory`
Processes	`CreateToolhelp32Snapshot`, `NtQueryInformationProcess`
Registry	`RegCreateKey`, `NtQueryKey`
System Information Functions	`GetComputerName`, `GetUserName`
Synchronization	`NtCreateMutant`, `NtWaitForSingleObject`

6.2.3 Function Categories

The categorization of Windows API functions and syscalls, as presented in Table 6.2, provides a foundation for defining the WBC micro-behaviors in a structured manner. This classification is publicly available at [223]. While existing classifications, such as the Win32 API programming reference [59], cover many functions, they often lack comprehensive coverage of older functions or Native API syscalls that are still relevant today. Hence, we developed a comprehensive and extensible categorization of Windows API and syscalls.

6.3 MalGraphIQ: A Graph-Based Behavior Analysis System

This section introduces our dynamic analysis approach, which comprises four main phases as shown in Figure 6.1: ❶ *Behavior Capture*, ❷ *Behavioral Data Processing*, ❸ *Behavior Analysis*, and ❹ *Behavior Visualization*. To promote open science, it is publicly and freely available under GNU/GPLv3 at [224]. Below, we first describe each phase in more detail and then discuss the threat model and the associated challenges of our analysis system.

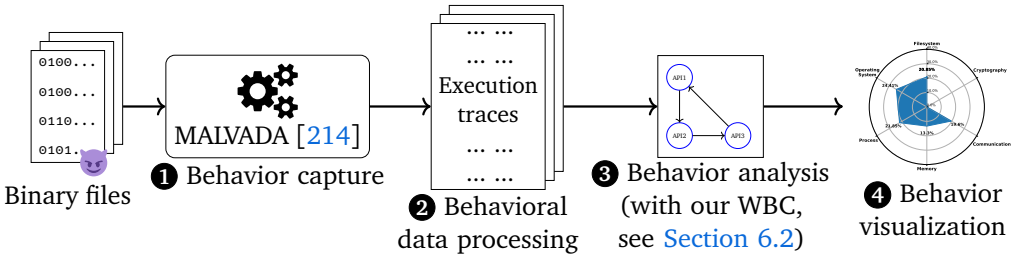


Figure 6.1 Workflow of our dynamic analysis approach (MALGRAPHIQ).

6.3.1 Detailed Description of System Phases

Phase 1: BEHAVIOR CAPTURE

The first phase involves the dynamic analysis of the submitted malware sample. We use our tool MALVADA [214], which leverages a modified version of the CAPEv2 sandbox to analyze the samples and generate the execution traces (see Section 5.2).

The result of this phase is a detailed report of the sample which includes information about the spawned processes, network activity, Windows APIs and system calls invoked (i.e., execution trace), and various resources accessed (e.g., files, registry entries).

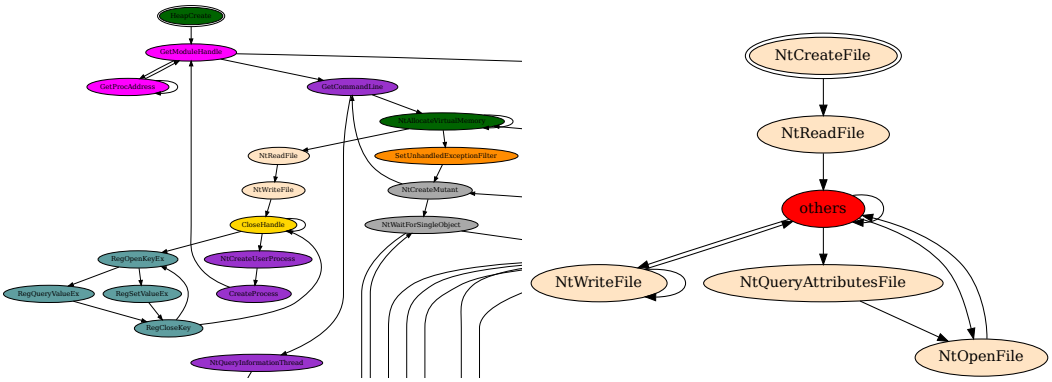
Phase 2: BEHAVIORAL DATA PROCESSING

In this phase, the execution trace generated in the previous step is structured into a more organized form. The system analyzes the execution traces of each process and represents them as a transition matrix. This matrix is then used to construct a behavior graph and the corresponding category graphs.

A *behavior graph* represents the sequence of function calls (for example, API or system calls) during the execution of a program. Nodes represent functions and edges indicate the order in which they are invoked, providing a visual map of the program's behavior. Similarly, a *category graph* is a specialized behavior graph that focuses exclusively on functions within a specific category, such as file operations, networking, or memory management. It groups related functions together and illustrates their interactions, allowing for focused analysis of specific behaviors while ignoring unrelated functions in other categories.

Recall that the categorization of the Windows APIs and system calls used in this dissertation is provided in [223] (see Section 6.2.3). For each graph, we introduce a "*Start node*", represented with a double border, to indicate the beginning of the execution trace. This node simplifies subsequent graph analysis and allows easy identification of the start point. Furthermore, each node in the graph is colored according to the category of the Windows API or system call it represents for easy visualization.

Another node labeled "*others*" is used in the category graphs to represent transitions to/from functions of any other category, reducing complexity and highlighting category-specific behaviors. Figure 6.2a and Figure 6.2b show, respectively,



(a) Cropped section of a behavior graph. **(b)** FILESYSTEM category graph.

Figure 6.2 Examples of the visual representation of an execution trace.

a cropped section of a behavior graph and a FILESYSTEM category graph, both derived from our experiments.

During this phase, functions with string input parameters are generalized to eliminate differences between ASCII and UNICODE versions (e.g., `DeleteFileA` and `DeleteFileW` are treated as `DeleteFile`). This ensures consistent representation and avoids redundancy.

Phase ❸: BEHAVIORAL ANALYSIS

In this phase, we aim to identify behaviors in the analyzed sample by comparing the generated graphs to the WBC. We implement a loop-free Depth-First Search [225] (DFS) backtracking algorithm to compare each of the WBC behavior patterns to the category graphs of all analyzed processes. Recall that a category graph represents the execution of a process in which all nodes share the same category, establishing its *behavioral context*.

The matching process of WBC behavior patterns against the category graphs is divided into three steps: (1) *Pattern feasibility check*: We check if the behavior pattern is viable by verifying that all nodes are present in the graph; (2) *Path identification*: We find all simple paths from the "*Start node*" to the first node of the behavior pattern; and (3) *Pattern matching*: The DFS backtracking algorithm is performed from the first node to the last node of the behavior pattern if it is reachable.

We use the NetworkX [226] library to assist with graph analysis, allowing users to configure the maximum number of intermediary nodes (m) before discarding a path and set a minimum pattern length (ℓ) for path matching. An *intermediary node* is a node that is not part of the behavior pattern but is positioned between two nodes that are. The minimum pattern length ℓ (measured in number of nodes) is used to automatically ignore behavior patterns such that $|\mathscr{B}| < \ell$, where $\mathscr{B}$ represents the set of nodes in the behavior pattern being searched. In our experiments, we enforce strict pattern matching (i.e., $m = 0$ and $\ell = 1$).

The output of this phase is the count of *occurrences* of each micro-behavior from the WBC in the analyzed graph, aggregated from all observed behavior patterns.

Table 6.3 Steps of pattern matching algorithm.

	Step 1	Step 2	Step 3	Result
B_{P_1}	✓	2	2	2
B_{P_2}	✓	1	0	0
B_{P_3}	✗			0

An *occurrence* of a micro-behavior is any simple path identified during the *Path identification* step that matches the specified behavior pattern, based on the configuration parameters m and ℓ. Formally, let $\mathcal{G}_{\mathcal{E}} = \langle F, s, E \rangle$ be a graph and let $\mathcal{M}$ be a micro-behavior composed of a set of behavior patterns, where each behavior pattern B_{P_i} consist of a sequence of n nodes such that $\mathcal{M} = \{B_{P_i} \mid i \in [1, \ldots, n]\}$. The total number of occurrences of $\mathcal{M}$ in $\mathcal{G}_{\mathcal{E}}$, denoted as $Oc(\mathcal{M}, \mathcal{G}_{\mathcal{E}})$, is defined as the sum of occurrences of each behavior pattern B_{P_i} within $\mathcal{G}_{\mathcal{E}}$. That is, $Oc(\mathcal{M}, \mathcal{G}_{\mathcal{E}}) = \sum_{i=1}^{n} Oc(B_{P_i}, \mathcal{G}_{\mathcal{E}})$, where $Oc(B_{P_i}, \mathcal{G}_{\mathcal{E}})$ represents the number of occurrences of the behavior pattern B_{P_i} in $\mathcal{G}_{\mathcal{E}}$.

To illustrate the pattern matching process, let us consider the following simplified example. Given the category graph $\mathcal{G}_{\mathcal{C}\mathcal{E}}$ from Figure 6.2b, suppose we want to match against it the example micro-behavior $\mathcal{M}$, which comprises three behavior patterns: B_{P_1} = `NtOpenFile`, B_{P_2} = `NtCreateFile` → `NtWriteFile`, and B_{P_3} = `DeleteFile`. Given the configuration parameters $m = 0$ and $\ell = 1$, the pattern matching algorithm for each step and pattern is summarized in Table 6.3. In Step 1 (pattern feasibility check), both B_{P_1} and B_{P_2} are indentified as feasible, while B_{P_3} is discarded since its only node is not present in $\mathcal{G}_{\mathcal{C}\mathcal{E}}$. In Step 2 (path identification), there are 2 simple paths (from the Start node s to the beginning of the behavioral pattern $v_1^{B_{P_i}}$) identified for B_{P_1} and 1 simple path for B_{P_2}. Specifically, for B_{P_1} these simple paths are: `NtCreateFile` → `NtReadFile` → `others` →`NtOpenFile`, and `NtCreateFile` → `NtReadFile` → `others` → `NtQueryAttributesFile` → `NtOpenFile`; and given that $n^{B_{P_2}} = s$ a single simple path is considered for B_{P_2}. Let us recall that the `others` node is ignored in the analysis but we retained it in the description for clarity. In Step 3 (pattern matching), B_{P_1} maintains the score since it is a single-node pattern. However, B_{P_2} is discarded since reaching `NtWriteFile` from `NtCreateFile` requires traversing at least `NtReadFile` (i.e., there are intermediary nodes) violating the condition $m = 0$.

The final result of matching B_{P_1}, B_{P_2} and B_{P_3} against $\mathcal{G}_{\mathcal{C}\mathcal{E}}$ is the sum of all their occurrences: $Oc(\mathcal{M}, \mathcal{G}_{\mathcal{C}\mathcal{E}}) = 2 + 0 + 0 = 2$.

Phase ❹: Behavior Visualization

The final phase involves visualizing the results of the pattern matching phase to gain insights into the sample's behavior. The visualization process involves: (1) *Discarding incomplete results*: We discard failed runs, which we define as those with less than 10% of matching WBC patterns. This threshold was determined empirically. (2) *Data transformation*: The remaining data is trimmed to the 90th percentile to remove outliers and then normalized using linear normalization [227]. This ensures that all data points are scaled to the range $[0, 100]$ for

consistent comparison across different micro-objectives. (3) *Plotting*: The transformed data is plotted to illustrate the relative influence of each micro-objective and the contribution of each micro-behavior to these objectives.

Figure 6.3 and Figure 6.4 show examples of the data visualization produced in this phase, helping analysts quickly understand how various micro-objectives and micro-behaviors contribute to the overall behavior of the analyzed sample. The BEHAVIOR VISUALIZATION phase is further detailed and discussed in Chapter A.

6.3.2 Threat Model

Our system operates in the context of dynamic malware analysis. The adversary in this model is a technically skilled malware developer who seeks to evade analysis, fool detection systems, and hide malicious behavior. The adversary has knowledge of our analysis techniques and access to our open source tools, including the WBC. Potential threats to our approach include evasion and anti-analysis techniques, WBC incompleteness, and trace noise. We describe these threats and possible mitigation solutions below.

Evasion and Anti-Analysis Techniques. Our approach relies on behavioral reports generated by a sandbox environment, specifically CAPE. Malware can use evasion tactics such as environmental controls, time delays, and dynamic unpacking to detect and evade the sandbox, resulting in incomplete or inaccurate behavioral reports as the malware refrains from displaying its full behavior [228, 229]. Although sandboxes implement anti-evasion measures, sophisticated evasion techniques can still bypass these defenses, resulting in inaccurate or incomplete behavioral data. To mitigate these risks, we can improve sandbox stealth to increase the likelihood of full malware detonation, ensuring more accurate capture of malicious behavior. In addition, our current approach does not detect behaviors that span multiple categories, limiting our ability to identify complex malicious actions. Improving our system's resilience to these threats requires further refinement of sandbox stealth measures and expanded behavioral detection capabilities across categories.

WBC Incompleteness. MALGRAPHIQ's reliance on predefined WBC behavioral patterns introduces the risk that new or undocumented malicious behaviors may go undetected. As malware evolves, it may employ novel techniques or previously undocumented API calls that are absent from WBC, reducing detection effectiveness. Furthermore, the open source nature of WBC allows adversaries to understand and circumvent its behavioral classifications. Regular updates to WBC are essential for improving detection capabilities.

Trace Noise. Malware may introduce random API calls to obfuscate the trace and camouflage malicious activities. To counter this, our system employs category-graph-level pattern matching instead of matching the entire execution trace, thereby mitigating the effect of unrelated noise. By focusing on category-level behavior, adversarial attempts to hide malicious activities by spoofing unrelated

category calls are less effective. We also support counting and ignore intermediate nodes during pattern matching, which can help detect patterns amid obfuscation. While allowing more intermediate nodes might improve detection, this comes at the cost of increased computational complexity in pattern matching.

6.4 Experimental Evaluation

The main goal of our MALGRAPHIQ is to identify relevant behaviors in unknown binaries to assist users in detecting potentially malicious activities. To validate the usability of the WBC and the overall effectiveness of our analysis system, we evaluate the applicability of MALGRAPHIQ using executions of real malware samples. This evaluation does not involve direct malware analysis or classification but rather focuses on assessing MALGRAPHIQ's suitability for identifying behaviors and correlating them with specific malware families. We conducted a series of experiments designed to answer the following research questions (RQs):

RQ1.- How effective are the WBC and our system in correlating Windows API functions and system calls with specific types of malicious behaviors in Windows malware?

RQ2.- How do the WBC's abstraction levels influence the accuracy and final results of the system?

RQ3.- How accurately can the WBC identify the malware family to which a given sample belongs?

We delve into these research questions by analyzing samples of known malware. This approach allows us to anticipate expected results, validate the system, and, consequently, verify it meets the objectives for which it was designed (described in Section 6.3). Our goal is to assess whether MALGRAPHIQ accurately identifies key behaviors in the samples and whether these insights enable users to detect commonalities and differences across analyzed binaries. Likewise, the results should prove useful at relating samples with the most similar malware family. Although we conducted controlled experiments, we anticipate MALGRAPHIQ to be equally valuable in real-world scenarios where the analyzed binaries are unknown. The generated plots will help users identify exhibited behaviors and determine which are common or unique. The results we present and discuss throughout the experimentation focus on the micro-objective and micro-behavior level.

Below we describe the experimental setup and then address the RQs by analyzing known malware samples. Finally, we discuss the validity of our work, categorizing it according to construct, internal, and external validity [230].

6.4.1 Experimental Setup

We analyzed 249 malware samples spanning four different families: `Gcleaner` (33.33%), `Alina` (26.50%), `Petya` (19.28%), and `Guloader` (20.89%). The samples were collected from dedicated malware databases and from the WinMET dataset [217].

`Gcleaner`, stylized as "`GCleaner`", is a downloader (or loader) malware that attempts to infect the system with other malware by performing actions such as checking file names, opening processes, or attempting to establish Internet connections [231]. `Alina` is a Point of Sale (POS) RAM Scraper that performs actions such as registry modification, process enumeration, or data exfiltration [232, 233]. `Petya` is a ransomware that performs typical actions such as file system traversal, file opening, and file encrypting [234, 235]. `Guloader`, stylized as "`GuLoader`", is a very sophisticated loader that can already include the payload, thus executing malicious actions without the need for an Internet connection. It also incorporates several anti-scanning techniques [236, 237] and performs actions such as process enumeration, file traversal, or file encrypting [238]. These behaviors are expected if the malware is executed correctly, allowing us to observe typical actions of each family.

6.4.2 Discussion of Results

RQ1. Correlation Between API Functions and Malicious Behaviors

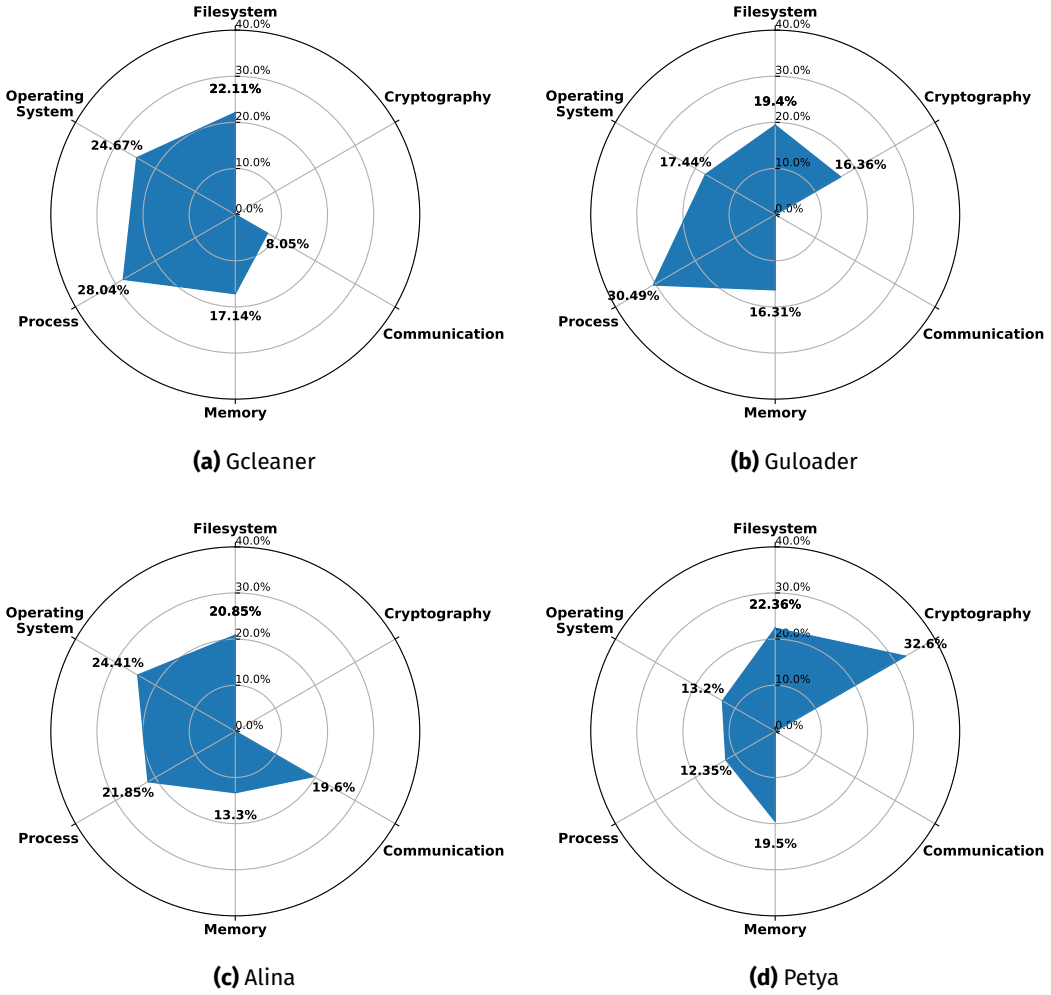
Our system generates two types of behavioral visualizations for each analyzed binary: a radar chart that provides a high-level overview of the micro-objectives from the WBC, and bar charts that present a detailed breakdown of the micro-behaviors within each micro-objective. These visualizations align with the abstraction levels in Table 6.1 and can be generated for multiple samples to infer behavior-related knowledge (Chapter A presents a discussion and some limitations in this regard). For multiple samples, the average number of occurrences is plotted. Figure 6.3 shows the micro-objectives representation of the families included in the experimentation, while Figure 6.4 shows examples of the PROCESS and FILESYSTEM micro-behaviors.

At a high level, the different malware families may exhibit similar behaviors, but differences become apparent when analyzing lower-level micro-behaviors. For example, both `Alina` and `Gcleaner` attempt to establish Internet connections, which is captured by the micro-objectives. `Petya` (a ransomware) and `Guloader` (which includes an encrypted shellcode) invoke the Crypto API, reflecting their need for cryptographic operations (see Figure 6.3).

A deeper analysis of the micro-behaviors reveals clear differences between the samples. For example, while `Gcleaner` and `Guloader` perform mutex-related operations and create processes and threads, only `Gcleaner` matches the "Process Enumeration" and "Open Thread" behaviors, while only `Guloader` matches the "Resume Thread" and "Open Process" behaviors (see Figure 6.4).

Similarly, both `Alina` and `Petya` exhibit around 20% of FILESYSTEM activity at the micro-objective level (Figure 6.3c and Figure 6.3d, respectively), but the micro-behavior representations highlight key differences. `Alina` frequently exhibits the "Alter Filename Extension" feature, while `Petya` rarely does so. In contrast, `Petya` exhibits intensive "Move File" activity, which is absent in `Alina` (see Figure 6.4b).

These results demonstrate that WBC, combined with our system, effectively detects behaviors during sample execution and infers relationships to specific malware



(a) Gcleaner **(b)** Guloader

(c) Alina **(d)** Petya

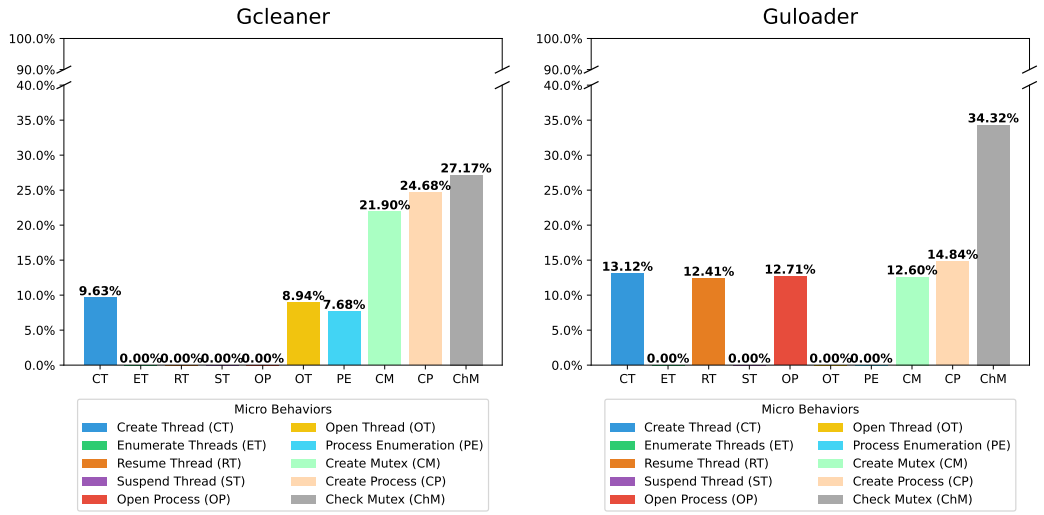
Figure 6.3 Micro-objectives visualization.

families. This approach identifies common actions across samples while also highlighting unique behaviors that distinguish different malware families.

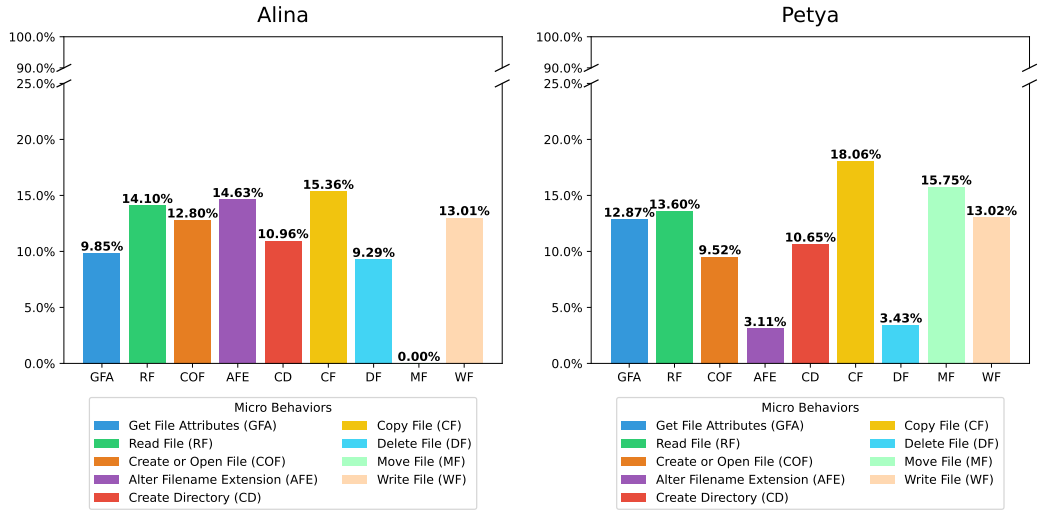
RQ2. Impact of WBC Abstraction Levels on Accuracy and Final Results

To evaluate the applicability of WBC's micro-objectives and micro-behaviors in identifying behaviors associated with different malware families, we transformed the results for each family into a vector, called *behavior vector*. Each vector comprises a representation of the corresponding score for each category. We generated three types of behavior vectors: (1) micro-objectives (6 dimensions), (2) micro-behaviors (30 dimensions), and (3) a combination of both (36 dimensions).

We validate these behavior vectors via a κ-fold cross-validation (κCV) with $\kappa = 5$ and $\kappa = 10$ [239]. Within each fold, malware samples were divided into κ partitions. We used $\kappa - 1$ partitions to generate a reference behavior vector for each family, while the remaining partition was used for validation. We tested the validation partition against reference behavior vectors from all families using three metrics: (1) cosine similarity, (2) Euclidean distance, and (3) Manhattan distance. For simplicity, we focus on the results obtained with cosine similarity



(a) PROCESS



(b) FILESYSTEM

Figure 6.4 Micro-behaviors visualization.

and $\kappa = 10$ for micro-objectives and micro-behaviors. The confusion matrix is shown in Figure 6.5. All confusion matrices generated from our experiments (using different metrics, values for κ, and behavior vectors) are presented in Chapter B.

The results show that using 30 dimensions (micro-behaviors) outperforms 6 dimensions (micro-objectives) in distinguishing between families. Specifically, the classification scores for `Gcleaner` (0.976 vs 0.840), `Alina` (0.939 vs 0.859), and `Guloader` (0.827 vs 0.385) clearly demonstrate this improvement. In the case of `Guloader`, the system had a hard time differentiating it from `Gcleaner` when micro-objectives were used. This was to be expected, as both are loaders and share several behaviors such as FILESYSTEM, PROCESS, and MEMORY, which are captured in Figure 6.3a and Figure 6.3b.

Using micro-behaviors greatly improves the accuracy in distinguishing between these seemingly similar families. Further improvement was seen when using the

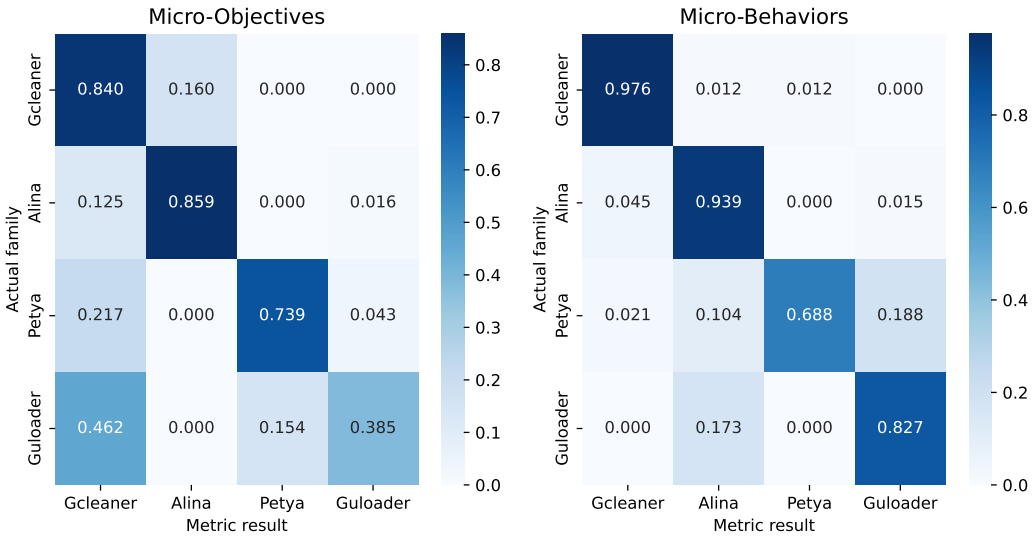


Figure 6.5 Performance using Cosine Similarity and $\kappa = 10$.

combined behavior vector (36 dimensions), with `Guloader`'s score increasing to 0.942, as seen in Figure B.1b.

However, higher dimensions did not always yield better results. For example, in the case of `Petya`, using micro-behaviors reduced the score from 0.739 to 0.688, although using the combined behavior vector improved it slightly to 0.717. Nevertheless, this did not outperform micro-objectives alone. This trend persisted throughout our experimentation with `Petya`, as shown in Figure 6.5. A detailed explanation of this trend would require an in-depth analysis of this malware family, which is beyond the scope of this study.

Based on our experiments, we found that the 6-dimensional micro-objective representation achieved reasonable performance, but was unreliable in distinguishing between similar families. On the other hand, the 30-dimensional microbehavior representation significantly improved performance, with a minimum score of ≈ 0.7. Furthermore, combining both vectors (36 dimensions) further improved accuracy. Thus, we conclude that increasing the number of dimensions positively impacts the accuracy of the system, and that higher dimensionality generally yields better results.

RQ3. Identifying Malware Families Using Behavior Patterns

We evaluated the results of the κ-fold cross-validation obtained in the previous RQ. Specifically, we tested each reference behavior vector against the behavior vector of the validation partition for each family. This approach was used to assess whether the behavior patterns identified in the previous phase can reliably identify similarities between malware families and potential unknown binaries. We tested the same three types of behavior vectors: (1) micro-objectives (μObj; 6 dimensions), (2) micro-behaviors (μBeh; 30 dimensions), and (3) the concatenation of both (*Both*; 36 dimensions). We also employed three similarity/distance metrics: cosine similarity, Euclidean distance, and Manhattan distance. Table 6.4

Table 6.4 Micro-average performance metrics of the κCV.

		$k = 5$		$k = 10$	
	Metric	Acc	$Prec, Rec, F1^{\star}$	Acc	$Prec, Rec, F1^{\star}$
μObj	Cosine	0.87	0.75	0.86	0.73
	Euclidean	0.87	0.75	0.86	0.72
	Manhattan	0.87	0.74	0.86	0.72
μBeh	Cosine	0.93	0.86	0.94	0.88
	Euclidean	0.94	0.88	0.95	0.90
	Manhattan	0.94	0.87	0.94	0.87
Both	Cosine	0.95	0.90	0.95	0.91
	Euclidean	**0.96**	**0.92**	**0.96**	**0.92**
	Manhattan	0.94	0.89	0.94	0.89

*Precision ($Prec$), Recall (Rec) and F1-Score ($F1$) have the same value.

shows the micro-average performance metrics, with more details available in the B. The metrics used are accuracy ($Acc = \frac{TP + TN}{TP + TN + FP + FN}$), precision ($Prec = \frac{TP}{TP + FP}$), recall ($Rec = \frac{TP}{TP + FN}$) and F1-score ($F1 = 2 \cdot \frac{Prec \cdot Rec}{Prec + Rec}$). True Positive ($TP$) and True Negative ($TN$) are, respectively, the number of behavior patterns correctly identified as present in the analyzed malware and the number of behaviors correctly identified as absent. In contrast, False Negative (FN) and False Positive (FP) are, respectively, the number of behavior patterns that are present but not detected, and the number of behaviors incorrectly identified as present in the sample when they are not.

The κCV results suggest that WBC and our system perform well with $\kappa = 5$ and $\kappa = 10$, with the worst accuracy of 0.87 and 0.86, and the worst F1-score of 0.74 and 0.72, respectively. We observed that higher-dimensional behavior vectors consistently outperformed lower-dimensional ones: *Both* vectors performed the best, followed by μBeh, and then μObj. These findings support the idea that higher dimensionality generally improves performance. Among the metrics tested, Euclidean distance performed the best overall, with an accuracy of 0.96 and an F1-score of 0.92 when using the combined vector (36 dimensions). Although increasing dimensionality can lead to better results, it can also increase computational costs in large-scale analysis, which is beyond the scope of this dissertation.

Our experiments demonstrate that WBC and MALGRAPHIQ can achieve an accuracy of 0.96 and an F1 score of 0.92 for identifying malware families when using Euclidean distance with a combined behavior vector, making our approach suitable for behavior identification and family correlation.

6.4.3 Threats to Validity

Construct Validity. Our work aims to identify behaviors using a behavioral analysis system and the WBC. While we perform controlled experiments to tune the system,

the exclusion of function parameters and accessed resources limits its ability to identify specific behaviors. This abstraction can reduce the accuracy the system and overlook malicious activity. To address this limitation, the analysis could be extended to incorporate resource usage and function parameters, thus providing more detailed information.

Furthermore, our system relies on predefined WBC behavioral patterns, which can lead to overfitting, especially when dealing with new or never-before-seen behaviors. As a result, the system's performance may be biased toward detecting known behaviors rather than recognizing evolving attack techniques. Incorporating more adaptive methods, such as machine learning techniques, would enable the system to handle new behaviors and evolving attack strategies.

Internal Validity. Our system relies on CAPE sandbox traces, which may introduce artifacts unrelated to the malware's behavior. Furthermore, effectiveness depends on the malware fully detonating, which does not always occur due to environmental factors or anti-analysis measures. To mitigate these threats, the environment was configured to be stealthy and consistency checks were applied across multiple anomalous sample runs, although direct and automatic verification of detonation was not possible. Improving internal validity could involve stronger verification mechanisms to ensure full detonation.

Higher dimensionality of behavior vectors (e.g., 30 or 36 dimensions) improves accuracy, but may also incur higher computational costs, which could impact the system when applied to large-scale datasets or real-time analysis. In this regard, future work will explore dimensionality reduction techniques and optimization strategies to improve scalability without sacrificing accuracy.

Finally, the system can be affected by trace noise, especially when analyzing more complex programs or those with many background processes. This can potentially affect internal validity by introducing irrelevant data into the analysis, which can reduce the overall accuracy of the identified behavioral patterns. A more advanced filtering mechanism for trace noise and irrelevant background behaviors will be developed to ensure that the system focuses on the most meaningful patterns.

External Validity. Our experiments focused on specific malware families, including loaders, ransomware, and point-of-sale RAM scrapers, using a relatively small sample set. This limits generalizability to other malware types or larger datasets, as the system may not perform equally well with all types of malware in real-world scenarios. Additionally, our analysis is context-specific, relying on API and system calls in Windows. Results may vary across different operating systems, platforms, or with other distance metrics, requiring further research to confirm robustness. Future work should extend the analysis to a broader range of malware families, operating systems, and platforms to improve generalizability.

Similarly, the system's effectiveness against malware that uses evasion or anti-analysis techniques [40, 45, 46, 48, 51, 53] has not been thoroughly tested. Therefore, advanced malware may evade analysis, resulting in inaccurate or incomplete behavioral patterns. To address this limitation, future evaluations

may include malware samples designed to employ evasive techniques to assess the system's robustness.

6.5 Related Work

Previous research has broadly addressed the challenge of identifying behaviors exhibited by unknown applications, primarily focusing on determining whether those behaviors are benign or malicious (*malware detection*). Some approaches go further by attempting to classify which malware type and family an unknown application belongs to (*malware classification*). These goals are typically pursued using four groups of techniques: graph-based, statistical, signature-based, and miscellaneous approaches. Most proposals analyze Windows applications using dynamic analysis techniques, running the applications in controlled environments to extract their execution traces, and subsequently analyzing those traces to detect malware behavior. Below we review each group in more detail.

6.5.1 Graph-Based Techniques

Graph-based techniques transform execution traces or other relevant behavioral indicators into structured graph representations, allowing for systematic analysis to determine whether a sample exhibits malicious behavior or to classify the malware family. The structure and semantics of these graphs depend on the features considered during analysis and the specific graph analysis techniques applied. In any case, these techniques aim to determine whether the graph exhibits characteristics of malicious behavior or, if such behavior is identified, to classify the malware sample accordingly.

Several studies have proposed different types of behavioral graphs to identify malicious activities [201, 240, 241, 242, 243]. Some of these graphs represent dependencies between system calls [201, 240] or API calls [242], while others provide high-level abstractions of system calls organized by functionality [243]. In these studies, graph similarity algorithms are used to compare graphs of unknown programs with reference samples previously classified as malware.

Markov chains, a well-known type of directed graph in graph theory, have also been used to model execution traces and analyze potential malicious behavior. In [241], a modified version of the Ether [244] malware analysis framework is used to trace program instructions, which are subsequently modeled as Markov chains. The similarity matrices derived from these chains are then analyzed to perform both detection and classification. Similarly, in [202], Markov chains are generated from existing datasets of API and system calls, which serve as an alternative for performing dynamic analysis, and similarity algorithms are applied to analyze these chains. Another approach in [245] applies feature selection to Markov chains representing Android app executions before building machine learning models for malware classification.

Other studies use API call graphs as an intermediate representation for artificial intelligence techniques. For example, in [246], graphs are transformed into word

vectors using a graph embedding technique and autoencoders are employed to detect malware. Similarly, in [247], graphs are processed with feature engineering to generate semantic vectors that feed different learning models capable of performing malware detection and classification.

6.5.2 Statistical Techniques

Statistical techniques extract features that describe the behavior of an application from a quantitative perspective, often enabling the use of machine learning algorithms for malware detection. This process, known as *feature engineering*, can involve different formalisms, but typically represents information related to frequent sequences of system calls.

A common formalism for feature extraction is n-grams, as explored in [200]. The authors concluded that a linear regression algorithm trained using stochastic gradient descent and working with 3-grams is the best choice for detecting malicious applications. In contrast, in [248], data mining and feature extraction are combined to generate n-grams representing events related to file system, network, registry activity, and process/thread management. The authors concluded that 4-grams combined with the confidence-weighted linear classification algorithm achieve the highest accuracy.

Another formalism for representing statistical features is association rules. In [199], the Apriori algorithm is used to efficiently discover associations from execution traces. These associations, representing frequent sequences of instructions, are then analyzed with machine learning algorithms such as decision trees and support vector machines, with the latter yielding the best results.

In [249], data vectors represent statistical features related to the frequencies of API calls and the actions taken before and after these calls. This work aims to identify common behavioral patterns in malicious programs rather than simply determining whether an application is malware. The authors apply a data mapping methodology to discover undesirable behaviors, describing six behavioral patterns commonly used by malware.

6.5.3 Signature-Based Techniques

Signature-based techniques use unique identifiers to represent events that occur during application execution, comparing these signatures to known malicious profiles. Signatures are typically created from sequences of API calls obtained through dynamic analysis, augmented with additional information such as resource usage [39] or data flow between calls [203]. These signatures are then clustered by their behavioral representation, using similarity algorithms [203], learning methods [39, 203], or pattern mining [204, 250] to perform malware detection or classification.

In [39], the arguments of API calls and the files accessed are included in the signatures, which are clustered using fuzzy clustering with a Euclidean norm. A similar approach is presented in [203], where execution traces are used to create

signatures containing frequent patterns of API functions and parameters, and clustering is applied for behavior detection.

In [204], DNA sequence alignment algorithms are used to create a signature database. API calls are grouped into representative classes and the sequences are transformed into sequences of call classes, increasing the level of abstraction in the signatures. Multiple sequence alignment and longest common subsequence algorithms are then combined to create signatures stored in a database, allowing unknown programs to be analyzed and compared for malicious behavior.

[250] also processes API call sequences, grouping them into functional categories to increase the level of abstraction before generating signatures using similarity digest algorithms [251]. Similarly, in [252], frequent API call patterns are translated into actions and machine learning models are built using random forests, decision trees, and support vector machines to detect malware based on these action signatures. The authors concluded that decision tree models achieve the highest accuracy.

6.5.4 Other Approaches

Several other approaches use alternative representations to analyze and classify malware behavior. In [253], visual representations are generated from sequences of API calls, using treemap visualizations to represent the frequency of API calls, system functionality, and parameter relevance, and graphical representations to represent the behavior of threads during program execution. These visualizations are grouped together, allowing an analyst to quickly determine whether a sample belongs to a particular malware family or requires further investigation.

The evolution of malware and its functionality over time is studied in [254], where malware samples are allowed to update themselves. Initial and updated versions are then compared by examining changes in the control flow graph, enriched with a system-level activity analysis. This process, called semantic-aware binary matching, helps identify new behaviors and changes introduced in malware samples.

Comparison with Related Work. Like most existing studies, our approach uses dynamic analysis by running applications in a sandbox engine, but our unique contribution lies in modeling the execution as a set of behavior and category graphs. Our approach focuses on detecting previously studied behaviors included in the WBC (see Section 6.2), helping malware researchers and analysts determine the actions of unknown binaries more easily and quickly.

Unlike previous work that focuses primarily on detecting malicious code, we aim to detect fundamental behaviors regardless of intent, by comparing these behaviors to patterns in the WBC. While previous work such as [201, 240, 243] focuses on modeling specific parts of an execution of interest, our approach captures the entire execution, providing a more complete representation of program behavior.

As for detection, [201] leverages taint analysis and models data flow between pairs of system calls, while [240] computes graph similarity using maximum

common subgraph algorithms. Similarly, [243] uses different similarity metrics for detection and classification. Instead, our approach employs a backtracking-based pattern matching algorithm (see Section 6.3) that allows for a more thorough comparison of behavioral patterns.

Furthermore, unlike previous studies, we provide public access to our system implementation and the behavior catalog, promoting reproducibility and transparency of our results.

6.6 Conclusions

In this chapter we introduced MALGRAPHIQ, a dynamic analysis system for identifying behaviors of Windows binaries of unknown origin, complemented by the Windows Behavior Catalog. Inspired by MITRE's MBC, the WBC defines and catalogs behavioral patterns using Windows API sequences and system calls, allowing for the effective identification of both benign and potentially malicious behaviors. Experimental evaluation demonstrated the system's ability to discern between different malware types and families, providing valuable insights into behavioral patterns, similarities, and differences.

We envision the visualizations generated by MALGRAPHIQ to help understanding the behavior of applications, something particularly useful when they are of unknown origin. For malware analysts, the tool can assist in the initial stages of analysis and triage, where it is essential to determine a sample's behavior, its interactions with the operating system, and, ideally, whether it is malicious.

To promote open science, both MALGRAPHIQ and the WBC are publicly accessible at [224] and [222], respectively. Our results indicate that this approach is effective in identifying malware behaviors and enabling behavioral comparisons between samples, ultimately aiding malware detection and classification.

Chapter 7

Conclusions and Future Work

This dissertation aims to answer the research questions stated in Chapter 1. This chapter summarizes the findings obtained for each research question and discusses open problems and potential directions for future work. In what follows, we briefly recapitulate the results of the dissertation.

7.1 Conclusions

This dissertation focused on the identification of behavior patterns indicative of potentially malicious activities that could compromise the system's security. The research initially aimed to detect security weaknesses during the development stage. To this end, we studied a specific source-code vulnerability to understand its causes, manifestations, and the defense and exploitation techniques previously proposed in the literature. Based on the findings, the research shifted to a broader approach centered on analyzing program behavior during execution. To conduct the analysis, we required a suitable dynamic analysis environment, an approach for analyzing execution traces to identify behavior patterns, and access to existing execution trace datasets or the creation of a new dataset. In this sense, we conducted a review of the available dynamic analysis environments, providing a guide that assists analysts in choosing the most suitable environment for their specific needs. Furthermore, we assessed the quality of existing execution trace datasets for behavior analysis. To address the limitations of publicly available datasets, we developed MALVADA, a framework designed to help generating large execution trace datasets. Using MALVADA, we created the WINMET dataset, which comprises nearly 10,000 execution traces of Windows malware. Furthermore, we created the WBC, a structured catalog that contains several categories of Windows behaviors, representing them as sequences of API and syscalls. We also developed MALGRAPHIQ, a behavior detection system that analyzes execution traces, generates behavior visualizations, and matches the WBC against them to identify patterns indicative of potentially malicious activity.

Research Question 1

To what extent the detection of behavior patterns through static analysis helps preventing the malicious activity?

As a first approach to static analysis, we conducted a systematic review of the literature on defenses and attacks related to the file-based TOCTOU vulnerability (see Chapter 3). This vulnerability arises when specific functions are executed while operating with the same filesystem object, making it a clear example of a vulnerable behavior patterns. Our study found that a wide variety of defenses have been proposed at different stages, including source code analysis, runtime detection, and post-mortem analysis. However, none offer a definitive solution, as the vulnerability is still frequently exploited in the wild. Detecting the vulnerability through static analysis via source-code pattern detection is helpful to raise warnings and reduce exploitation risks, but it does not prevent the vulnerability from occurring. The responsibility falls on the programmers, who must refactor the code so the vulnerability does not take place. This is often a tedious and error-prone task, as secure coding requires a high level of expertise. Definitive solutions would require modifying the kernel, filesystem, or API, but these approaches introduce backward compatibility issues, making them impractical for widespread adoption.

Based on these findings, our research shifted toward a broader approach, focusing on dynamic analysis of Windows programs.

Research Question 2

How do modern sandbox environments support dynamic analysis for capturing runtime behaviors of binaries in diverse scenarios?

In Chapter 4 we conducted a survey to identify available modern malware sandboxes. We selected those sandboxes able to analyze at least Windows programs and actively maintained, and we checked key features including configuration complexity, cost, supported operating systems, detection and classification capabilities, and privacy considerations. Sandboxes were categorized as open source or commercial, and we assessed their suitability in different scenarios both researchers and industry professionals may face, such as budget limitations, time constraints, customization requirements, technical expertise or privacy concerns. Our study confirmed that sandboxes are essential for dynamic analysis and a fundamental tool in the fight against malware, as they monitor the program execution in controlled and isolated environments without endangering other devices. Commercial sandboxes, typically provided as SaaS solutions, prioritize ease of use but offer limited customization. Open source sandboxes provide flexibility but often require advanced technical knowledge to configure and maintain. Despite their differences, both types remain critical tools for malware analysis, though evasion techniques and performance limitations continue to pose challenges.

Research Question 3

To what extent the available execution trace datasets (if any) are comprehensive and suitable for behavioral analysis?

Chapter 5 describes the development of MALVADA, a framework designed to help generating large datasets of execution traces, and the creation of WINMET, a dataset containing approximately 10,000 malware execution traces. Before creating WINMET, we analyzed other publicly available execution trace datasets and found that these are often incomplete or oversimplified. Some group malware families and types under the same category, while others removed essential contextual information by reducing execution traces to simplified representations, such as sequences of numeric identifiers, to optimize their use in artificial intelligence models. In contrast, WINMET retains the contextual information present in CAPE analysis reports, including hooked system and API calls, parameters, return values, mutexes, and accessed OS resources. We concluded that prior execution trace datasets are insufficient for compehensive behavioral analysis due to their lack of detail and contextual information, which limits their interpretability and usefulness. MALVADA and WINMET address this gap by providing a framework designed to generate datasets and a rich, detailed execution traces dataset, respectively.

Research Question 4

To what extent are we able to detect malicious behavior patterns through dynamic analysis?

The final part of this dissertation focused on the development of a dynamic analysis approach for detecting behavior patterns in execution traces. Chapter 6 presents MALGRAPHIQ, a system designed to identify behavioral patterns in unknown binaries by analyzing execution traces captured through the CAPE sandbox. Alongside MALGRAPHIQ, we also developed the WBC, a collection of fundamental behaviors commonly exhibited by Windows binaries, including both benign and malicious programs. The results of our experimental evaluation demonstrated that MALGRAPHIQ, in conjunction with the WBC, effectively detected behavior patterns across known malware families. The system successfully distinguished common and unique behaviors specific to certain malware types. The visualizations generated by MALGRAPHIQ can assist analysts and researchers in interpreting the behavior of any unknown binary, providing a clearer understanding of its actions. However, evasive and polymorphic malware remains a challenge, indicating that further improvements in behavioral analysis techniques are necessary.

Together, these findings emphasize the complementary roles of static and dynamic analysis, the importance of robust sandbox environments, and the critical need for high-quality datasets to advance behavioral malware detection.

7.2 Future Work and Open Problems

This research contributes to the detection of potentially malicious behavior patterns by studying source-code vulnerabilities, tools and data for malware behavior analysis, and behavior-based detection using execution traces. However, several limitations remain, and multiple research directions emerge from our findings, offering opportunities to improve the proposed approaches and contribute to more effective malware detection and prevention systems.

7.2.1 Future Work

Below, we outline potential improvements that could enhance our research and its results and generated artifacts.

- The survey on modern malware sandboxes serves as a reference for selecting analysis environments (see Chapter 4). However, future work should include empirical testing with malware samples crafted to assess anti-evasion capabilities, verifying whether advertised features and vendor claims withstand real-world threats. Expanding the study to include dynamic analysis tools beyond sandboxes, as well as commercial tools available only via trial versions, would enhance its completeness.
- MALVADA could benefit from performance improvements, as processing large sets of reports remains time-consuming. Enhancing the labeling process is also necessary to address cases where existing tools fail to assign a malware family, resulting in unassigned labels. Strengthening privacy-preserving mechanisms is critical to enable safe sharing of execution traces.
- The WINMET dataset requires further development to enhance its content and usability. The next version should classify samples not only by malware families but also by types. Furthermore, the dataset should be expanded to include samples executed on multiple versions of Windows, and the sample distribution should be balanced to prevent overrepresentation of specific families. The current dataset distribution method (which require users to download and uncompress the entire dataset) could also be improved by splitting the dataset into smaller, family or type-based, volumes to facilitate easier access.
- The Windows Behavior Catalog should be extended to cover additional behaviors defined in the MBC, such as anti-behavioral or anti-static analysis techniques, and ideally encompass functional examples of the most prevalent system and API calls used by malware [68]. Each behavior should be implemented through multiple methods, reflecting alternatives used to achieve the same behavior, thus improving coverage and detection accuracy.
- MALGRAPHIQ could benefit from several enhancements. The backtracking-based pattern-matching algorithm should be optimized and extended to analyze behavior graphs spanning across multiple categories, enabling the detection of complex behaviors. The system currently overlooks contextual information such as function parameters, return values, and accessed system resources, limiting detection accuracy. Future versions should incorporate this data into

the behavior-matching process. Furthermore, normalization and visualization methods require refinement to address issues identified during experimentation.

- Behavior graphs could benefit from integration with graph databases such as Neo4j, allowing their manipulation and enhancing query capabilities and facilitating large-scale graph comparison. Moreover, experimentation indicated that higher dimensionality of behavior vectors improves detection accuracy. While this observation holds in the controlled experimentation carried out in this research, behavior vectors should be further examined by exploring dimensionality reduction techniques and optimization strategies to enhance their scalability without compromising the overall system accuracy.
- This dissertation prioritized manual validation to ensure transparency and result interpretability. Future work should explore integrating machine learning techniques into behavior detection, with a focus on explainable artificial intelligence. Combining automated detection with interpretable models could improve performance metrics while maintaining the clarity needed for malware analysts.

7.2.2 Open and Emerging Research Directions

In this section we present the future research directions identified from the results of this dissertation. Some represent new or underexplored areas, while others are well-established challenges that remain unsolved.

- The systematic review presented in Chapter 3 highlighted that existing defenses are inadequate, as TOCTOU vulnerabilities persist and are actively exploited in the wild. A universal solution is unlikely, as it would require modifying core OS components, posing backward compatibility issues. Future work should focus on practical detection mechanisms at the source code level or at runtime, particularly kernel-level monitoring, without introducing significant performance overhead.
- The survey on modern malware sandboxes (see Chapter 4) highlighted transparency issues in SaaS environments and usability barriers in open source sandboxes. Future research should prioritize developing user-friendly, easy to configure, and open source solutions resilient to evasion and anti-analysis techniques.
- During the development of MALVADA and the creation of WINMET we faced challenges arising from the absence of a standardized malware naming scheme, like multiple names referring to the same malware family, vendors reusing names for different families, or the same label being assigned to entirely different families depending on the vendor. Although various naming schemes have been proposed, none has been universally adopted. Future research should support the development of universal malware naming and classification standards.
- Based on our experimentation, MALGRAPHIQ and the WBC proved to be ef-

ficient at detecting behaviors. Future research should explore how modern and sophisticated malware, particularly those employing evasion and anti-analysis techniques, can be represented through the visualizations generated by MALGRAPHIQ and how the WBC can be updated accordingly.

- Future research should focus on reducing noise in execution traces by identifying the precise moment when the sample execution begins, therefore distinguishing malware behavior from background processes generated by the operating system or the execution environment.

Chapter 8

Conclusiones y Trabajo Futuro

Esta tesis tiene como objetivo responder las preguntas de investigación planteadas en el Capítulo 1. Este capítulo resume los resultados obtenidos, contestando a cada pregunta de investigación, y discute problemas abiertos y posibles direcciones para trabajos futuros.

8.1 Conclusiones

Esta tesis se ha centrado en la identificación de patrones de comportamiento indicativos de actividades potencialmente maliciosas que podrían comprometer la seguridad de los sistemas. El objetivo inicial de la investigación era detectar fallos de seguridad durante la fase de desarrollo. Para ello, hemos estudiado una vulnerabilidad específica de código fuente para comprender sus causas, cómo se manifieesta y las técnicas de defensa y explotación propuestas previamente en la literatura. A partir de los resultados obtenidos, la investigación se orientó hacia un enfoque más amplio, centrado en el análisis de comportamientos de los programas durante su ejecución. Para hacer este tipo de análisis es necesario disponer de un entorno de ejecución adecuado, uno o más métodos para analizar las trazas de ejecución en busca de patrones de comportamiento, y acceso a conjuntos de datos de trazas de ejecución o, en su defecto, crear uno nuevo. En este sentido, realizamos una revisión de los entornos de análisis dinámico disponibles, proporcionando una guía que ayude a los analistas a elegir el entorno más adecuado para sus necesidades. Además, evaluamos la calidad de los conjuntos de datos de trazas de ejecución existentes para su aplicación en el análisis del comportamiento. Para abordar las limitaciones de los conjuntos de datos disponibles públicamente, desarrollamos MALVADA, un sistema diseñado para ayudar a generar grandes conjuntos de datos de trazas de ejecución. Utilizando MALVADA, creamos el conjunto de datos WINMET, que comprende casi 10.000 trazas de ejecución de malware de Windows. Además, creamos el WBC, un catálogo estructurado que contiene varias categorías de comportamientos de Windows, representándolos como secuencias de llamadas al sistema y a la API de Windows. También desarrollamos MALGRAPHIQ, un sistema de detección de comportamientos que analiza trazas de ejecución, genera visualizaciones de

comportamientos y los compara con el WBC para identificar patrones indicativos de actividades potencialmente maliciosas.

Pregunta de Investigación 1

¿En qué medida la detección de patrones de comportamiento mediante análisis estático ayuda a prevenir actividades maliciosas?

Como primer enfoque hacia el análisis estático, llevamos a cabo una revisión sistemática de la literatura sobre defensas y ataques relacionados con la vulnerabilidad de condición de carrera en operaciones con ficheros *Time-of-Check to Time-of-Use* (TOCTOU) (véase Chapter 3). Esta vulnerabilidad surge cuando se ejecutan funciones que operan sobre el mismo objeto del sistema de ficheros, lo que la convierte en un claro ejemplo de patrones de comportamiento vulnerables. Nuestro estudio encontró que se han propuesto defensas en diferentes etapas, incluyendo análisis de código fuente, detección en tiempo de ejecución y análisis post-mortem. Sin embargo, ninguna ofrece una solución definitiva, ya que la vulnerabilidad aún es explotada con frecuencia en entornos reales. La detección de la vulnerabilidad mediante análisis estático de código fuente es útil para generar advertencias y reducir los riesgos de su explotación, pero no evita que la vulnerabilidad ocurra. La responsabilidad recae sobre los programadores, quienes deben refactorizar el código para que la vulnerabilidad no se produzca. Esta tarea suele ser tediosa y propensa a errores, ya que la programación segura requiere un alto nivel de experiencia. Las soluciones definitivas requerirían modificar el núcleo del sistema operativo, el sistema de ficheros o la API, pero estos enfoques acarrean problemas de retrocompatibilidad, haciéndolos poco prácticos para su adopción generalizada.

En base a estos hallazgos, la investigación cambió hacia un enfoque más amplio, centrado en el análisis dinámico de programas Windows.

Pregunta de Investigación 2

¿Cómo apoyan los entornos sandbox modernos el análisis dinámico para capturar comportamientos de ejecución de binarios en diversos escenarios?

En Chapter 4 realizamos una búsqueda para identificar los entornos modernos de *sandbox* para malware disponibles. Seleccionamos aquellos capaces de analizar al menos binarios de Windows y que no estuvieran abandonados, y verificamos características clave como la complejidad de configuración, su coste, sistemas operativos soportados, capacidades de detección y clasificación, y consideraciones de privacidad. Dividimos los entornos entre código abierto o comerciales, y evaluamos su idoneidad en diferentes escenarios que podrían enfrentar investigadores y profesionales de la industria, como limitaciones de presupuesto, restricciones de tiempo, requisitos de personalización, conocimientos técnicos o implicaciones de privacidad. El estudio confirmó que estos entornos son esenciales para el análisis dinámico y una herramienta crucial en la lucha contra el malware y binarios desconocidos, ya que monitorizan la ejecución de dichos binarios en entornos controlados y aislados sin poner en peligro otros dispositivos. Las *sandboxes* comerciales, normalmente ofrecidas como soluciones SaaS, priorizan la facilidad

de uso pero ofrecen personalización limitada. Las *sandboxes* de código abierto permiten flexibilidad pero a menudo requieren conocimientos técnicos avanzados para su configuración y mantenimiento. A pesar de sus diferencias, ambos tipos siguen siendo herramientas cruciales para el análisis de malware, aunque las técnicas de evasión y las limitaciones de rendimiento continúan siendo un desafío mayor.

Pregunta de Investigación 3

¿En qué medida los conjuntos de datos de trazas de ejecución disponibles (si los hay) son completos y adecuados para el análisis de comportamiento?

Chapter 5 describe el desarrollo de MALVADA, un sistema diseñado para ayudar a generar grandes conjuntos de datos de trazas de ejecución, y la creación de WINMET, un conjunto de datos que contiene aproximadamente 10.000 trazas de ejecución de malware. Antes de crear WINMET, analizamos otros conjuntos de datos de trazas de ejecución públicamente disponibles y encontramos que a menudo son incompletos o se han simplificado en exceso. Algunos agrupan familias y tipos de malware bajo la misma categoría, mientras que otros eliminaron información contextual esencial al reducir las trazas de ejecución a representaciones simplificadas, como secuencias de identificadores numéricos, para optimizar su uso en modelos de inteligencia artificial. En contraste, WINMET conserva la información contextual presente en los informes de análisis de CAPE, incluyendo llamadas al sistema y API interceptadas, parámetros, valores de retorno, *mutexes* y recursos del sistema operativo accedidos. Concluimos que los conjuntos de datos ya existentes son insuficientes para un análisis de comportamiento en profundidad debido a su falta de detalle e información contextual, lo que limita su interpretabilidad y utilidad. MALVADA y WINMET abordan esta carencia proporcionando, respectivamente, un sistema diseñado para generar conjuntos de datos y un conjunto de trazas de ejecución detallado y enriquecido.

Pregunta de Investigación 4

¿En qué medida somos capaces de detectar patrones de comportamiento malicioso mediante análisis dinámico?

La parte final de esta tesis se centró en el desarrollo de un enfoque de análisis dinámico para detectar patrones de comportamiento en trazas de ejecución. Chapter 6 presenta MALGRAPHIQ, un sistema diseñado para identificar patrones de comportamiento en binarios desconocidos mediante el análisis de trazas de ejecución capturadas con la *sandbox* CAPE. Junto con MALGRAPHIQ, también desarrollamos el «Windows Behavior Catalog» (WBC), una colección de comportamientos fundamentales comúnmente presentes en programas en Windows, tanto benignos como maliciosos. Los resultados de nuestra evaluación experimental demostraron que MALGRAPHIQ, junto con el WBC, detectan eficazmente patrones de comportamiento en diversas familias de malware conocidas. El sistema distinguió con éxito comportamientos comunes y específicos de ciertos tipos de malware. Las visualizaciones generadas por MALGRAPHIQ pueden ayudar a analistas e investigadores a interpretar el comportamiento de cualquier binario

desconocido, proporcionando una comprensión más clara de sus acciones. Sin embargo, el malware evasivo y polimórfico sigue siendo un desafío, lo que indica que son necesarias más mejoras en las técnicas de análisis de comportamiento.

8.2 Trabajo Futuro y Problemas Abiertos

Esta investigación tuvo como objetivo contribuir a la detección de patrones de comportamiento potencialmente maliciosos mediante el estudio de vulnerabilidades en código fuente, herramientas y datos para el análisis de comportamiento de malware, y la detección basada en comportamiento utilizando trazas de ejecución. Sin embargo, existen varias limitaciones y surgen múltiples líneas de investigación a partir de los resultados, ofreciendo oportunidades para mejorar los enfoques propuestos y contribuir a sistemas más efectivos de detección y prevención de malware.

8.2.1 Trabajo Futuro

A continuación se destacan las mejoras que podrían enriquecer la investigación y mejorar los resultados y artefactos generados.

- El estudio sobre *sandboxes* modernos de malware sirve como referencia para la selección de entornos de análisis (véase Chapter 4). Sin embargo, el trabajo futuro debería incluir pruebas empíricas con muestras de malware diseñadas para evaluar capacidades anti-evasivas, verificando si las características y afirmaciones publicitadas resisten amenazas reales. Ampliar el estudio para incluir herramientas de análisis dinámico más allá de los *sandboxes*, así como herramientas comerciales disponibles solo en versiones de prueba, mejoraría su alcance.
- MALVADA podría beneficiarse de mejoras de rendimiento, ya que el procesamiento de grandes conjuntos de reportes sigue siendo lento. También es necesario mejorar el proceso de etiquetado para abordar los casos en que las herramientas existentes no asignan una familia de malware. También reforzar los mecanismos de preservación de la privacidad es fundamental para poder compartir de forma segura las trazas de ejecución.
- El conjunto de datos WINMET requiere un mayor desarrollo para mejorar su contenido y utilidad. La próxima versión debería clasificar las muestras no sólo por familias de malware, sino también por tipos. Además, el conjunto de datos debería ampliarse para incluir muestras ejecutadas en múltiples versiones de Windows, y la distribución de las muestras debería equilibrarse para evitar la sobrerrepresentación de familias específicas. El método actual de distribución del conjunto de datos (que requiere que los usuarios descarguen y descompriman todo el conjunto de datos) también podría mejorarse dividiendo el conjunto de datos en volúmenes más pequeños, basados en familias o tipos, para facilitar el acceso.
- El WBC debería ampliarse para cubrir comportamientos adicionales definidos en el MBC, como técnicas de anti-análisis, e idealmente abarcar ejemplos fun-

cionales de las llamadas al sistema y a la API más utilizadas por malware [68]. Cada comportamiento debería implementarse a través de múltiples métodos, reflejando las distintas alternativas utilizadas para lograr el mismo comportamiento, mejorando así la cobertura y la precisión de la detección.

- MALGRAPHIQ podría beneficiarse de varias mejoras. El algoritmo de reconocimiento de patrones basado en *backtracking* debería optimizarse para analizar gráficos de comportamiento que abarquen varias categorías, lo que permitiría la detección de comportamientos complejos. En la actualidad, el sistema pasa por alto cierta información contextual como los parámetros de las funciones, los valores de retorno y los recursos del sistema a los que se accede, lo que limita la precisión de la detección. Las versiones futuras deberían incorporar estos datos al proceso de reconocimiento de comportamientos. Además, los métodos de normalización y visualización deben perfeccionarse para resolver los problemas detectados durante la experimentación.
- Los grafos de comportamiento se podrían integrar con bases de datos orientadas a grafos como Neo4J, lo que permitiría la manipulación de los grafos y mejoraría la capacidad de consulta y la comparación de grafos a gran escala. Por otro lado, los resultados de la experimentación sugieren que una mayor dimensionalidad en los vectores de comportamiento aumenta la precisión de la detección. Si bien es cierto que esta afirmación es cierta para los experimentos llevados a cabo en este trabajo, los vectores de comportamiento deberían ser examinados más en profundidad. Por ejemplo, explorando técnicas de reducción de dimensiones y estrategias de optimización para mejorar la escalabilidad de los mismos sin afectar a la precisión del sistema.
- En esta tesis se ha dado prioridad a la validación manual para garantizar la transparencia y la interpretabilidad de los resultados. El trabajo futuro debería explorar la integración de técnicas de aprendizaje automático en la detección de comportamientos, manteniendo presentes los principios de inteligencia artificial explicable. Combinar la detección automatizada con modelos interpretables podría mejorar las métricas de rendimiento, manteniendo al mismo tiempo la claridad necesaria para los analistas de malware.

8.2.2 Líneas de Investigación Abiertas y Emergentes

En esta sección presentamos las líneas de investigación futuras que surgen a raíz del trabajo realizado y los resultados obtenidos. Algunas representan nuevas líneas de investigación, mientras que otras son problemas ya establecidos que aún no se han resuelto.

- La revisión sistemática llevada a cabo en el Chapter 3 resaltó que las defensas existentes siguen siendo inadecuadas, ya que las vulnerabilidades TOCTOU persisten y son explotadas activamente en entornos reales. Es poco probable que exista una solución universal, ya que requeriría modificar componentes centrales del sistema operativo, lo que plantearía problemas de retrocompatibilidad. A partir de este punto, los esfuerzos de investigación deberían centrarse en mecanismos prácticos de detección a nivel de código fuente o en tiempo de

ejecución, particularmente en la monitorización a nivel de núcleo del sistema operativo, sin introducir una sobrecarga significativa de rendimiento.

- El estudio en entornos modernos de análisis de malware (véase Chapter 4) concluyó que existen actualmente problemas de transparencia en entornos SaaS y barreras de usabilidad en *sandboxes* de código abierto. El foco de la investigación futura debería priorizar el desarrollo de soluciones de código abierto fáciles de instalar, configurar y usar, e idealmente resistentes a técnicas de evasión y anti-análisis.
- Durante el desarrollo de MALVADA y la creación de WINMET nos encontramos problemas que surgen dada la ausencia de un esquema de nomenclatura estándar de malware, como múltiples nombres haciendo referencia a la misma familia, algunas empresas reutilizando el mismo nombre para distintas familias, o la misma etiqueta usada para referirse a familias totalmente distintas dependiendo de la empresa. Aunque previamente se han propuesto distintos esquemas de nomenclatura, ninguno se ha adoptado universalmente. Los esfuerzos de investigación deberían centrarse en desarrollar una nomenclatura universal de malware y estándarse de clasificación.
- En base a nuestros experimentos, MALGRAPHIQ y el WBC han demostrado ser efectivos a la hora de detectar comportamientos. A partir de este momento, la investigación debería explorar cómo malware moderno y sofisticado, particularmente aquellas familias que usan técnicas de evasión y anti-análisis, pueden representarse con las gráficas generadas por MALGRAPHIQ y cómo el WBC se puede actualizar acorde a estos comportamientos.
- Los futuros trabajos de investigación deberían centrarse en tratar de reducir al máximo el ruido en las trazas de ejecución mediante la identificación del momento preciso en que la ejecución de la muestra comienza, distinguiendo, así, entre el comportamiento del malware y el comportamiento de los procesos ejecutados por el sistema operativo o el entorno de ejecución.

Research Outcomes

This chapter outlines the scientific results and other contributions developed during the course of this dissertation. In addition, contributions made to other open source projects are highlighted.

Scientific Publications

As a scientific result of this dissertation, two papers have been published in peer-reviewed journals and another two papers are currently under review:

- R. Raducu, R. J. Rodríguez, and P. Álvarez, "Defense and Attack Techniques Against File-Based TOCTOU Vulnerabilities: A Systematic Review," in IEEE Access, vol. 10, pp. 21742-21758, 2022, DOI: 10.1109/ACCESS.2022.3153064. Impact factor (JCR): 3.9 (3.6 without self citations), rank 73/158 (Q2; Computer Science, Information Systems).
- R. Raducu, A. Villagrasa-Labrador, R. J. Rodrígue,z and P. Álvarez, "MALVADA: A framework for generating datasets of malware execution traces," in SoftwareX, vol. 30, 2025, DOI: 10.1016/j.softx.2025.102082. Impact factor (JCR): 3.4 (3.3 without self citations), rank 38/108 (Q2; Computer Science, Software Engineering).
- R. Raducu, R. J. Rodríguez, and P. Álvarez, "A Graph-Based Dynamic Analysis System for Behavior Detection in Windows Applications". Currently *under review*, submitted to The Computer Journal. Impact factor (JCR): 1.5 (1.4 without self citations), rank 71/144 (Q2; Computer Science, Theory & Methods).
- R. Raducu, A. Zarras, R. J. Rodríguez, and P. Álvarez, "The Sandbox Reloaded: A Guide to Modern Malware Sandbox". Currently *under review*, submitted to the 20th International Conference on Availability, Reliability and Security (ARES 2025). B rank (28.19% of 784 ranked venues) according to CORE2023.

Open Source Projects

In the spirit of open science, and to support reproducibility and extensibility, the majority of our results are publicly accessible and freely available:

- Windows Behavior Catalog (WBC): Collection of fundamental behaviors for Windows OS, represented as a sequence of Windows API and/or syscalls.

Available at https://github.com/reverseame/windows-behavior-catalog under the GNU GPL-3.0 license (accessed on February 19, 2025).

- MALGRAPHIQ: Tool that transforms malware sandbox reports (at the time being from CAPEv2, but can be extended) and execution traces into transition matrices, behavior graphs, and category graphs. It compares WBC's patterns to them and plots occurrences, generating visual representations of the identified behaviors. Available at https://github.com/reverseame/MalGraphIQ under the GNU GPL-3.0 license (accessed on February 19, 2025).
- MALVADA: Framework that parses one or more CAPEv2 reports and processes them to generate curated execution traces, helping to generate execution trace datasets. Available at https://github.com/reverseame/MALVADA under the GNU GPL-3.0 license (accessed on February 19, 2025).
- Windows Malware Execution Traces (WINMET) dataset: Collection of execution traces of different malware families, analyzed with CAPEv2 sandbox and processed with MALVADA. DOI: 10.5281/zenodo.12647555 (accessed on February 19, 2025).
- CAPE Hook Generator: Tool that generates new hook skeletons (*hookdefs*) for `capemon`, the CAPEv2 monitor, to help users extend `capemon`'s functionality. Available at https://github.com/reverseame/cape-hook-generator under the GNU GPL-3.0 license (accessed on February 19, 2025).
- Windows API and syscalls categories: Classification of Windows API (WinAPI) functions and system calls (syscalls), including the Native API (NTAPI), according to their category. Available at https://github.com/reverseame/winapi-categories under the GNU GPL-3.0 license (accessed on February 19, 2025).

Contribution to Other Open Source Projects

Throughout the development of this thesis, several contributions were made to open source projects via pull requests, opening issues, or by participating in discussions that resulted in commits to the main repository:

- CAPEv2 Sandbox. Contributed to update the documentation: https://github.com/kevoreilly/CAPEv2/commits?author=RazviOverflow (accessed on November 20, 2024).
- CAPE's Monitor (capemon). Contributed to update the documentation and the set of hooked functions: https://github.com/kevoreilly/capemon/commits?author=RazviOverflow (accessed on November 20, 2024); and reported an issue: https://github.com/kevoreilly/capemon/issues/50 (accessed on November 20, 2024).
- Malware Behavior Catalog (MBC). Contributed by creating the "*How to cite*" file and by updating CAPE signature mappings: https://github.com/MBCProject/mbc-markdown/commits?author=RazviOverflow (accessed on

November 20, 2024); and started a discussion about a registry-related behavior name: https://github.com/MBCProject/mbc-markdown/discussions/100 (accessed on November 20, 2024).

- Cryptopp-pem. Reported an issue related to the `MSVC` compiler: https://github.com/noloader/cryptopp-pem/issues/10 (accessed on November 20, 2024).
- botan. Reported an issue related to the `ICC` compiler: https://github.com/randombit/botan/issues/2748 (accessed on November 20, 2024).
- Cutter. Reported two issues: https://github.com/rizinorg/cutter/issues/2895 and https://github.com/rizinorg/cutter/issues/3388 (accessed on November 20, 2024).

Funding Acknowledgments

This doctoral dissertation has been funded by the University of Zaragoza through the OTRI (Research Results Transfer Office) projects *LEBIS: INTELIGENCIA DE NEGOCIO A GRAN ESCALA SOBRE ECOSISTEMAS DE PROVEEDORES DE SERVICIO (TIN2017-84796-C2-2-R)* and *INTELIGENCIA DE NEGOCIO Y CIBERSEGURIDAD EN LA NUBE (LTI3A0108-02)*, and by the Government of Aragón through the *Diputación General de Aragón* (DGA) Predoctoral Grant 2021–2025.

Declaration of Generative AI and AI-Assisted Technologies in the Writing Process

During the preparation of this dissertation the author used ChatGPT-4 in order to improve readability and language. After using this tool/service, the author reviewed and edited the content as needed and takes full responsibility for the content of the publication.

Bibliography

[1] Javier Guerrero, *Malware and operating systems*, [Online; https://www.pandasecurity.com/en/mediacenter/malware-and-operating-systems/, accessed on September 22, 2024], 2010.

[2] AV-TEST, *Security Report 2019/20*, [Online; https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2019-2020.pdf, accessed on September 22, 2024], 2019.

[3] AV-TEST, *Malware Statistics & Trends Report*, [Online; https://www.av-test.org/en/statistics/malware/, accessed on September 22, 2024], 2024.

[4] Check Point Software Technologies Ltd., *The State of Cyber Security 2025*, [Online; https://www.checkpoint.com/security-report/?flz-category=items&flz-item=report--cyber-security-report-2025, accessed on February 6, 2025], 2025.

[5] NSA, *Cybersecurity Year in Review 2023*, [Online; https://media.defense.gov/2023/Dec/19/2003362479/-1/-1/0/NSA_Cybersecurity_YiR23_Book_508.PDF, accessed on October 27, 2024], Dec. 2023.

[6] K. Zettl-Schabath, J. Bund, M. Müller, and C. Borrett, *EuRepoC Cyber Conflict Briefing – 2023 Cyber Activity Balance*, [Online; https://eurepoc.eu/publication/eurepoc-cyber-conflict-briefing-2023-cyber-activity-balance/, accessed on June 17, 2024], Jan. 2024.

[7] ENISA, *Threat Landscape 2024*, [Online; https://www.enisa.europa.eu/publications/enisa-threat-landscape-2024, accessed on January 22, 2025], Sep. 2024.

[8] W. S. McPhee, "Operating System Integrity in OS/VS2", *IBM Systems Journal*, vol. 13, no. 3, pp. 230–252, 1974. DOI: 10.1147/sj.133.0230.

[9] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb, "Security Analysis and Enhancements of Computer Operating Systems", Institute of Computer Sciences and Technology, National Bureau of Standards, Gaithersburg, MD, Tech. Rep. NBSIR 76-1041, Apr. 1976.

[10] MITRE, *MITRE CVE - TOCTOU Search Results*, [Online; https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=file-based+TOCTOU, accessed on January 28, 2025], Jan. 2025.

[11] National Vulnerability Database, *NVD - TOCTOU Search Results*, [Online; https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&search_type=all&cwe_id=CWE-59&isCpeNameSearch=false, accessed on January 28, 2025], Jan. 2025.

[12] Statcounter Global Stats, *Desktop Operating System Market Share Worldwide*, [Online; https://gs.statcounter.com/os-market-share/desktop/worldwide, accessed on January 21, 2025], 2025.

[13] K. O'Reilly and A. Brukhovetskyy, *CAPE: Malware Configuration And Payload Extraction*, [Online; https://github.com/kevoreilly/CAPEv2, accessed on February 2, 2025], version 2, 2024.

[14] T. Warszawski and P. Bailis, "ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications", in *Proceedings of the 2017 ACM International Conference on Management of Data*, Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 5–20, ISBN: 9781450341974. DOI: 10.1145/3035918.3064037.

[15] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan, "Concurrency Attacks", in *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism (HotPar'12)*, Berkeley, CA: USENIX Association, 2012, p. 15.

[16] BeyondTrust, *Microsoft Vulnerabilities Report 2021*, [Online; https://www.beyondtrust.com/assets/documents/BeyondTrust-Microsoft-Vulnerabilities-Report-2021.pdf, accessed on May 30, 2021], Mar. 2021.

[17] BeyondTrust, *Microsoft Vulnerabilities Report 2024*, [Online; https://assets.beyondtrust.com/assets/documents/BT_whitepaper_Microsoft-Vulnerabilities-Report-2024.pdf, accessed on January 21, 2025], 2024.

[18] M. Bishop and M. Dilger, "Checking for Race Conditions in File Accesses", *Computing Systems*, vol. 9, no. 2, pp. 131–152, 1996, ISSN: 0895-6340.

[19] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How Double-Fetch Situations Turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel", in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, Aug. 2017, pp. 1–16, ISBN: 978-1-931971-40-9.

[20] P. Wang, K. Lu, G. Li, and X. Zhou, "DFTracker: Detecting Double-Fetch Bugs by Multi-Taint Parallel Tracking", *Front. Comput. Sci.*, vol. 13, no. 2, pp. 247–263, Apr. 2019, ISSN: 2095-2228. DOI: 10.1007/s11704-016-6383-8.

[21] M. Ambrosin, M. Conti, R. Lazzeretti, M. M. Rabbani, and S. Ranise, "Collective Remote Attestation at the Internet of Things Scale: State-of-the-Art and Future Challenges", *IEEE Communications Surveys Tutorials*, vol. 22, no. 4, pp. 2447–2461, 2020. DOI: 10.1109/COMST.2020.3008879.

[22] O. Arias, D. Sullivan, H. Shan, and Y. Jin, "LAHEL: Lightweight Attestation Hardening Embedded Devices using Macrocells", in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 305–315. DOI: 10.1109/HOST45689.2020.9300257.

[23] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith, "TOCTOU, Traps, and Trusted Computing", in *International Conference on Trusted Computing*, vol. 4968, Villach, Austria: Springer-Verlag, Mar. 2008, pp. 14–32, ISBN: 978-3-540-68978-2. DOI: 10.1007/978-3-540-68979-9_2.

[24] X. Chang, B. Xing, J. Liu, Z. Yuan, and L. Sun, "Robust and Efficient Response to TCG TOCTOU Attacks in TPVM", in *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, IEEE, 2012, pp. 323–328.

[25] J. Wei and C. Pu, "Modeling and Preventing TOCTTOU Vulnerabilities in Unix-style File Systems", *Computers & Security*, vol. 29, no. 8, pp. 815–830, 2010, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2010.09.004.

[26] GNU, *The meaning of the File Attributes (The GNU C Library)*, [Online; https://www.gnu.org/software/libc/manual/html_node/Attribute-Meanings.html, accessed on March 3, 2025].

[27] J. Wei and C. Pu, "TOCTTOU vulnerabilities in unix-style file systems: An anatomical study", in *4th USENIX Conference on File and Storage Technologies (FAST 05)*, San Francisco, CA, Dec. 2005, p. 12.

[28] D. Plohmann, M. Clauss, S. Enders, and E. Padilla, "Malpedia: A Collaborative Effort to Inventorize the Malware Landscape", *The Journal on Cybercrime & Digital Investigations*, vol. 3, no. 1, 2018.

[29] M. Sikorski and A. Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*. No Starch Press, 2012.

[30] M. Yong Wong, M. Landen, M. Antonakakis, D. M. Blough, E. M. Redmiles, and M. Ahamad, "An inside look into the practice of malware analysis", in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 3053–3069, ISBN: 9781450384544.

[31] P. Szor, *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005, ISBN: 0321304543.

[32] A. Kleymenov and A. Thabet, *Mastering Malware Analysis: A malware analyst's practical guide to combating malicious software, APT, cybercrime, and IoT attacks*. Packt Publishing Ltd, 2022, ISBN: 9781803240244.

[33] K. Cucci, *Evasive Malware: A Field Guide to Detecting, Analyzing, and Defeating Advanced Threats*. No Starch Press, 2024, ISBN: 9781718503267.

[34] M. Y. Wong, M. Landen, F. Li, F. Monrose, and M. Ahamad, "Comparing malware evasion theory with practice: Results from interviews with expert analysts", in *Twentieth Symposium on Usable Privacy and Security (SOUPS 2024)*, Philadelphia, PA: USENIX Association, Aug. 2024, pp. 61–80, ISBN: 978-1-939133-42-7. [Online]. Available: https://www.usenix.org/conference/soups2024/presentation/yong-wong.

[35] K. Liu, H. B. K. Tan, and X. Chen, "Binary Code Analysis", *Computer*, vol. 46, no. 8, pp. 60–68, 2013, ISSN: 0018-9162.

[36] E. Eilam, *Reversing: Secrets of Reverse Engineering*. Wiley, 2011, ISBN: 9781118079768.

[37] B. Dang, A. Gazet, E. Bachaalany, and S. Josse, *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. Wiley, 2014, ISBN: 9781118787397.

[38] P. Szor and P. Ferrie, "Hunting for metamorphic", in *Virus bulletin conference*, Citeseer, 2001.

[39] K. Tsyganok, E. Tumoyan, L. Babenko, and M. Anikeev, "Classification of Polymorphic and Metamorphic Malware Samples Based on Their Behavior", in *Proceedings of the Fifth International Conference on Security*

of Information and Networks, ser. SIN '12, Jaipur, India: Association for Computing Machinery, 2012, pp. 111–116, ISBN: 9781450316682.

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

[41] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns", in *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C.: USENIX Association, Aug. 2003. [Online]. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/static-analysis-executables-detect-malicious-patterns.

[42] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection", in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007, pp. 421–430.

[43] M. V. Yason, *The Art of Unpacking*, [Online; https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf, accessed on January 25, 2025], 2007.

[44] A. Mantovani, S. Aonzo, Y. Fratantonio, and D. Balzarotti, "RE-Mind: A first look inside the mind of a reverse engineer", in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 2727–2745, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/mantovani.

[45] 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*, ser. ROOTS, Vienna, Austria: Association for Computing Machinery, 2017, pp. 1–21, ISBN: 9781450353212.

[46] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, "Malware dynamic analysis evasion techniques: A survey", *ACM Comput. Surv.*, vol. 52, no. 6, Nov. 2019, ISSN: 0360-0300.

[47] M. Kim, H. Cho, and J. H. Yi, "Large-scale analysis on anti-analysis techniques in real-world malware", *IEEE Access*, vol. 10, pp. 75 802–75 815, 2022. DOI: 10.1109/ACCESS.2022.3190978.

[48] T. Shields, "Anti-debugging–a developers view", *Veracode Inc., USA*, 2010.

[49] F. Peter, *The "Ultimate" Anti-Debugging Reference*, [Online; https://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf, accessed on January 25, 2025], May 2011.

[50] M. N. Gagnon, S. Taylor, and A. K. Ghosh, "Software protection through anti-debugging", *IEEE Security & Privacy*, vol. 5, no. 3, pp. 82–84, 2007.

[51] P. Chen, C. Huygens, L. Desmet, and W. Joosen, "Advanced or Not? A Comparative Study of the Use of Anti-debugging and Anti-VM Techniques in Generic and Targeted Malware", in *ICT Systems Security and Privacy Protection*, J.-H. Hoepman and S. Katzenbeisser, Eds., Springer, 2016, pp. 323–336, ISBN: 978-3-319-33630-5.

[52] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in

modern malware", in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008, pp. 177–186.

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

[54] T. Apostolopoulos, V. Katos, K.-K. R. Choo, and C. Patsakis, "Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks", *Future Generation Computer Systems*, vol. 116, pp. 393–405, 2021, ISSN: 0167-739X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X20330284.

[55] A. S. Filho, R. J. Rodríguez, and E. L. Feitosa, "Evasion and Countermeasures Techniques to Detect Dynamic Binary Instrumentation Frameworks", *Digital Threats: Research and Practice*, vol. 3, no. 2, p. 28, Feb. 2022, ISSN: 2692-1626.

[56] M. D. Ernst, "Static and dynamic analysis: Synergy and duality", in *WODA 2003: Workshop on Dynamic Analysis*, Portland, OR, USA, May 2003, pp. 24–27.

[57] D. Solomon, "The Windows NT kernel architecture", *Computer*, vol. 31, no. 10, pp. 40–47, 1998.

[58] Microsoft, *User Mode and Kernel Mode*, [Online; https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode, accessed on January 27, 2025], 2024.

[59] Microsoft, *Windows API index*, [Online; https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list, accessed on January 27, 2025], 2023.

[60] Microsoft, *New Low-Level Binaries*, [Online; https://learn.microsoft.com/en-us/windows/win32/win7appqual/new-low-level-binaries, accessed on January 27, 2025], 2021.

[61] P. Yosifovich, A. Ionescu, M. E. Russinovich, and D. A. Solomon, *Windows Internals, Part 1: System architecture, processes,threads, memory management,and more*, 7th. Redmond, WA: Microsoft Press, 2017, ISBN: 978-0-7356-8418-8.

[62] T. Nowak, *NTAPI Undocumented Functions. The Undocumented Functions for Microsoft Windows NT/2000/XP/Win7*, [Online; http://undocumented.ntinternals.net/, accessed on January 27, 2025], 2024.

[63] M. Maltsev, *NtDoc: The native NT API online documentation*, [Online; https://ntdoc.m417z.com/, accessed on January 27, 2024], 2024.

[64] Jurczyk, Mateusz, *Windows System Call Tables (NT/2000/XP/2003/Vista/7/8/10/11)*, [Online; https://github.com/j00ru/windows-syscalls, accessed on January 27, 2025], 2024.

[65] MITRE, *Native API, Technique T1106 - Enterprise | MITRE ATT&CK)*, [Online; https://attack.mitre.org/techniques/T1106/, accessed on January 27, 2025], 2024.

[66] MITRE, *Debugger Evasion, Technique T1622 - Enterprise | MITRE ATT&CK*, [Online; https://attack.mitre.org/techniques/T1622/, accessed on January 27, 2025], 2022.

[67] Roccia, Thomas and Lesueur, Jean-Pierre, *Evasion using direct syscalls - Unprotect Project*, [Online; https://unprotect.it/technique/evasion-using-direct-syscalls/, accessed on January 27, 2025], 2023.

[68] mr.d0x, *MalAPI.io*, [Online; https://malapi.io, accessed on January 27, 2025], 2024.

[69] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation", *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, pp. 203–216, Dec. 1993, ISSN: 0163-5980.

[70] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications", in *6th USENIX Security Symposium (USENIX Security 96)*, San Jose, CA: USENIX Association, Jul. 1996. [Online]. Available: https://www.usenix.org/conference/6th-usenix-security-symposium/secure-environment-untrusted-helper-applications.

[71] V. Prevelakis and D. Spinellis, "Sandboxing applications", in *2001 USENIX Annual Technical Conference (USENIX ATC 01)*, Boston, MA: USENIX Association, Jun. 2001. [Online]. Available: https://www.usenix.org/conference/2001-usenix-annual-technical-conference/sandboxing-applications.

[72] MITRE, *Virtualization/Sandbox Evasion, Technique T1497 - Enterprise | MITRE ATT&CK)*, [Online; https://attack.mitre.org/techniques/T1497/, accessed on January 27, 2025], 2024.

[73] MITRE, *Malware Behavior Catalog*, [Online; https://github.com/MBCProject/mbc-markdown, accessed on January 27, 2025], Dec. 2024.

[74] P. Wang, K. Lu, G. Li, and X. Zhou, "A Survey of the Double-fetch Vulnerabilities", *Concurrency and Computation: Practice and Experience*, vol. 30, no. 6, e4345, 2018. DOI: 10.1002/cpe.4345. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4345.

[75] R. V. Steiner and E. Lupu, "Attestation in Wireless Sensor Networks: A Survey", vol. 49, no. 3, Sep. 2016, ISSN: 0360-0300. DOI: 10.1145/2988546.

[76] Z. Fang, W. Han, and Y. Li, "Permission based Android Security: Issues and Countermeasures", *Computers & Security*, vol. 43, pp. 205–218, 2014, ISSN: 0167-4048. DOI: 10.1016/j.cose.2014.02.007.

[77] C. E. McDowell and D. P. Helmbold, "Debugging Concurrent Programs", *ACM Comput. Surv.*, vol. 21, no. 4, pp. 593–622, Dec. 1989, ISSN: 0360-0300. DOI: 10.1145/76894.76897.

[78] E. Lee, "The Problem with Threads", *Computer*, vol. 39, no. 5, pp. 33–42, 2006. DOI: 10.1109/MC.2006.180.

[79] Flysystem, *Time-of-check Time-of-use (TOCTOU) Race Condition in league/flysystem*, [Online; https://github.com/thephpleague/flysystem/security/advisories/GHSA-9f46-5r25-5wfm, accessed on January 28, 2025], Jun. 2021.

[80] Flysystem, *Linux Kernel: TOCTOU in Exec System*, [Online; https://github.com/google/security-research/security/advisories/GHSA-c45w-xwww-rfgg, accessed on January 28, 2025], Dec. 2024.

[81] VMWare, *VMSA-2020-0023.3*, [Online; https://www.vmware.com/security/advisories/VMSA-2020-0023.html, accessed on January 28, 2025], Oct. 2020.

[82] Red Hat, *CVE-2021-30465 - Red Hat Customer Portal*, [Online; https://access.redhat.com/security/cve/cve-2021-30465, accessed on January 28, 2025], May 2021.

[83] Adobe, *Adobe Security Bulletin*, [Online; https://helpx.adobe.com/security/products/creative-cloud/apsb20-11.html, accessed on January 28, 2025], Mar. 2020.

[84] National Vulnerability Database, *NVD - CVE-2024-50379 | TOCTOU Race Condition*, [Online; https://nvd.nist.gov/vuln/detail/CVE-2024-50379, accessed on January 18, 2025].

[85] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic Literature Reviews in Software Engineering–a Systematic Literature Review", *Information and Software Technology*, vol. 51, no. 1, pp. 7–15, 2009, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2008.09.009.

[86] A. P. Siddaway, A. M. Wood, and L. V. Hedges, "How to Do a Systematic Review: a Best Practice Guide for Conducting and Reporting Narrative Reviews, Meta-analyses, and Meta-syntheses", *Annual Review of Psychology*, vol. 70, no. 1, pp. 747–770, 2019.

[87] G. Gu, *Computer Security Conference Ranking and Statistic*, [Online; https://people.engr.tamu.edu/guofei/sec_conf_stat.htm, accessed on January 28, 2025], 2022.

[88] A. Zamboni, A. Thommazo, E. Hernandes, and S. Fabbri, "StArt Uma Ferramenta Computacional de Apoio à Revisão Sistemática", in *Congresso Brasileiro de Software (CBSoft'10)*, Salvador, Brazil, 2010, pp. 91–96.

[89] E. Hernandes, A. Zamboni, S. Fabbri, and A. D. Thommazo, "Using GQM and TAM to Evaluate StArt-A Tool that Supports Systematic Review", *CLEI Electronic Journal*, vol. 15, no. 1, pp. 3–3, Apr. 2012.

[90] M. J. Page et al., "The prisma 2020 statement: An updated guideline for reporting systematic reviews", *BMJ*, vol. 372, 2021.

[91] M. Bishop, "Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux", University of California at Davis, Davis, CA, Tech. Rep. Report CSE-95-9, 1995.

[92] D. Dean and A. J. Hu, "Fixing Races for Fun and Profit: How to Use access(2)", in *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA: USENIX Association, Aug. 2004, pp. 195–206.

[93] C. Ko, G. Fink, and K. Levitt, "Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring", in *Tenth Annual Computer Security Applications Conference*, 1994, pp. 134–144. DOI: 10.1109/CSAC.1994.367313.

[94] B. Goyal, S. Sitaraman, and S. Venkatesan, "A Unified Approach to Detect Binding Based Race Condition Attacks", in *Int'l Workshop on Cryptology & Network Security (CANS)*, 2003, p. 16.

[95] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow Anomaly Detection", in *2006 IEEE Symposium on Security and Privacy (SP'06)*, 2006, 15 pp.–62. DOI: 10.1109/SP.2006.12.

[96] O. Laadan et al., "Finding Concurrency Errors in Sequential Code: OS-Level, in-Vivo Model Checking of Process Races", in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS XIII)*, Napa, California: USENIX Association, May 2011, p. 20.

[97] O. Laadan, N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, and J. Nieh, "Pervasive Detection of Process Races in Deployed Systems", in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal: Association for Computing Machinery, 2011, pp. 353–367, ISBN: 9781450309776. DOI: 10.1145/2043556.2043589.

[98] T. Yu, W. Srisa-an, and G. Rothermel, "SimRacer: An automated Framework to Support Testing for Process-level Races", in *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*, ser. ISSTA 2013, Lugano, Switzerland: Association for Computing Machinery, Jul. 2013, pp. 167–177, ISBN: 9781450321594. DOI: 10.1145/2483760.2483771. [Online]. Available: 10.1145/2483760.2483771.

[99] T. Yu, W. Srisa-an, and G. Rothermel, "An Automated Framework to Support Testing for Process-level Race Conditions", *Software Testing, Verification and Reliability*, vol. 27, no. 4-5, e1634, 2017.

[100] F. Capobianco et al., "Employing Attack Graphs for Intrusion Detection", in *Proceedings of the New Security Paradigms Workshop (NSPW '19)*, San Carlos, Costa Rica: Association for Computing Machinery, 2019, pp. 16–30. DOI: 10.1145/3368860.3368862.

[101] A. Aggarwal and P. Jalote, "Monitoring the Security Health of Software Systems", in *17th International Symposium on Software Reliability Engineering*, 2006, pp. 146–158. DOI: 10.1109/ISSRE.2006.32.

[102] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva, "Portably Solving File TOCTTOU Races with Hardness Amplification", in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, San Jose, California: USENIX Association, 2008.

[103] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva, "Portably Preventing File Race Attacks with User-mode Path Resolution", IBM Research, Yorktown Heights, NY, Tech. Rep. RC24572 (W0806-008), 2008.

[104] S. Chari, S. Halevi, and W. Z. Venema, "Where Do You Want to Go Today? Escalating Privileges by Pathname Manipulation", in *Proceedings of the Network and Distributed System Security Symposium (NDSS 2010)*, San Diego, CA, Mar. 2010, pp. 1–16.

[105] M. Payer and T. R. Gross, "Protecting Applications Against TOCTTOU Races by User-space Caching of File Metadata", in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, New York, NY, USA: Association for Computing Machinery, Jul. 2012, pp. 215–226, ISBN: 9781450311762. DOI: 10.1145/2365864.2151052.

[106] X. Cai, R. Lale, X. Zhang, and R. Johnson, "Fixing Races for Good: Portable and Reliable U File-system Race Detection", in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*

(ASIA CCS '15), New York, NY, USA: Association for Computing Machinery, 2015, pp. 357–368.

[107] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman, "RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities", in *10th USENIX Security Symposium (USENIX Security 01)*, Washington, D.C.: USENIX Association, Aug. 2001, pp. 165–176.

[108] C. Ko and T. Redmond, "Noninterference and Intrusion Detection", in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002, pp. 177–187. DOI: 10.1109/SECPRI.2002.1004370.

[109] E. Tsyrklevich and B. Yee, "Dynamic Detection and Prevention of Race Conditions in File Accesses", in *Proceedings of the 12th conference on USENIX Security Symposium (SSYM'03)*, Washington, D.C.: USENIX Association, Aug. 2003, p. 17.

[110] J. Park, G. Lee, S. Lee, and D.-k. Kim, "RPS: An Extension of Reference Monitor to Prevent Race-attacks", in *Advances in Multimedia Information Processing (PCM 2004)*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 556–563.

[111] K. S. Lhee and S. J. Chapin, "Detection of File-based Race Conditions", *International Journal of Information Security*, vol. 4, no. 1-2, pp. 105–119, Feb. 2005, ISSN: 16155262. DOI: 10.1007/s10207-004-0068-2.

[112] P. Uppuluri, U. Joshi, and A. Ray, "Preventing Race Condition Attacks on File-systems", in *Proceedings of the 2005 ACM symposium on Applied computing (SAC '05')*, Santa Fe, New Mexico: Association for Computing Machinery, 2005, pp. 346–353, ISBN: 1581139640.

[113] J. A. Kupsch and B. P. Miller, "How to Open a File and Not Get Hacked", in *2008 Third International Conference on Availability, Reliability and Security*, 2008, pp. 1196–1203. DOI: 10.1109/ARES.2008.53.

[114] J. Rouzaud-Cornabas, P. Clemente, and C. Toinard, "An Information Flow Approach for Preventing Race Conditions: Dynamic Protection of the Linux OS", in *2010 Fourth International Conference on Emerging Security Information, Systems and Technologies*, 2010, pp. 11–16. DOI: 10.1109/SECURWARE.2010.10.

[115] H. Vijayakumar, J. Schiffman, and T. Jaeger, "A Rose by Any Other Name or an Insane Root? Adventures in Name Resolution", in *2011 Seventh European Conference on Computer Network Defense*, 2011, pp. 1–8. DOI: 10.1109/EC2ND.2011.17.

[116] H. Vijayakumar, J. Schiffman, and T. Jaeger, "STING: Finding Name Resolution Vulnerabilities in Programs", *21st USENIX Security Symposium (USENIX Security 12)*, pp. 585–599, Aug. 2012.

[117] H. Vijayakumar, J. Schiffman, and T. Jaeger, "Process Firewalls: Protecting Processes During Resource Access", in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, New York, NY, USA: Association for Computing Machinery, 2013, pp. 57–70, ISBN: 9781450319942.

[118] H. Vijayakumar and T. Jaeger, "The Right Files at the Right Time", in *Proceedings of the 5th IEEE Symposium on Configuration Analytics and Automation (SafeConfig 2012)*, Springer, Oct. 2013, pp. 119–133.

[119] H. Vijayakumar, X. Ge, M. Payer, and T. Jaeger, "JIGSAW: Protecting Resource Access by Inferring Programmer Expectations", in *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA: USENIX Association, Aug. 2014, pp. 973–988.

[120] C. Pu and J. Wei, "A Methodical Defense against TOCTTOU Attacks: The EDGI Approach", in *Proceedings of the 2006 International Symposium on Secure Software Engineering*, May 2006.

[121] X. Zhou, G. Li, K. Lu, and S. Wang, "Enhancing the Security of Parallel Programs via Reducing Scheduling Space", in *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, 2014, pp. 133–138. DOI: 10.1109/DASC.2014.33.

[122] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel, "Operating System Transactions", in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '0)*, New York, NY, USA: Association for Computing Machinery, 2009, pp. 161–176, ISBN: 9781605587523. DOI: 10.1145/1629575.1629591.

[123] T. Kim and N. Zeldovich, "Practical and Effective Sandboxing for Non-root Users", in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, San Jose, CA: USENIX Association, Jun. 2013, pp. 139–144, ISBN: 978-1-931971-01-0.

[124] A. Bijlani and U. Ramachandran, "A Lightweight and Fine-Grained File System Sandboxing Framework", in *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys '18)*, Jeju Island, Republic of Korea, 2018, ISBN: 9781450360067. DOI: 10.1145/3265723.3265734.

[125] C. Mulliner and B. Michéle, "Read It Twice! A Mass-Storage-Based TOCTTOU Attack", in *6th USENIX Workshop on Offensive Technologies (WOOT 12)*, E. Bursztein and T. Dullien, Eds., Bellevue, WA: USENIX Association, Aug. 2012.

[126] Y. Lee et al., "Ghost Installer in the Shadow: Security Analysis of App Installation on Android", in *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017, pp. 403–414. DOI: 10.1109/DSN.2017.33.

[127] N. Borisov and R. Johnson, "Fixing Races for Fun and Profit: How to Abuse atime", in *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD: USENIX Association, Jul. 2005. [Online]. Available: https://www.usenix.org/conference/14th-usenix-security-symposium/fixing-races-fun-and-profit-how-abuse-atime.

[128] X. Cai, Y. Gui, and R. Johnson, "Exploiting Unix File-system Races via Algorithmic Complexity Attacks", in *30th IEEE Symposium on Security and Privacy*, 2009, pp. 27–41. DOI: 10.1109/SP.2009.10.

[129] R. Bisbey and D. Hollingsworth, "Protection Analysis Project Final Report", *ISI/RR-78-13, DTIC AD A*, vol. 56816, 1978.

[130] N. Viennot et al., *RacePro*, [Online; https://github.com/columbia/racepro, accessed on January 28, 2025.] 2011.

[131] J. A. Kupsch and B. P. Miller, *Safefile Library and Documentation*, [Online; https://research.cs.wisc.edu/mist/safefile/, accessed on January 28, 2025.] 2008.

[132] A. Dunn and D. Porter, *Operating System Demonstrating System Transactions*, [Online; https://github.com/ut-osa/txos, accessed on January 28, 2025], 2017.
[133] H. Vijayakumar et al., *Process Firewall*, [Online; https://github.com/siis/pfwall, accessed on January 28, 2025.] 2014.
[134] T. Kim and N. Zeldovich, *Mbox*, [Online; https://pdos.csail.mit.edu/archive/mbox/, accessed on January 28, 2025.] 2013.
[135] A. Bijlani et al., *SandFS. A File System Sandboxing Framework*, [Online; https://sandfs.github.io/, accessed on January 28, 2025], 2019.
[136] L. Lu, A. C. Arpaci-dusseau, R. H. Arpaci-dusseau, and S. Lu, "A Study of Linux File System Evolution 1 Introduction", in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, San Jose, CA: USENIX Association, Feb. 2013, pp. 31–44.
[137] S. Designer et al., *Kernel Patches from the Openwall Project*, [Online; https://www.openwall.com/linux/, accessed on October 28, 2021], 2002.
[138] A. Yokoyama et al., "Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion", in *Research in Attacks, Intrusions, and Defenses*, F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, Eds., Cham: Springer International Publishing, 2016, pp. 165–187, ISBN: 978-3-319-45719-2.
[139] R. J. Adams, P. Smart, and A. S. Huff, "Shades of grey: Guidelines for working with the grey literature in systematic reviews for management and organizational studies", *International Journal of Management Reviews*, vol. 19, no. 4, pp. 432–454, 2017.
[140] Q. Mahood, D. Van Eerd, and E. Irvin, "Searching for grey literature for systematic reviews: Challenges and benefits", *Research Synthesis Methods*, vol. 5, no. 3, pp. 221–234, 2014.
[141] M. Botacin, "What do malware analysts want from academia? a survey on the state-of-the-practice to guide research developments", in *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '24, Padua, Italy: Association for Computing Machinery, 2024, pp. 77–96, ISBN: 9798400709593.
[142] CERT Estonia (CERT-EE), *Cuckoo3 - Malware analysis tool*, [Online; https://github.com/cert-ee/cuckoo3, accessed on February 2, 2025].
[143] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system", in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.
[144] Polska CERT (CERT.PL, *DRAKVUF Sandbox*, [Online; https://github.com/CERT-Polska/drakvuf-sandbox, accessed on February 2, 2025].
[145] Brian Baskin, *Noriben - Portable, Simple, Malware Analysis Sandbox*, [Online; https://github.com/Rurik/Noriben, accessed on February 2, 2025].
[146] ANYRUN FZCO, *ANY.run*, [Online; https://any.run/, accessed on February 2, 2025].
[147] Hybrid Analysis GmbH (Crowdstrike), *Hybrid Analysis*, [Online; https://www.hybrid-analysis.com/, accessed on February 2, 2025].

[148] J. LLC, *Joe Sandbox*, [Online; February2,2025, accessed on February 2, 2025].
[149] Recorded Future, *Triage*, [Online; https://tria.ge/, accessed on February 2, 2025].
[150] OPSWAT Inc, *Filescan.IO*, [Online; https://www.filescan.io/scan, accessed on February 2, 2025].
[151] Malwation Cyber Security Technology Inc, *Threat.Zone - Advanced Online Malware Sandbox & CDR Platform*, [Online; https://threat.zone/, accessed on February 2, 2025].
[152] Norman, *Norman Antivirus & Security Software for Home & Busines*, [Online; https://www.norman.com/, accessed on February 3, 2025].
[153] C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox", *IEEE Security & Privacy*, vol. 5, no. 2, pp. 32–39, 2007.
[154] Claudio Guarnieri et al., *Cuckoo Sandbox*, [Online; https://cuckoosandbox.org/about, accessed on February 3, 2025].
[155] T. Mandl, U. Bayer, and F. Nentwich, "ANUBIS ANalyzing unknown BInarieS the automatic way", in *Virus bulletin conference*, vol. 1, 2009, p. 02.
[156] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic analysis of malicious code", *Journal in Computer Virology*, vol. 2, no. 1, pp. 67–77, Aug. 2006, ISSN: 1772-9904.
[157] Joxean Koret, *Zero Wine: Malware Behavior Analysis*, [Online; https://zerowine.sourceforge.net/, accessed on February 3, 2025], 2009.
[158] Chae Jong Bin, *Zero Wine Tryouts: An open source malware analysis tool*, [Online; https://zerowine-tryout.sourceforge.net/, accessed on February 3, 2025], 2013.
[159] V. Iyengar, M. Koser, R. Binjve, and A. Gat, *Detux: The Multiplatform Linux Sandbox*, [Online; https://github.com/detuxsandbox/detux, accessed on September 24, 2024].
[160] Tencent, *HaboMalHunter: Habo Linux Malware Analysis System*, [Online; https://github.com/Tencent/HaboMalHunter, accessed on February 3, 2025].
[161] Monnappa K A, *Limon - Sandbox for Analyzing Linux Malwares*, [Online; https://github.com/monnappa22/Limon, accessed on February 3, 2025].
[162] Daniel Uhříček, *LiSa – Multiplatform Linux Sandbox for Analyzing IoT Malware*, [Online; https://excel.fit.vutbr.cz/submissions/2019/058/58.pdf, accessed on February 3, 2025].
[163] S. Yonamine., Y. Taenaka., and Y. Kadobayashi., "Tamer: A sandbox for facilitating and automating iot malware analysis with techniques to elicit malicious behavior", in *Proceedings of the 8th International Conference on Information Systems Security and Privacy - ForSE*, INSTICC, SciTePress, 2022, pp. 677–687, ISBN: 978-989-758-553-1.
[164] Patrik Lantz, *DroidBox: Dynamic analysis of Android apps*, [Online; https://github.com/pjlantz/droidbox, accessed on February 3, 2025].
[165] Y. Cui, Y. Sun, and Z. Lin, "Droidhook: A novel api-hook based android malware dynamic analysis sandbox", *Automated Software Engineering*, vol. 30, no. 1, p. 10, Feb. 2023, ISSN: 1573-7535.

[166] Crowdstrike, *Crowdstrike Falcon Sandbox Malware Analysis*, [Online; https://www.crowdstrike.com/wp-content/uploads/2022/12/crowdstrike-falcon-sandbox-data-sheet.pdf, accessed on February 3, 2025], 2021.
[167] OPSWAT Inc, *MetaDefender Sandbox*, [Online; https://www.opswat.com/products/metadefender/sandbox, accessed on February 3, 2025].
[168] Internet 2.0, *Malcore: Simple File Analysis*, [Online; https://malcore.io/, accessed on February 3, 2025].
[169] VMRay, *VMRay*, [Online; https://www.vmray.com/, accessed on February 3, 2025], .
[170] Kaspersky, *Kaspersky Sandbox*, [Online; https://www.kaspersky.com/enterprise-security/malware-sandbox, accessed on February 3, 2025].
[171] Trend Micro Inc, *Deep Discovery Analyzer*, [Online; https://www.trendmicro.com/en_us/business/products/network/advanced-threat-protection/analyzer.html, accessed on February 3, 2025].
[172] Zscaler Inc, *Zscaler Sandbox*, [Online; https://www.zscaler.com/products-and-solutions/cloud-sandbox, accessed on February 3, 2025].
[173] Fortinet Inc, *FortiGuard Sandbox*, [Online; https://www.fortinet.com/support/support-services/fortiguard-security-subscriptions/inline-sandboxing, accessed on February 3, 2025].
[174] Bitdefender, *Bitdefender Sanddbox*, [Online; https://www.bitdefender.com/en-us/oem/sandbox-service, accessed on February 3, 2025].
[175] Reversing Labs, *Spectra Analyze*, [Online; https://www.reversinglabs.com/products/spectra-analyze, accessed on February 3, 2025].
[176] Palo Alto Networks, *Advanced Wildfire*, [Online; https://www.paloaltonetworks.com/network-security/advanced-wildfire, accessed on February 3, 2025].
[177] Sophos Ltd, *Sophos Sandstorm*, [Online; https://sophos.optrics.com/downloads/sophos-sandstorm-datasheet.pdf, accessed on February 3, 2025].
[178] VIPRE Security Group Inc, *ThreatAnalyzer*, [Online; https://vipre.com/products/threat-intelligence/threat-analyzer/, accessed on February 3, 2025].
[179] D. O'Brien, *Symantec Sandboxing*, [Online; https://www.broadcom.com/products/cybersecurity/network/network-protection/sandboxing, accessed on February 3, 2025].
[180] SecondWrite, Inc, *DeepView Sandbox*, [Online; https://www.secondwrite.com/products/deepview-sandbox/, accessed on February 3, 2025].
[181] Trellix, *Trellix Malware Analysis*, [Online; https://www.trellix.com/assets/data-sheets/trellix-malware-analysis.pdf, accessed on February 3, 2025].
[182] Intezer, *The Autonomous SOC Solution for Malware Analysis*, [Online; https://intezer.com/malware-analysis-solution/, accessed on February 3, 2025].
[183] Comodo Security Solutions, Inc, *Valkyrie*, [Online; https://valkyrie.comodo.com/, accessed on February 3, 2025].

[184] mcarmanize, *esfriend: A minimal malware analysis sandbox for macOS*, [Online; https://github.com/mcarmanize/esfriend, accessed on February 3, 2025].

[185] A. Küchler, A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti, "Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes", in *NDSS 2021, Network and Distributed Systems Security Symposium*, ISOC, Virtual, United States: Internet Society, Feb. 2021. DOI: 10.14722/ndss.2021.24475. [Online]. Available: https://hal.science/hal-04611612.

[186] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools", *ACM Comput. Surv.*, vol. 44, no. 2, Mar. 2008, ISSN: 0360-0300.

[187] S. Neuner et al., *Enter sandbox: Android sandbox comparison*, 2014. arXiv: 1410.7749 [cs.CR].

[188] A. A. R. Melvin and G. J. W. Kathrine, "A quest for best: A detailed comparison between drakvuf-vmi-based and cuckoo sandbox-based technique for dynamic malware analysis", in *Intelligence in Big Data Technologies—Beyond the Hype*, J. D. Peter, S. L. Fernandes, and A. H. Alavi, Eds., Singapore: Springer Singapore, 2021, pp. 275–290, ISBN: 978-981-15-5285-4.

[189] S. Ž. Ilić, M. J. Gnjatović, B. M. Popović, and N. D. Maček, "A pilot comparative analysis of the Cuckoo and Drakvuf sandboxes: An end-user perspective", *Vojnotehnički glasnik/Military Technical Courier*, vol. 70, no. 2, pp. 372–392, 2022.

[190] S. Lee, H. Jeon, G. Park, and J. Youn, "Design of automation environment for analyzing various IoT malware", *Tehnički vjesnik*, vol. 28, no. 3, pp. 827–835, 2021.

[191] S. Madan, S. Sofat, and D. Bansal, "Tools and techniques for collection and analysis of internet-of-things malware: A systematic state-of-art review", *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 10, Part B, pp. 9867–9888, 2022, ISSN: 1319-1578.

[192] M. Mehra and D. Pandey, "Event triggered malware: A new challenge to sandboxing", in *2015 Annual IEEE India Conference (INDICON)*, Dec. 2015, pp. 1–6.

[193] J. Juwono, C. Lim, and A. Erwin, "A Comparative Study of Behavior Analysis Sandboxes in Malware Detection", in *International Conference on New Media (CONMEDIA)*, Nov. 2015, p. 73.

[194] K. Yoshioka, Y. Hosobuchi, T. Orii, and T. Matsumoto, "Vulnerability in Public Malware Sandbox Analysis Systems", in *2010 10th IEEE/IPSJ International Symposium on Applications and the Internet*, 2010, pp. 265–268.

[195] R. Mogicato and A. Zermin, *Design and Implementation of a Collaborative, Lightweight Malware Analysis Sandbox using Container Virtualization*, [Online; https://files.ifi.uzh.ch/CSG/staff/vonderassen/extern/theses/mp-zermin-mogicato.pdf, accessed on February 4, 2025], Zürich, Switzerland: Communication Systems Group, Department of Informatics, Feb. 2023.

[196] O. Olowoyeye, *Evaluating Open Source Malware Sandboxes with Linux malware*, [Online; https://hdl.handle.net/10292/11842, accessed on February 4, 2025], 2018.
[197] O. A. Aslan and R. Samet, "A Comprehensive Review on Malware Detection Approaches", *IEEE Access*, vol. 8, pp. 6249–6271, 2020.
[198] P. K. Mvula, P. Branco, G.-V. Jourdan, and H. L. Viktor, "Evaluating Word Embedding Feature Extraction Techniques for Host-Based Intrusion Detection Systems", *Discover Data*, vol. 1, no. 1, p. 2, 2023, ISSN: 2731-6955.
[199] J. Dai, R. K. Guha, and J. Lee, "Efficient Virus Detection Using Dynamic Instruction Sequences", *J. Comput.*, vol. 4, no. 5, pp. 405–414, 2009.
[200] R. Canzanese, S. Mancoridis, and M. Kam, "System Call-Based Detection of Malicious Processes", in *2015 IEEE International Conference on Software Quality, Reliability and Security*, Aug. 2015, pp. 119–124.
[201] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and Efficient Malware Detection at the End Host", in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09, Montreal, Canada: USENIX Association, 2009, pp. 351–366.
[202] E. Amer and I. Zelinka, "A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence", *Computers & Security*, Feb. 2020.
[203] Y. Qiao, Y. Yang, L. Ji, and J. He, "Analyzing Malware by Abstracting the Frequent Itemsets in API Call Sequences", in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2013, pp. 265–270.
[204] Y. Ki, E. Kim, and H. K. Kim, "A Novel Approach to Detect Malware Based on API Call Sequence Analysis", *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, p. 659 101, 2015.
[205] F. O. Catak, A. F. Yazı, O. Elezaj, and J. Ahmed, "Deep learning based Sequential model for malware analysis using Windows exe API Calls", *PeerJ Computer Science*, vol. 6, 2020.
[206] C. W. Kim, "Ntmaldetect: A machine learning approach to malware detection using native api system calls", *ArXiv*, 2018.
[207] R. J. S. Angelo Schranko De Oliveira, "Behavioral Malware Detection using Deep Graph Convolutional Neural Networks", *International Journal of Computer Applications*, vol. 174, no. 29, pp. 1–8, Apr. 2021, ISSN: 0975-8887.
[208] D. Kim and H. K. Kim, "Automated Dataset Generation System for Collaborative Research of Cyber Threat Analysis", *Security and Communication Networks*, vol. 2019, no. 1, p. 6 268 476, 2019.
[209] D. Čeponis and N. Goranin, "Towards a robust method of dataset generation of malicious activity for anomaly-based HIDS training and presentation of AWSCTD dataset", *Baltic Journal of Modern Computing*, vol. 6, no. 3, pp. 217–234, 2018.
[210] K. O'Reilly, *capemon: CAPE's monitor*, [Online; https://github.com/kevoreilly/capemon, accessed on February 4, 2025], 2024.

[211] R. Raducu, R. J. Rodríguez, and P. Álvarez, *CAPE Hook Generator*, [Online; https://github.com/reverseame/cape-hook-generator, accessed on February 4, 2025], version 1.0, Jun. 2024.
[212] *VirusTotal*, [Online; https://www.virustotal.com/, accessed on February 8, 2024].
[213] S. Sebastián and J. Caballero, "AVclass2: Massive Malware Tag Extraction from AV Labels", in *Proceedings of the 36th Annual Computer Security Applications Conference*, ser. ACSAC '20, New York, NY, USA: Association for Computing Machinery, 2020, pp. 42–53, ISBN: 9781450388580.
[214] R. Raducu, A. Villagrasa-Labrador, R. J. Rodríguez, and P. Álvarez, *MALVADA - A Windows Malware Execution Traces Dataset generation framework*, [Online; https://github.com/reverseame/MALVADA, accessed on February 4, 2025], 2024.
[215] K. O'Reilly and A. Brukhovetskyy, *CAPE Sandbox Book*, [Online; https://capev2.readthedocs.io/en/latest/index.html, accessed on February 4, 2025], 2024.
[216] Microsoft, *PE Format*, [Online; https://learn.microsoft.com/en-us/windows/win32/debug/pe-format, accessed on February 5, 2025], 2024.
[217] R. Raducu, A. Villagrasa-Labrador, R. J. Rodríguez, and P. Álvarez, *WinMET Dataset*, [Online; https://doi.org/10.5281/zenodo.12647555, accessed on February 5, 2025], Jul. 2024.
[218] M. Botacin, F. Ceschin, R. Sun, D. Oliveira, and A. Grégio, "Challenges and pitfalls in malware research", *Computers & Security*, vol. 106, p. 102 287, 2021, ISSN: 0167-4048.
[219] A. Zeller, "Program Analysis: A Hierarchy", in *Proceedings of the Workshop on Dynamic Analysis (WODA 2003)*, 2003, pp. 1–4.
[220] Microsoft, *SetWindowsHookExA function (winuser.h)*, [Online; https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowshookexa, accessed on February 6, 2025], Sep. 2023.
[221] B.G.Ryder, "Constructing the Call Graph of a Program", *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 216–226, 1979.
[222] R. Raducu, R. Rodríguez, and P. Álvarez, *Windows Behavior Catalog*, [Online; https://github.com/reverseame/windows-behavior-catalog, accessed on February 6, 2025], 2024.
[223] R. Raducu, R. J. Rodríguez, and P. Álvarez, *Windows API and Syscalls categories*, [Online; https://github.com/reverseame/winapi-categories, accessed on February 6, 2025], Jul. 2024.
[224] R. Raducu, R. J. Rodríguez, and P. Álvarez, *MalGraphIQ*, [Online; https://github.com/reverseame/MalGraphIQ, accessed on February 6, 2025], 2024.
[225] D. C. Kozen, "The Design and Analysis of Algorithms ", in *The Design and Analysis of Algorithms*. New York, NY: Springer New York, 1992, ch. Depth-First and Breadth-First Search, pp. 19–24, ISBN: 978-1-4612-4400-4.
[226] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX", in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11–15.

[227] A. Jahan and K. L. Edwards, "A state-of-the-art survey on the influence of normalization techniques in ranking: Improving the materials selection process in engineering design", *Materials & Design (1980-2015)*, vol. 65, pp. 335–342, 2015, ISSN: 0261-3069.

[228] Roccia, Thomas and Lesueur, Jean-Pierre, *Unprotect Project*, [Online; https://unprotect.it, accessed on January 25, 2025], 2024.

[229] Check Point Software Technologies Ltd., *Evasion Techniques Encyclopedia*, [Online; https://evasions.checkpoint.com/, accessed on January 25, 2025], 2024.

[230] D. S. Cruzes and L. ben Othmane, "Empirical Research for Software Security", in CRC Press, 2017, ch. Threats to Validity in Empirical Software Security Research, p. 26.

[231] A. Elshinbary, *Deep Analysis of GCleaner*, [Online; https://n1ght-w0lf.github.io/malware%20analysis/gcleaner-loader, accessed on February 8, 2025], Jul. 2023.

[232] R. J. Rodríguez, "Evolution and characterization of point-of-sale RAM scraping malware", *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 3, pp. 179–192, Aug. 2017.

[233] N. Huq, *PoS RAM Scraper Malware: Past, Present, and Future*, [Online; https://documents.trendmicro.com/assets/wp/wp-pos-ram-scraper-malware.pdf, accessed on February 8, 2025].

[234] D. O'Brien, *Internet Security Threat Report: Ransomware 2017*, [Online; https://docs.broadcom.com/doc/istr-ransomware-2017-en, accessed on February 8, 2025], 2017.

[235] A. Y. Prasetya, K. I. Aini, and C. Lim, "Comparative Analysis of Attack Behavior Patterns in Petya, CryptInfinite, and Locky Ransomware Using Hybrid Analysis", in *2023 IEEE International Conference on Cryptography, Informatics, and Cybersecurity (ICoCICs)*, 2023, pp. 29–34.

[236] U. Wanve, *GuLoader: Peering Into a Shellcode-based Downloader*, [Online; https://www.crowdstrike.com/blog/guloader-malware-analysis/, accessed on September 29, 2024], Jun. 2020.

[237] N. P. Yturriaga, *GuLoader: The NSIS Vantage Point*, [Online; https://www.trellix.com/blogs/research/guloader-the-nsis-vantage-point/, accessed on September 29, 2024], Jan. 2023.

[238] A. Bukhteyev and A. Olshtein, *Cloud-Based Malware Delivery: The Evolution of GuLoader*, [Online; https://research.checkpoint.com/2023/cloud-based-malware-delivery-the-evolution-of-guloader/, accessed on September 29, 2024], May 2023.

[239] T. Fushiki, "Estimation of prediction error by using K-fold cross-validation", *Statistics and Computing*, vol. 21, no. 2, pp. 137–146, Apr. 2011, ISSN: 1573-1375.

[240] Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel, "Fast malware classification by automated behavioral graph matching", in *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, ser. CSIIRW '10, Oak Ridge, Tennessee, USA: Association for Computing Machinery, 2010, pp. 1–4, ISBN: 9781450300179.

[241] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis", *Journal in Computer Virology*, vol. 7, no. 4, pp. 247–258, Nov. 2011, ISSN: 2274-2042.

[242] Y. Ding, X. Xiaoling, C. Sheng, and L. Ye, "A malware detection method based on family behavior graph", *Computers & Security*, vol. 73, Oct. 2017.

[243] S. D. Nikolopoulos and I. Polenakis, "A graph-based model for malware detection and classification using system-call groups", *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 29–46, Feb. 2017, ISSN: 2263-8733.

[244] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware Analysis via Hardware Virtualization Extensions", in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08, Alexandria, Virginia, USA: Association for Computing Machinery, 2008, pp. 51–62, ISBN: 9781595938107.

[245] M. Ficco, "Detecting IoT Malware by Markov Chain Behavioral Models", in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, 2019, pp. 229–234.

[246] H. Jiang, T. Turki, and J. T. L. Wang, "DLGraph: Malware Detection Using Deep Learning and Graph Embedding", in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2018, pp. 1029–1033.

[247] C. Li et al., "DMalNet: Dynamic malware analysis based on API feature engineering and graph learning", *Computers & Security*, vol. 122, p. 102 872, 2022, ISSN: 0167-4048.

[248] A. Pektaş and T. Acarman, "Malware classification based on API calls and behaviour analysis", *IET Information Security*, vol. 12, no. 2, pp. 107–117, 2018.

[249] M. Alazab, S. Venkataraman, and P. Watters, "Towards Understanding Malware Behaviour by the Extraction of API Calls", in *2010 Second Cybercrime and Trustworthy Computing Workshop*, 2010, pp. 52–59.

[250] S. Gupta, H. Sharma, and S. Kaur, "Malware Characterization Using Windows API Call Sequences", in *Security, Privacy, and Applied Cryptography Engineering*, C. Carlet, M. A. Hasan, and V. Saraswat, Eds., Cham: Springer International Publishing, 2016, pp. 271–280, ISBN: 978-3-319-49445-6.

[251] F. Breitinger, B. Guttman, M. McCarrin, V. Roussev, and D. White, "Approximate Matching: Definition and Terminology", National Institute of Standards and Technology (NIST), Tech. Rep. NIST Special Publication 800-168, May 2014.

[252] H. S. Galal, Y. B. Mahdy, and M. A. Atiea, "Behavior-based features model for malware detection", English, *Journal of Computer Virology and Hacking Techniques*, vol. 12, no. 2, pp. 59–67, 2016.

[253] P. Trinius, T. Holz, J. Göbel, and F. C. Freiling, "Visual analysis of malware behavior using treemaps and thread graphs", in *2009 6th International Workshop on Visualization for Cyber Security*, 2009, pp. 33–38.

[254] M. Lindorfer, A. Di Federico, F. Maggi, P. M. Comparetti, and S. Zanero, "Lines of Malicious Code: Insights into the Malicious Software Industry", in *Proceedings of the 28th Annual Computer Security Applications Confer-*

ence, ser. ACSAC'12, Orlando, Florida, USA: Association for Computing Machinery, 2012, pp. 349–358, ISBN: 9781450313124.

[255] Steven Knuchel (Xylitol), *Alina 3.4 (POS Malware)*, [Online; https://www.xylibox.com/2013/02/alina-34-pos-malware.html, accessed on February 18, 2025], Feb. 2013.

[256] Trend Micro (US), *ALINA*, [Online; https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/alina, accessed on February 18, 2025], Sep. 2014.

[257] Cybersecurity & Infrastructure Security Agency, *Petya Ransomware | CISA)*, [Online; https://www.cisa.gov/news-events/alerts/2017/07/01/petya-ransomware, accessed on February 18, 2025], 2017.

[258] Varutra, *Petya Ransomware Attack - Threat Advisory Report)*, [Online; https://www.varutra.com/threat-advisory-report-on-petya-ransomware-critical-severity/, accessed on February 18, 2025], 2017.

[259] Josh Grunzweig, *Alina: Casting a Shadow on POS*, [Online; https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/alina-casting-a-shadow-on-pos/, accessed on February 19, 2025], May 2013.

[260] Josh Grunzweig, *Alina: Following The Shadow Part 2*, [Online; https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/alina-following-the-shadow-part-2/, accessed on February 19, 2025], Jun. 2013.

[261] Black Lotus Labs, *Alina Point of Sale Malware Still Lurking in DNS*, [Online; https://blog.lumen.com/alina-point-of-sale-malware-still-lurking-in-dns, accessed on February 18, 2025], Jul. 2020.

[262] Dell SecureWorks Counter Threat Unit(TM) Threat Intelligence, *Point-of-Sale Malware Threat Analysis*, [Online; https://www.secureworks.com/research/point-of-sale-malware-threats, accessed on February 18, 2025], May 2013.

[263] K. Sood and S. Hurley, *NotPetya Ransomware Attack [Technical Analysis]*, [Online; https://www.crowdstrike.com/en-us/blog/petrwrap-ransomware-technical-analysis-triple-threat-file-encryption-mft-encryption-credential-theft/, accessed on February 18, 2025], 2017.

Appendix A

Occurrences Normalization and Visualization

The pattern matching process produces the number of *occurrences* of each behavior pattern from the WBC against a given graph. Before transforming the data into their corresponding visualizations, execution traces deemed incomplete (i.e., those that correspond to a failed execution) are discarded. In large-scale malware analysis, failures can occur for various reasons, such as sandbox component crashes, virtualization engine issues, or memory exhaustion. We empirically determine that an execution trace matching at least 10% of the WBC is sufficient to be considered a successful execution. This threshold is applied to filter out incomplete results.

The data undergoes a transformation process in which the *score* of each micro-objective or micro-behavior is calculated. Hereafter, the term *category* refers to either of these concepts. First, the occurrence values are clipped to the 0.9 quantile to reduce the influence of outliers. A linear normalization [227] is then applied to scale the values uniformly. This step is essential because **the cardinality of occurrences differs across categories**. That is, the maximum number of occurrences is inconsistent. As an example, suppose that the scores for `Cryptography` range from 0 to 25, while those for `Filesystem` range from 0 to 3200. In this context, 17 occurrences in `Cryptography` could be as significant as 1600 occurrences in `Filesystem`.

Therefore, applying a consistent scale is necessary to ensure comparability across categories. To achieve this, data is normalized on a per-category basis, using the values from all samples[1] for each specific category. Linear normalization is applied to scale the data to the range $[0, 100]$:

$$n_{ij} = \frac{r_{ij}}{r_j^{\max}}(\max - \min) + \min$$

where n_{ij} is the normalized value, r_{ij} is the number of *occurrences* for sample i in category j, $r_j^{\max}$ is the maximum number of occurrences across all samples for

[1] In this context, the terms *sample* and *execution trace* are used interchangeably, as each execution trace represents the execution of a single sample.

category j, max $= 100$, and min $= 0$. This normalization preserves the minimum nonzero value $r_j^{\min}$, which would otherwise be mapped to 0 under standard Min–Max normalization. This is undesirable as it would mask the presence of the categories with the lowest occurrences. The normalization ensures that $n_{ij} = 0 \iff r_{ij} = 0$. As a result, all values are adjusted to the range $[j^{\min}, 100]$, where $j^{\min}$ is the smallest normalized value for category j, and $0 \leq j^{\min} \leq 100$.

The final step is to compute a representative value for each category **based on the values of all the samples**. First, the arithmetic mean is calculated for each category. Then, the mean is expressed as a percentage relative to the sum of means across all categories. We refer to this result as the *score* of the category. This approach is referred to as *per-category normalization*, as the score is derived from the values of each category across all samples. Other normalization techniques were considered, as discussed below.

We also evaluated *per-sample normalization*, an alternative approach that assigns a score to each category within a single sample, instead of assigning a score to each sample within a single category. While per-category normalization is the default behavior for MALGRAPHIQ, this behavior can be modified as documented in the MALGRAPHIQ's repository [224]. Each normalization method focuses on different aspects:

- Per-category normalization compares how samples perform within the same category.
- Per-sample normalization shows the relative importance of categories within each sample.

The following sections illustrate and compare both normalization approaches. First, we describe how data is transformed through each processing step before generating the visualizations. For clarity, we use the three test execution traces provided in MALGRAPHIQ's repository. We then present the visual differences between the plots produced using per-category and per-sample normalization for these samples. Finally, we demonstrate the effect of processing a single execution trace. In the examples, CRYPTO, COMM, and OS stand for CRYPTOGRAPHY, COMMUNICATION, and OPERATING SYSTEM, respectively.

A.1 Multiple Execution Traces

Table A.1 presents the number of occurrences of each micro-objective from the WBC for the three execution traces. Table A.2 shows the results of clipping the data to the 0.9 quantile using *midpoint* interpolation. That is, any value exceeding this threshold is replaced by $\frac{i+j}{2}$, where i is the value corresponding to the 0.9 quantile of the category and j is the actual value before clipping. Values below the threshold remain unchanged. This approach preserves data structure while reducing the impact of outliers.

Afterward, the data is normalized to the range $[j^{\min}, 100]$. Table A.3 shows the results of per-category normalization, while Table A.4 presents the results of per-sample normalization.

Table A.1 Original occurrences.

#	FILESYSTEM	CRYPTO	COMM	MEMORY	PROCESS	OS
1	64	0	0	27	14	18
2	2	0	0	4	2	4
3	327	5	0	264	44	223

Table A.2 Clipped occurrences.

#	FILESYSTEM	CRYPTO	COMM	MEMORY	PROCESS	OS
1	64.0	0.0	0	27.0	14.0	18.0
2	2.0	0.0	0	4.0	2.0	4.0
3	195.5	2.5	0	145.5	29.0	120.5

Per-category normalization scales the values for each category across all samples (column-wise). For example, sample #3 from Table A.2 has a value of 195.5 in the FILESYSTEM category. Since this is the highest value in that category, it is assigned as max (100), and the values for other samples within the same category are scaled proportionally. This process is repeated for each category. In contrast, per-sample normalization scales the values for each sample across all categories (row-wise). Consider sample #2 from Table A.2, for which both MEMORY and OPERATING SYSTEM have a value of 4.0. Since this is the highest value in that sample, it is considered as max, and the values for the remaining categories within the same sample are scaled relative to it.

Table A.3 Per-category normalization.

#	FILESYSTEM	CRYPTO	COMM	MEMORY	PROCESS	OS
1	32.74	0.00	0.00	18.56	48.28	14.94
2	1.02	0.00	0.00	2.75	6.90	3.32
3	100.00	100.00	0.00	100.00	100.00	100.00

Table A.4 Per-sample normalization.

#	FILESYSTEM	CRYPTO	COMM	MEMORY	PROCESS	OS
1	100.00	0.00	0.00	42.19	21.88	28.13
2	50.00	0.00	0.00	100.00	50.00	100.00
3	100.00	1.28	0.00	74.43	14.83	61.64

After normalizing the data, the arithmetic mean is calculated for each category. Let us recall that the goal of this process is to transform the data so that the resulting visual representations reflect the relative score of each category with respect to the overall behaviors observed during execution. When multiple execution traces are processed, the score for each category is obtained by computing the arithmetic mean of the normalized values across all samples. Table A.5 presents the arithmetic means from both normalization methods: per-category (Table A.3) and per-sample (Table A.4). Before plotting the data, the mean values are converted into percentages, producing the final score for each category, as shown in Table A.6.

Table A.5 Arithmetic mean score.

Micro-objective	Table A.3	Table A.4
FILESYSTEM	44.59	83.33
CRYPTOGRAPHY	33.33	0.43
COMMUNICATION	0.00	0.00
MEMORY	40.44	72.20
PROCESS	51.72	28.90
OPERATING SYSTEM	39.42	63.25

Table A.6 Final percent score.

Micro-objective	Table A.3 score (%)	Table A.4 score (%)
FILESYSTEM	21.28	33.59
CRYPTOGRAPHY	15.91	0.17
COMMUNICATION	0.00	0.00
MEMORY	19.30	29.10
PROCESS	24.69	11.65
OPERATING SYSTEM	18.82	25.49

The final scores are then plotted into their corresponding visualizations, in which the difference between both normalization techniques becomes more evident. Figure A.1 depicts the micro-objective scores from Table A.6. For illustrative purposes, we also included a comparison between the visualizations of the PROCESS' micro-behaviors in Figure A.2.

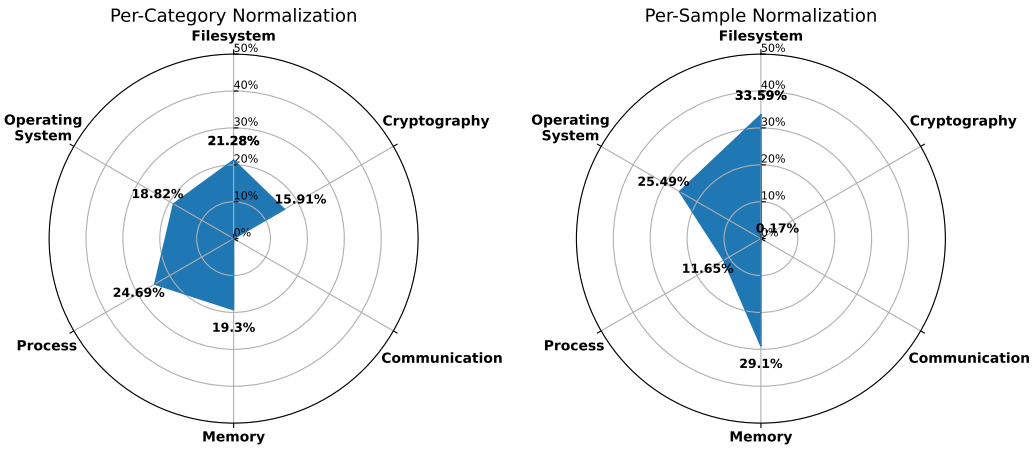


Figure A.1 Micro-objectives visualizations (scores from Table A.6).

A.2 Single Execution Trace

When MALGRAPHIQ is used to analyze a single execution trace, the resulting visualizations may lack clarity. To demonstrate this, we present the results of processing a single execution trace using both normalization methods. For simplicity, all processing steps described in Tables A.1 to A.6 are consolidated into a

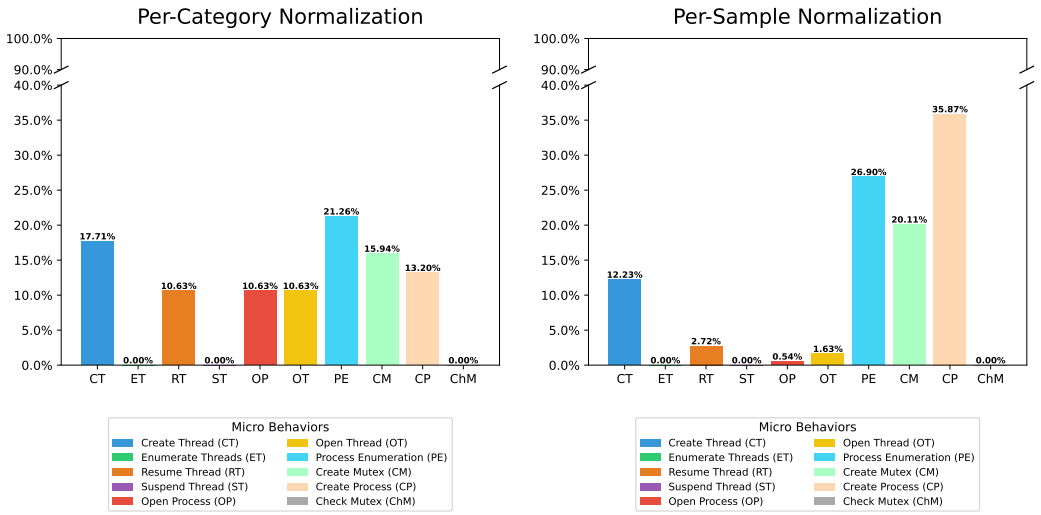


Figure A.2 PROCESS' micro-behaviors visualizations.

single table (Table A.7). The final category scores are visualized in Figure A.3. Additionally, for illustration, Figure A.4 shows a comparison of the PROCESS micro-objective derived from the same single execution trace.

Table A.7 Summary of occurrences transformation into score.

	FILESYSTEM	CRYPTO	COMM	MEMORY	PROCESS	OS
Occurrences	327	5	0	264	44	223
Per-category						
Normalization	100.0	100.0	0.0	100.0	100.0	100.0
Mean	100.0	100.0	0.0	100.0	100.0	100.0
Score (%)	20.0	20.0	0.0	20.0	20.0	20.0
Per-sample						
Normalization	100.0	1.53	0.0	80.73	13.46	68.20
Mean	100.0	1.53	0.0	80.73	13.46	68.20
Score (%)	37.89	0.58	0.0	30.59	5.10	25.84

With per-category normalization, all categories with nonzero occurrences appear equally represented because the max and min values are identical. This occurs because, with only one sample, there are no other samples for comparison within each category. As a result, all categories are balanced, and the single sample receives the same relative score across them. This limitation should be considered when interpreting single-sample outputs. It also highlights the need for further research to refine behavior representations for single-trace analysis.

In contrast, per-sample normalization preserves the initial occurrence counts, which exposes the cardinality imbalance between categories. Categories with lower occurrence counts are assigned lower final scores. For example, the CRYPTO micro-behavior from Table A.7 has a 0.58% score, making it nearly imperceptible. This is undesirable as it perpetuates the pattern cardinality bias: categories with fewer behavior patterns (and thus fewer possible occurrences) are not necessarily less relevant, but still receive lower scores.

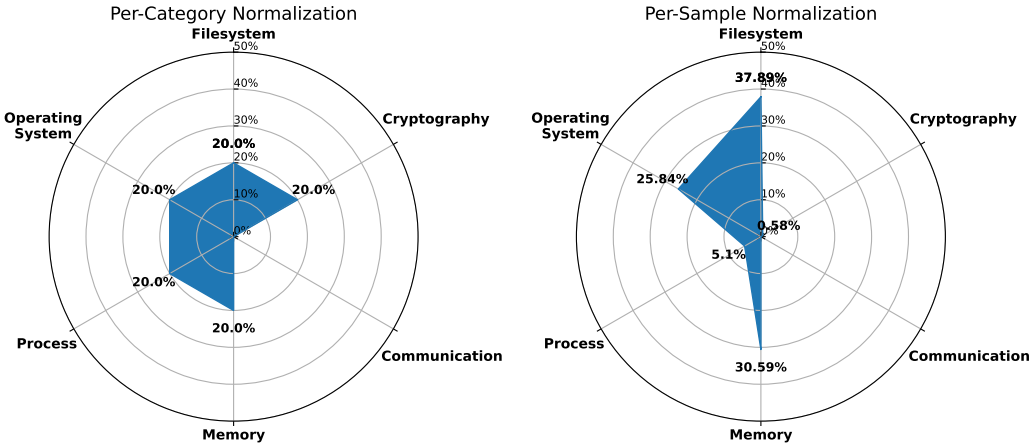


Figure A.3 Micro-objectives visualizations for one sample.

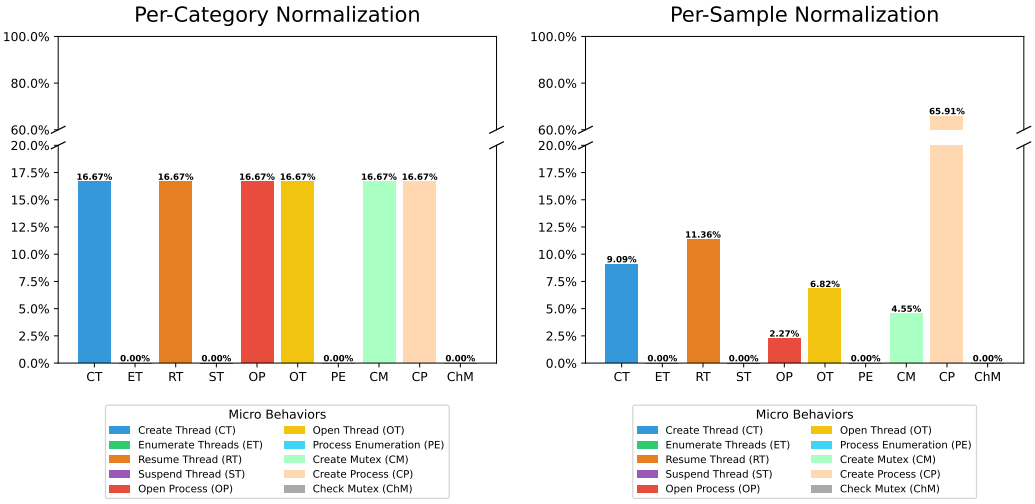


Figure A.4 PROCESS micro-behavior visualizations for one sample.

A.3 Conclusion

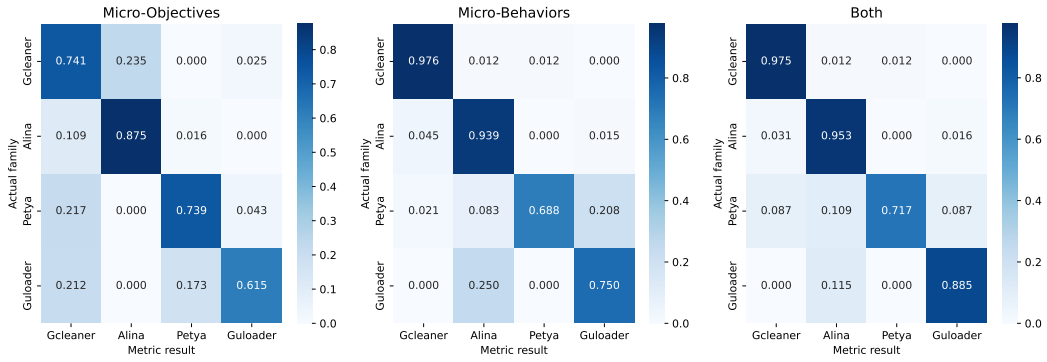
The analysis in the preceding sections highlights the differences between the two normalization methods and some limitations of our approach. Per-category normalization is suitable for comparing the same category across multiple samples. However, it becomes uninformative when applied to a single execution trace, as all categories appear uniformly distributed. In contrast, per-sample normalization is useful for identifying the distribution of behaviors within a single sample but loses comparability across samples and reinforces the cardinality bias.

Due to these observations, per-category normalization is selected as the default in MALGRAPHIQ, as it better reflects category significance when analyzing multiple execution samples. Nevertheless, for single execution traces, the results can be misleading. This remains an open research problem that requires further investigation.

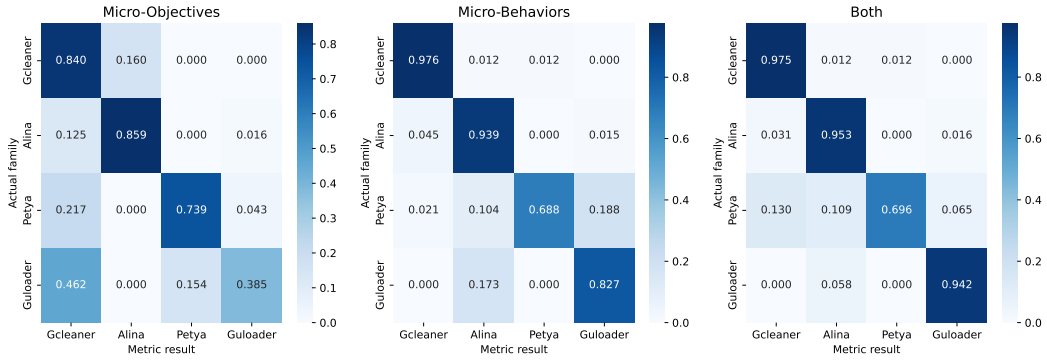
Appendix B

Confusion Matrices

Figures B.1 to B.3 show the confusion matrices from our experiments applying κCV with $\kappa = 5$ and $\kappa = 10$ for the `Gcleaner`, `Alina`, `Petya`, and `Guloader` families using *Micro-Objectives*, *Micro-Behaviors*, or *Both* behavior vectors, with results measured using different metrics (specifically, cosine similarity, Euclidean distance and Manhattan distance). As discussed in Section 6.4.2, higher dimensionality of the employed behavior vectors is usually better. That is, using *Both* (36 dimensions) behavior vectors typically performs better than *Micro-Behaviors* (30 dimensions), and *Micro-Behaviors* perform better than *Micro-Objectives* (6 dimensions).

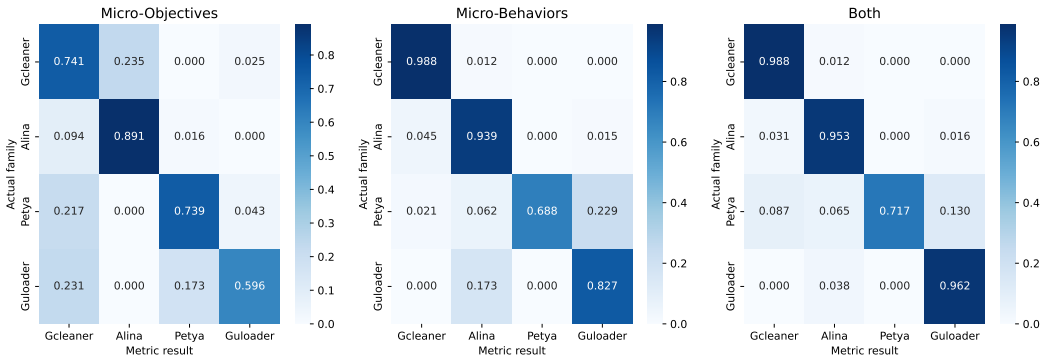


(a) Cosine similarity $\kappa = 5$

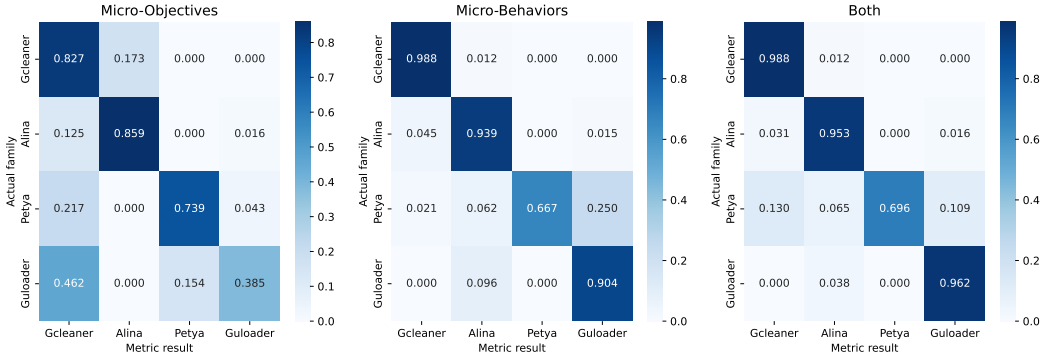


(b) Cosine similarity $\kappa = 10$

Figure B.1 Confusion matrices of different κ—folds and behavior vectors, using cosine similarity.

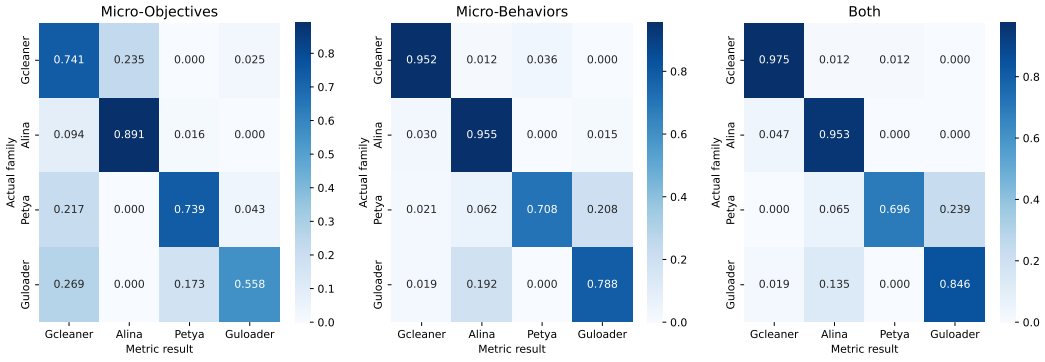


(a) Euclidean distance $\kappa = 5$

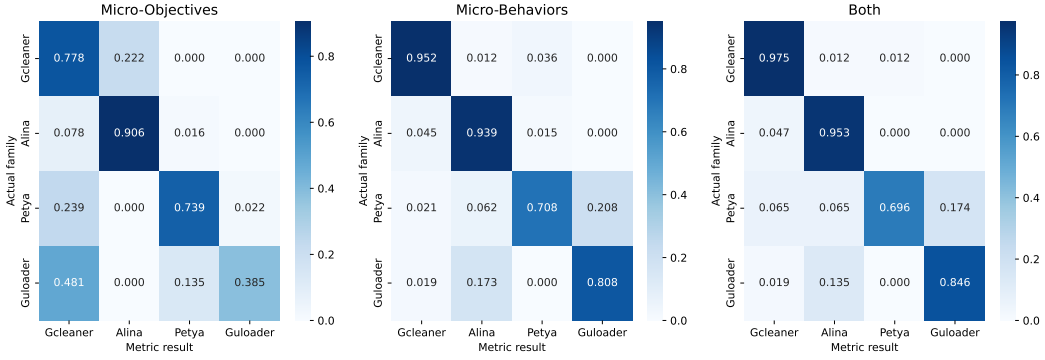


(b) Euclidean distance $\kappa = 10$

Figure B.2 Confusion matrices of κ—folds and behavior vectors, using Euclidean distance.



(a) Manhattan distance $\kappa = 5$



(b) Manhattan distance $\kappa = 10$

Figure B.3 Confusion matrices of κ—folds and behavior vectors, using Manhattan distance.

Appendix C

Behavior Visuals

In this section we provide the remaining of the generated visuals during our experimentation (Section 6.4). Namely, the micro-behavior level representations for the FILESYSTEM, CRYPTOGRAPHY, COMMUNICATION, MEMORY, PROCESS, and OPERATING SYSTEM micro-objectives.

These visual representations are intended to assist users in understanding the behaviors exhibited by a given set of binaries. In the context of malware analysis, MALGRAPHIQ can support the characterization of malware families. The micro-objective visualizations (Figure 6.3) can be leveraged to characterize malware families (i.e, discover which behavior categories predominate a given family), while the micro-behavior visualizations (Figures C.1 to C.6) reveal specific behaviors observed during execution.

For example, high scores in the FILESYSTEM and MEMORY micro-objectives may suggest the analyzed execution samples are ransomware, droppers or worms, as these types of malware often perform intensive I/O operations and allocate memory to unpac payloads. Higher PROCESS and OPERATING SYSTEM scores could align with privilege escalation or process manipulation activities, commonly employed by malware like rootkits. A strong presence of sc Cryptography likely corresponds to ransomware or malware performing encryption tasks. Finally, the COMMUNICATION category is a clear indication the sample attempted to establish Internet connection(s), a behavior typically performed by spyware, backdoors or malware using command-and-control (C2) communication.

To demonstrate the usefulness of these visualizations, we analyze the `Gcleaner`, `Guloader`, `Alina`, and `Petya` malware families.

Figure 6.3 shows the micro-objective scores for the analyzed malware families. The results indicate that the FILESYSTEM, OPERATING SYSTEM, and MEMORY categories consistently score the highest across families. The PROCESS micro-objective is also present, but its score varied depending on the family. However, identifying the specific behaviors within these categories requires examining the micro-behavior results. The following key differentiating behaviors can be observed for each family:

- CRYPTOGRAPHY is present only in `Petya` and `Guloader`.
- Only `Gcleaner` and `Alina` attempt to perform COMMUNICATION operations.
- While PROCESS is present in all families, `Guloader` seems to be more intenssive, which could be an indicative of techniques like process or thread manipulation.

The remaining categories seem to have similar results across families. In order to find out what behavior generated these results, one must check the micro-behavior level figures (Figures C.1 to C.6).

Regarding the FILESYSTEM micro-behavior (Figure C.1), all analyzed families match with it, though the underlying purposes vary. `Alina`, a POS RAM Scraper, may require unpacking before execution or may copy itself to other locations [255, 256]. Ransomware families such as `Petya` traverse the filesystem to encrypt files and other system objects [234, 235]. `Gcleaner` writes the malicious payload to the `%APPDATA%` folder [231]. `Guloader` performs filesystem operations when downloading and storing malicious files and payloads from cloud services [236, 237, 238].

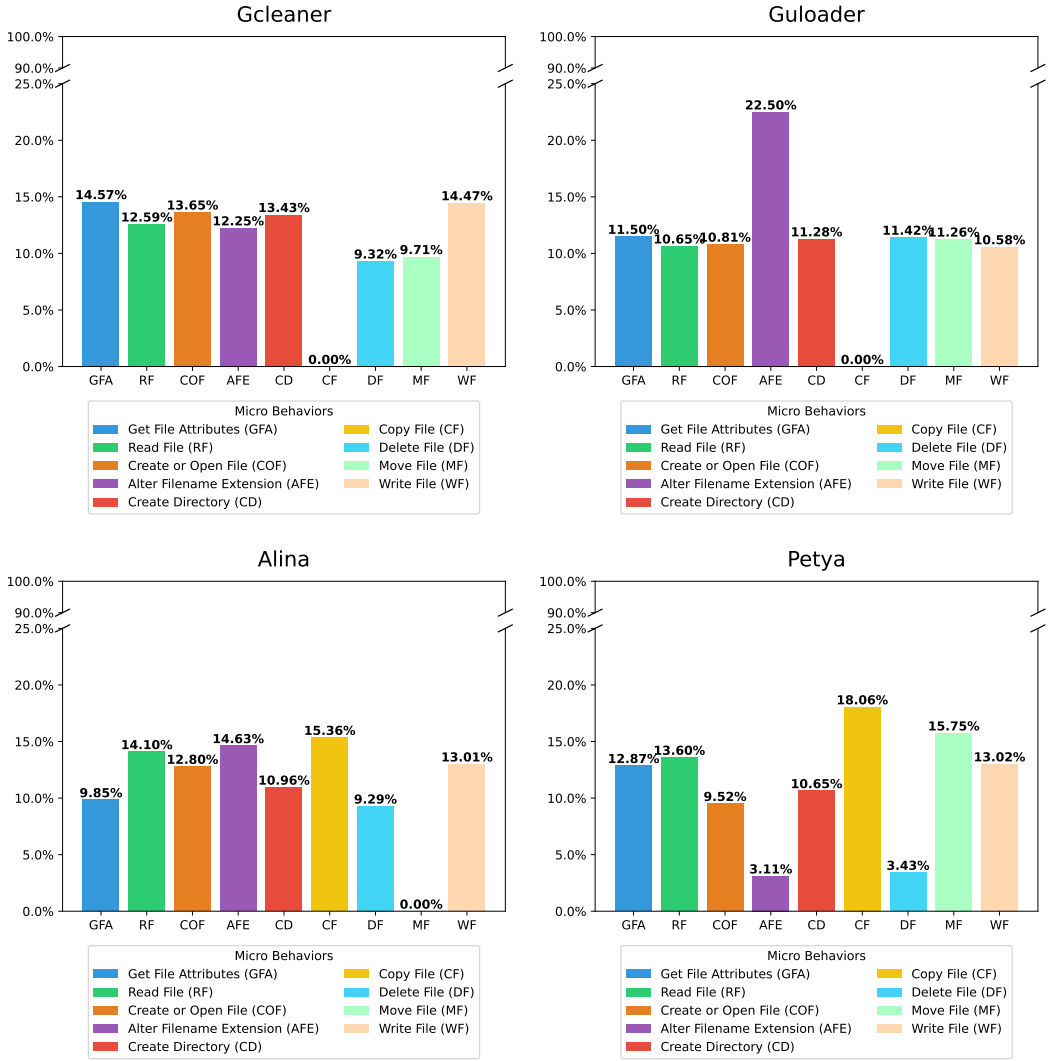


Figure C.1 FILESYSTEM's micro-behavior scores.

The CRYPTOGRAPHY micro-behavior visualization (Figure C.2) shows `Alina` and `Gcleaner` samples exhibit no activity for this category, as they do not perform cryptographic operations. In contrast, `Guloader` [236] and `Petya` [257, 258] show cryptographic behavior. Notably, neither family matches the "Encrypt Data" micro-behavior. This suggests that their cryptographic algorithms or implementations may differ from those defined in the WBC. Similarly, neither matches the "Decrypt Data" micro-behavior. In the case of ransomware like `Petya`, this is expected, as decryption typically occurs only when a ransom is paid.

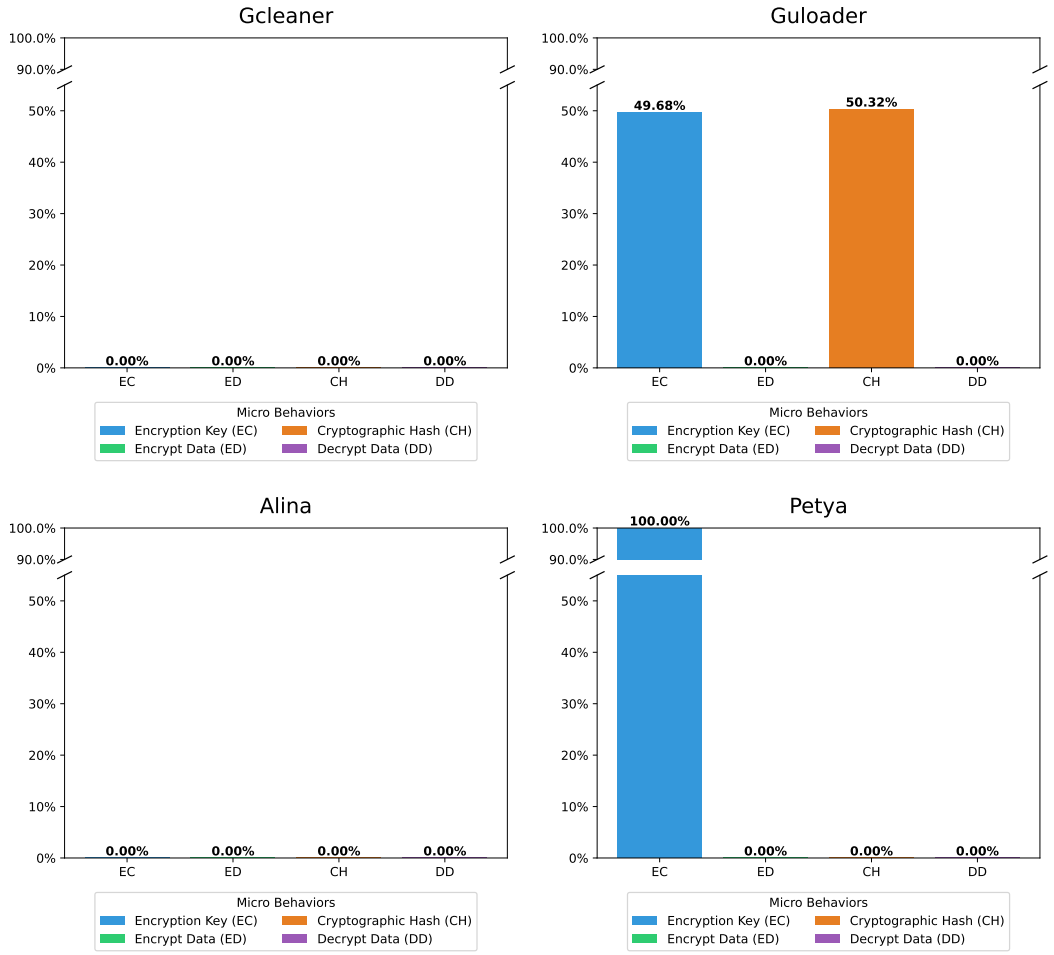


Figure C.2 CRYPTOGRAPHY's micro-behavior scores.

As for the COMMUNICATION micro-behavior (Figure C.3), the results indicate that only `Alina` and `Gcleaner` exhibit network communication activity. For `Alina`, this behavior likely corresponds to the data exfiltration stage [259, 260] or the use of DNS queries to disguise exfiltration traffic [261]. For `Gcleaner`, the activity results from attempts to contact multiple Command and Control (C2) servers [231].

Our experiments show that the MEMORY category (Figure C.4) does not provide distinctive information, as all analyzed families exhibit similar behavior. This aligns with the inherent nature of the Windows operating system, where the execution of any binary involves memory management maneuvers. Nonetheless, this observation suggests a potential refinement or extension of WBC's MEMORY behaviors.

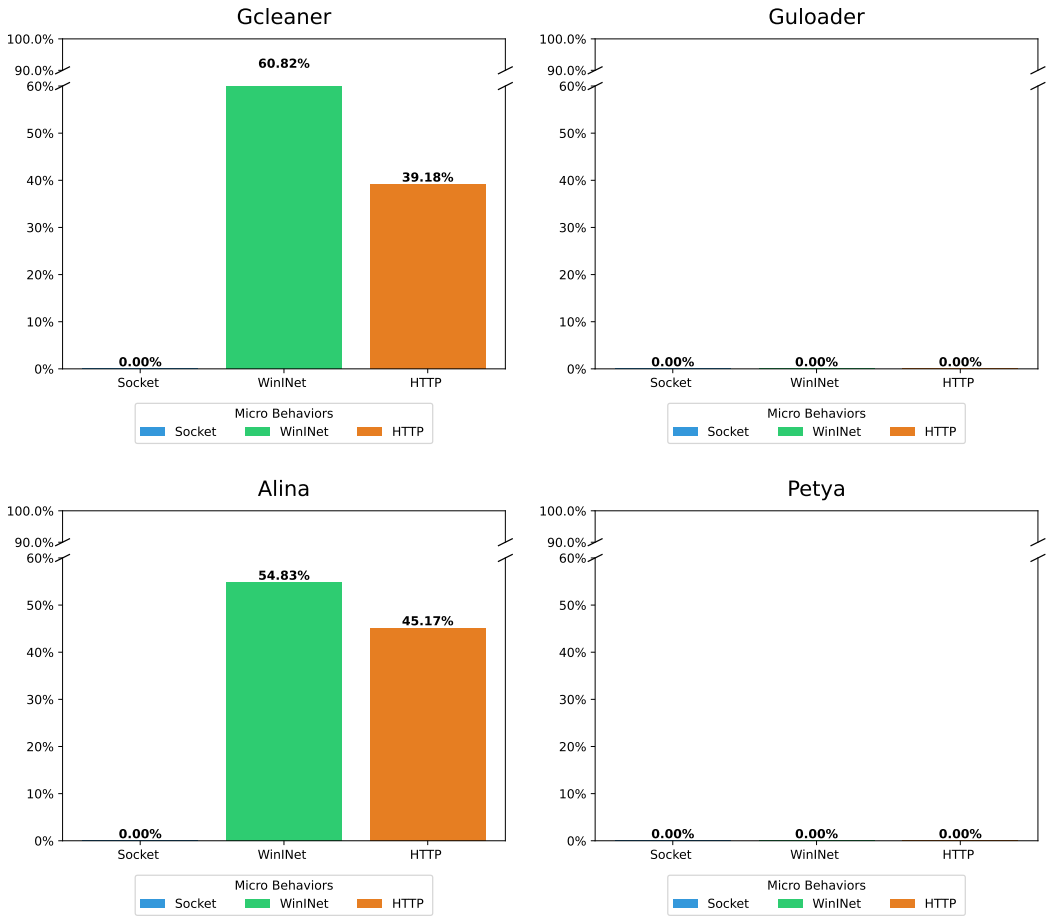


Figure C.3 COMMUNICATION's micro-behavior scores.

Despite the similarity in scores, the families do perform distinct memory-related operations. `Alina` searches the RAM of infected systems using regular expressions to locate Track 1 and Track 2 payment card data [262]. `Gcleaner` downloads payloads and injects them into memory, commonly performing process hollowing or DLL injection [231]. `Guloader` injects shellcode into legitimate processes, performs process hollowing, and manipulates memory regions [236, 237, 238]. `Petya` scrapes memory to extract credentials, loads and manipulates data structures in memory, and allocates memory during execution [234, 263].

The PROCESS results (Figure C.5) show that all families match the "Create Thread", "Create Process", and "Create Mutex" micro-behaviors. Only `Gcleaner` and `Alina` exhibit activity corresponding to the "Open Thread" micro-behavior. All families, except `Petya`, match the "Check Mutex" micro-behavior. `Guloader` is the only family that does not perform "Process Enumeration", while `Gcleaner` is the only one that does not match the "Resume Thread" micro-behavior. None of the families exhibit the "Enumerate Threads" or "Suspend Thread" micro-behaviors.

The observed behaviors align with the documented characteristics of these families. `Alina` iterates over processes as part of its data search stage and creates mutexes to detect prior infections [260]. `Gcleaner` executes multiple payloads by spawning new processes or injecting into existing ones [231]. `Guloader` cre-

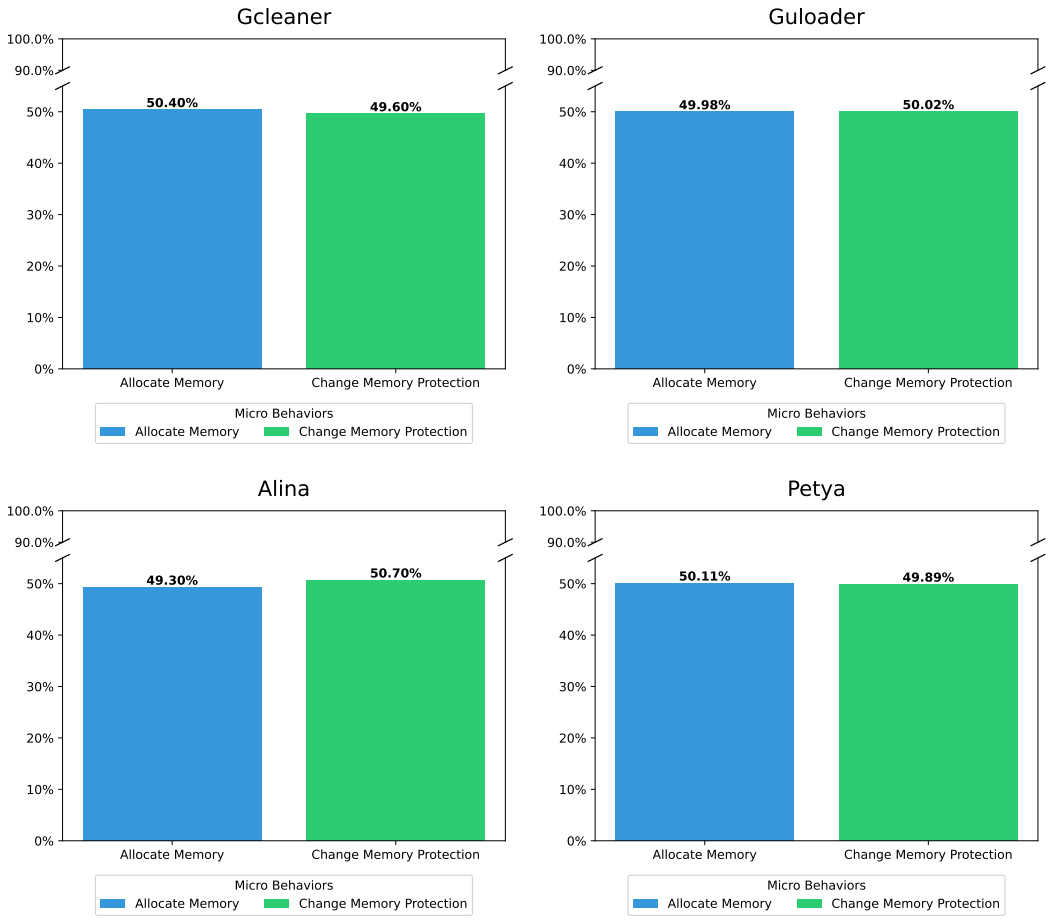


Figure C.4 MEMORY's micro-behavior scores.

ates new processes and opens existing ones [238]. `Petya` launches processes to facilitate propagation and opens existing processes for credential theft [234, 263].

Regarding the OPERATING SYSTEM micro-objective (Figure C.6), all matches for `Alina` correspond to the "Registry" micro-behavior. This aligns with the family's known persistence technique, which involves adding entries to system startup registry keys to ensure execution upon system reboot [232, 256]. The remaining families exhibit activity in both "Registry" and "Environment Variables" micro-behaviors.

Although the results based on matches against WBC behavior patterns provide useful insights, they must be interpreted with caution, as the possibility of false positives or false negatives is not zero. For instance, neither `Guloader` nor `Petya` matched any COMMUNICATION behavior. This does not necessarily imply the absence of network activity. Rather, it may indicate that the malware employs advanced techniques to propagate across local networks or communicates using methods not covered by the Windows API or system calls defined in the WBC. Alternatively, it could result from a detonation failure during analysis.

Similarly, high scores in categories such as FILESYSTEM andMEMORY may reflect behaviors related to standard operating system processes, such as the loader,

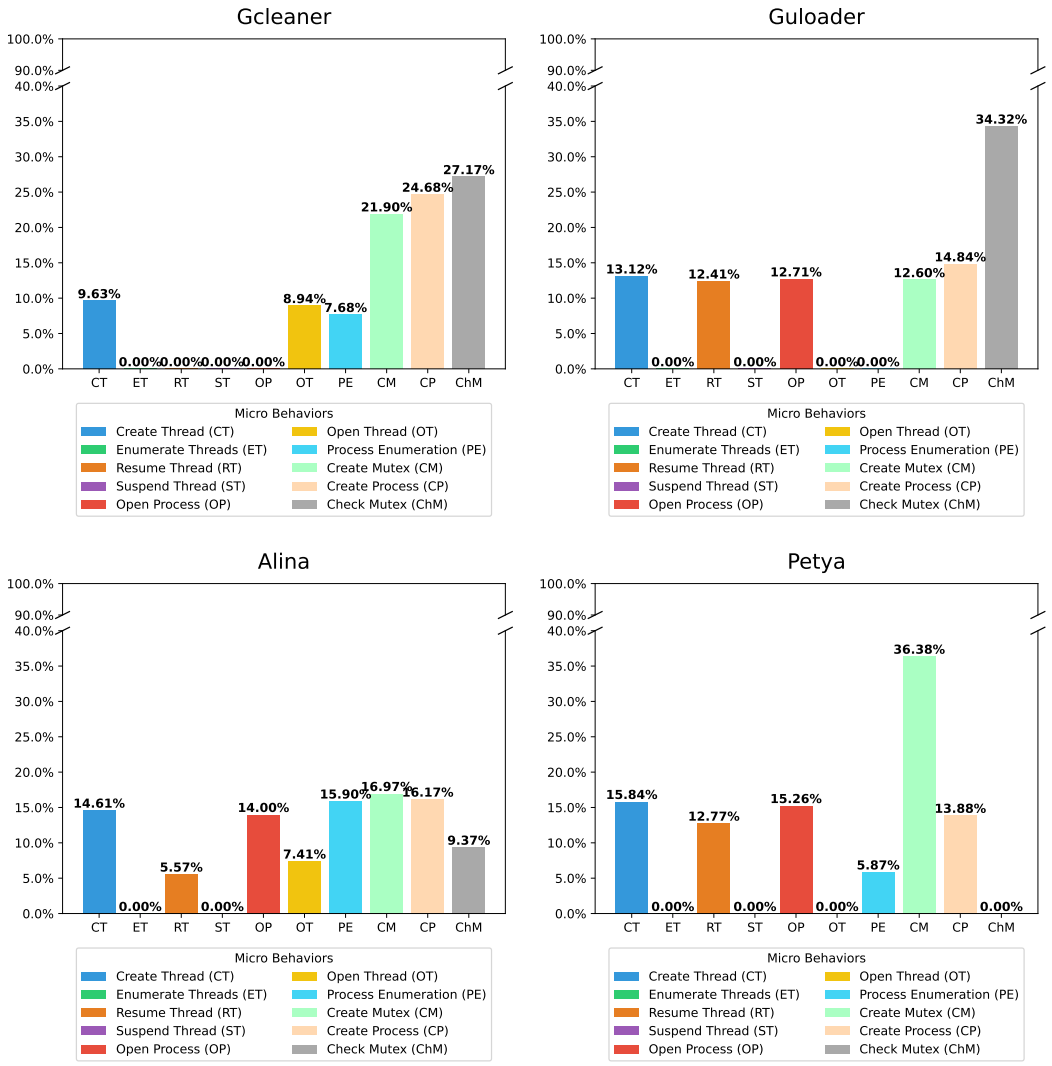


Figure C.5 PROCESS' micro-behavior scores.

rather than malware-specific activity. This consideration applies to any micro-objective or micro-behavior, as discussed in Section 6.3.2 and Section 6.4.3.

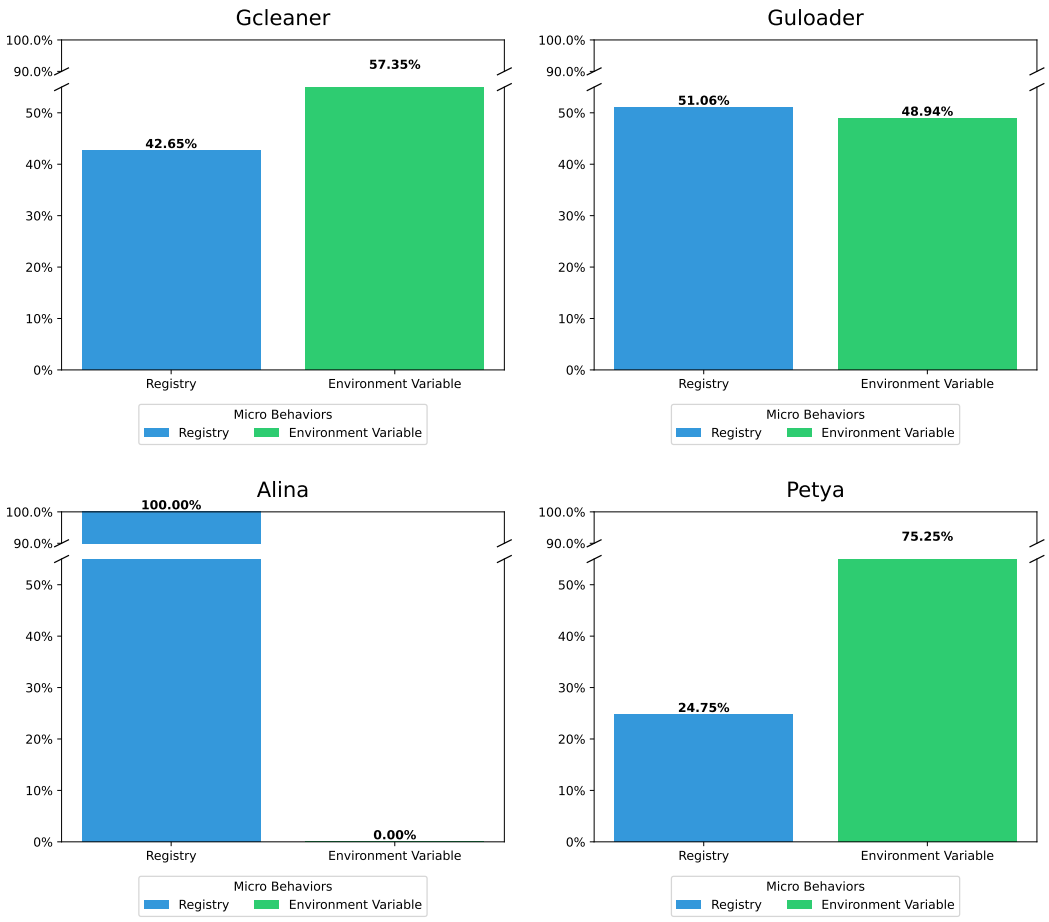


Figure C.6 OPERATING SYSTEM's micro-behavior scores.