POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

A Framework for the Analysis of File Infection Malware

Supervisors:

Prof. Cataldo Basile

Dr. Juan Caballero

Candidate:

Lorenzo Ippolito

Academic Year 2023/2024
Torino

Ai miei nonni.

Abstract

Over the past two decades, malicious software, commonly known as malware, has become one of the largest threats to digital systems. File infectors, a class of malware, spread by injecting their malicious code into legitimate executables. Such infected files are routinely collected by cybersecurity vendors. The mixture of malicious and benign code in infected executables makes it challenging to detect and classify file infectors. This thesis presents a novel framework for the analysis of file infectors. Our framework takes a malware sample as input and examines the permanent modifications made to executables within a sandbox environment to determine if the given sample is a file infector. The original and modified executables are input to the classification module that leverages a novel executable differ to compare them and determine the type of file infector (i.e., appender, prepender, impersonator). We evaluate the effectiveness of our framework on 350 executables belonging to 70 malware families.

Acknowledgements

Throughout this journey, I am deeply grateful to the numerous individuals who have supported and guided me.

First and foremost, I want to express my deep appreciation to my two advisors, Prof. Cataldo Basile and Dr. Juan Caballero. I am particularly indebted to Dr. Caballero for his exceptional guidance and analytical mind, which have significantly enriched my academic experience and development as a researcher. I am immensely grateful to Prof. Davide Balzarotti for his invaluable mentorship and inspiring ideas during my time in France, which have been instrumental in shaping my academic journey. I extend heartfelt thanks to all the professors and staff at Politecnico di Torino, Eurecom, and IMDEA Software Institute for their dedication and expertise. Their contributions have played a crucial role in shaping me into the academic individual I am today.

To my newfound friends from diverse corners of the globe, I offer my sincere appreciation for enriching my life immeasurably through cultural exchange and companionship during these five remarkable years. To my longtime friends, who have been my pillars of strength and comfort throughout this journey, I express my gratitude for your consistent support and encouragement.

I reserve my deepest and sincerest gratitude for my family. I am certain that my grandparents would share the same pride in the person I have become, just as much as my aunts and uncles do today. Lastly, I am profoundly grateful to my parents. No words can adequately express all the sacrifices they have made for me, just as no words can capture the depth of gratitude I hold for how they always make me feel fortunate.

Table of Contents

List of Figures

List of Tables

Acronyms

API

Application Programming Interface

AV

Antivirus

COFF

Common Object File Format

CPU

Central Processing Unit

GB

Gigabyte

JSON

JavaScript Object Notation

MS-DOS

Microsoft Disk Operating System

PE

Portable Executable

RAM

Random Access Memory

RVA

Relative Virtual Address

VM

Virtual Machine

VT

Virus Total

Chapter 1

Introduction

In the digital age, the proliferation of interconnected systems and the adoption of technology have revolutionized the way we interact with information. However, this digital revolution has also ushered in new challenges, chief among them being the omnipresent threat of malicious software, commonly referred to as malware. Malware encompasses a broad spectrum of software designed with malicious intent, ranging from relatively benign nuisances to sophisticated cyber weapons capable of causing widespread disruption and damage. Among the various malware classes, file infectors, also known as viruses, stand out as a particularly insidious class, capable of injecting malicious code within legitimate target files. The history of computer file infectors dates back to the early days of computing, with seminal examples like the Michelangelo virus [1] and the CIH virus [2] demonstrating the disruptive potential of malicious software. Over the years, the landscape of file infectors has evolved, adapting to advancements in technology and security measures. File infectors have been defined in various ways across different contexts and periods. The fundamental characteristic of a file infector is its capacity to alter existing files within a system, distinct from the actions of creating new files or deleting existing ones. File infectors operate by injecting malicious code into pre-existing benign files. Among the infected files, executable files are particularly targeted because they allow to run the malicious code. When executed, infected executables can propagate the file infector by infecting additional files. For this reason our work will be focused on executables.

Recent research conducted by Dambra et al. [3] underscores the inherent challenge in accurately classifying certain classes of malware, particularly file infectors. This challenge arises from the natural behavior of file infectors, which involves injecting snippets of malicious code into benign files. As a result, this mixture of benign and malicious code complicates detection and classification efforts. Given their established presence spanning three decades, we are also particularly interested in assessing the ongoing significance of file infectors in contemporary contexts and the extent to which they are still being utilized. In standard literature [4], file infectors are often classified based on various features, with infection mode being one of them. File infectors exhibit different modes of infection documented in existing literature. These include *Appenders*, which inject their malicious code at the end of the executable; *Prependers*, which inject code at the beginning of the executable; *Cavity Infectors*, which inject code interstitially within the executable; and *Companion Viruses*, which inject malicious code into a newly crafted file somehow linked to the target code, to execute beforehand. *Macro viruses* instead infect document files (i.e., Word, Excel), thus this research will not delve into their consideration.

The goal of this thesis is to design and implement a framework for the analysis of file infection malware. This framework aims to not only identify such malware but also to categorize them into distinct types based on their infection behavior.

Our research aims to distinguish file infectors from other malware classes through a meticulous analysis of file system modifications post-execution of a given sample. Dynamic analysis is performed within a sandbox environment, where the presence of permanently modified executables is determined for each sample. If a sample lacks permanently modified executables, it serves as a key indication that it is not a file infector. The executable files displaying permanent modifications are analyzed by a PE Differ module, which can parse Portable Executable (PE) components (i.e., headers, sections), extracts their features and generates a clear textual representations of the alterations. Additionally, the module computes measures of similarity between the original and modified files, providing an overview of each sample's behavior. Finally, the obtained similarity scores are fed into a Classifier module, which utilizes this data to categorize the analyzed sample into different types (e.g., Appender, Prepender).

For our dataset, we utilized a subset of the malware dataset featured in Dambra's research [3], sourced from VirusTotal's (VT) feed. Specifically, our dataset consists solely of PE malware executables, encompassing 70 distinct malware families. This dataset is carefully balanced, with each family containing a consistent number of 100 samples, resulting in a total of 7,000 samples. Of those, we analyzed 5 randomly selected samples from each family, resulting in a total of 350 samples analyzed.

With this dataset, we derived diverse outcomes. Initially, following sandbox dynamic analysis and employing analysis parameters congruent with an established study [3], we determined that great majority of the families detonated, indicating the manifestation of observable behaviors. Among these detonated samples, we successfully identified 94 samples of 22 families as file infectors, distinguished by the presence of permanently modified executables. We conducted a manual analysis of the samples to identify their respective file infector types. Our analysis specifically identified the presence of 4 families classified as Appenders, 16 families classified as Prependers, and 2 families classified as Impersonators. Subsequently, we utilized a custom-built PE executable differ to extract similarity results, tailoring our approach to the behavioral patterns exhibited by file infectors. By delineating various dimensions for feature extraction, we prepared the dataset to be fed into the classification module. Employing a machine learning multiclass classifier based on the Random Forests model, we proceeded to construct our classification model.

To facilitate model evaluation, we adopted a static random split, dividing the dataset into Training and Testing sets. The Training Set, encompassing 70% of the samples, was utilized for model training, while the Testing Set, comprising 30% of the dataset, served for assessing the model's performance. The obtained results demonstrate remarkably high precision, with all scores reaching a perfect 1.0. While these findings may initially seem encouraging, it is imperative to acknowledge the possibility of overfitting, especially considering the relatively limited size of the analyzed dataset.

To summarize, the main contributions of this thesis are:

- A novel framework for the analysis of file infection malware
- A novel PE executable differ that operates at component level
- Evaluation on 350 malware samples

Chapter 2

Related work

2.1 File Infectors

In the realm of cybersecurity and malware analysis, file infectors have long been a focal point of research and study. Several seminal works have contributed significantly to our understanding of file infectors and their impact on computing systems. In this section, we briefly discuss four influential papers on file infectors and highlight how our work differs from them. Cohen's groundbreaking work [5] introduced the concept of computer file infectors and laid the foundation for file infector research. He proposed formal definitions of file infectors and discussed their potential threat to computer systems. Cohen's experiments with file infector creation and propagation provided valuable insights into the behavior of file infectors. Szor's comprehensive book [6] delves into the intricacies of computer file infectors, offering insights into their anatomy, propagation mechanisms, and defense strategies. It provides a detailed analysis of real-world file infectors and their impact on computer systems, along with practical guidance on file infector detection and mitigation techniques. Saltzer and Schroeder's seminal paper [7] discussed various security principles and mechanisms for protecting information in computer systems. While not focused exclusively on file infectors, their work addressed broader issues of computer security, including access control, authentication, and auditing. It provided a framework for understanding security threats and designing secure systems. Skoudis and Zeltser's book [8] provides a comprehensive overview of malware, including file infectors, worms, Trojans, and other malicious software. It covers various aspects of malware analysis, including detection, classification, and mitigation strategies. The book offers practical insights into combating malware threats in modern computing environments.

While these seminal papers have contributed significantly to our understanding of file infectors, our work differs in focus and scope. Unlike traditional file infector research, which often focuses on understanding and mitigating the impact of file infectors on computer systems, or their family classification, our work is centered around building a novel framework for detecting and classifying file infectors based on their infection behavior.

2.2 Fuzzy Hashes

While cryptographic hash functions enable the immediate identification of differences between two files down to the byte level, they do not provide insight into the degree of difference between them. Fuzzy hashing, also known as similarity hashing, is a technique that enables the detection of similarity among files. Depending on the algorithm used, it can indicate either a distance or a similarity score. Fuzzy hashes are quite present and well studied, they were introduced to deal both with email signatures computation and to forensics artifacts correlation. In our work we will use different types of similarity functions for binary comparison, measuring the similarity between PE files.

There are different fuzzy algorithms used in research, *SSDeep*, proposed in 2006 by Kornblum [9], is a Context-Triggered Piecewise Hash, a method that generates a hash by dividing the input into several segments, computing conventional hashes for each segment, and subsequently merging these individual hashes into a unified string. SSDeep represents one of the earliest fuzzy hashes and several limitations for practical applications have been raised [10] since its creation, and other similarity hashes have been created. Oliver et al. proposed *TLSH* [11] as a newer technique specifically designed for binary analysis, its approach is based on Locality Sensitive Hashing, which focus on distribution of n-grams. It calculates the hash by first processing the byte string with a sliding window creating a bucket array, then the quartile points of this array are computed in order to get first the digest header and consequently the digest body, finally the final digest is constructed by concatenation.

Among all these fuzzy algorithms, different researchers have tried to perform large scale experiments on their accuracy and effectiveness, reaching contradictory conclusions. While Upchurch and Zhou [12] suggested TLSH as completely ineffective for binary comparison, Azab et al. [13] considered it one of the best available solutions for this problem. In 2018, Pagani et al. [14] measured how these algorithms perform in different scenarios and provided interpretations about the reasons why results vary widely, stating TLSH as a better alternative with respect to SSDeep. We looked at this work as one of the latest and well documented papers on this topic, and we decided to use both SSDeep and TLSH in order to calculate similarity between PE executables.

2.3 Executable Diffing

While fuzzy hashes can be used for calculating the similarity between digital files, with no need of understanding neither the file-type nor the context associated with it, sometimes there could be the necessity to dig a little bit more about the differences between specific types of files, such as executable ones. In the earlier section, we explored fuzzy hashes, which operate at the byte level by employing their techniques on input data in the form of byte blobs, however there always be a widespread interest for comparing different versions of the same executable. According to Flake [15], while numerous tools exist for comparing different versions of the same source code, the comparison between executables has always posed a greater challenge.

BinDiff [16] is a comparison tool designed for analyzing binary files, it operates by focusing on the abstract structure of an executable rather than the concrete assembly-level instructions in its disassembly. It generates signatures for each function creating initial matches between two sets of signatures for different executables. A match is established if a signature occurs uniquely in both examined subsets of signatures. Subsequently, callgraphs are employed to discover more matches, examining subsets of functions called from matched functions and increasing the likelihood of finding new unique matches. This iterative process continues until no further matches can be identified. With BinDiff is thus possible to obtain a list of associated functions, identifying structural similarities and differences in executables.

In our study, we opted for the use of similarity hashes due to the inherent difficulties associated with performing disassembly operations on malware. Malware often employs packers or intricate code obfuscation techniques, making it challenging to analyze at the source code level. By operating at the byte level, similarity hashes provided us with a means to overcome these complexities, although this approach sacrificed clarity regarding the source code.

2.4 Malware Detection and Classification

Malware detection and classification are crucial tasks in cybersecurity, and numerous studies have contributed to advancing the field. The task of malware detection involves the development and implementation of techniques and tools to recognize and mitigate malicious software or code that can compromise the security of computer systems. Family classification in the realm of cybersecurity refers to grouping similar or related malware instances based on their characteristics, code similarities, and behavior patterns. Establishing these connections is crucial for understanding the broader landscape of cyberthreats.

Various researchers have delved into the realm of malware detection and classification, offering diverse perspectives and methodologies. Among them, Kalash et al. [17] explored the application of deep convolutional neural networks to classify malware. Additionally, Ye at al. [18] conducted an in-depth survey encompassing a range of data mining techniques employed in malware detection. Furthermore, Aslan and Samet [19] undertook a comprehensive review, critically analyzing and summarizing existing approaches in malware detection. In a similar vein, Pascanu et al. [20] investigated the efficacy of recurrent networks in classifying malware. Recently Dambra et al. [3] comprehensively explored the intricacies of machine learning in malware classification. They investigated the impact of dataset characteristics, feature extraction techniques, and model performance evaluation metrics on classification accuracy, providing valuable insights for researchers.

Our work differs from these studies as we focus specifically on detecting and classifying file infectors. Unlike traditional malware detection approaches that classify malware into families, we propose a novel framework for the analysis of file infectors based on their infecting behavior. This approach enables us to develop more targeted and effective detection and classification techniques tailored to the unique characteristics of file infectors.

Chapter 3

Dataset

For our study, we utilize a subset of the malware dataset featured in the paper by Dambra et al. [3]. The source dataset comprises PE malware executables from VirusTotal (VT) [21] feed, a real-time stream of JSON-encoded reports generated by the execution of those samples in VT's virtual machines. VT's report can contains not only file hashes, strings extracted from the executables, the open sockets and different kind of metadatas, but also detection labels assigned by different antivirus (AV) engines. In their paper, Dambra et al. collected 118,111 malicious PE executables divided into different datasets referenced as Malware Balanced (M_B) composed by 670 families with 67,000 samples, Benign (B) composed by 16,611 samples, Malware Unbalanced (M_U) composed by 1,500 families with 18,000 samples and Malware Generic (M_G) composed by 16,500 samples. According to their paper they collected reports and samples from the VT feed for 82 non-consecutive days between August 2021 and March 2022, maintaining only those samples detected by at least one AV engine and with a file type equal to "32-bit non-installer PE executable". Out of all the collected executables, 64-bit PE executables, dynamic-link libraries (DLLs), and executables generated by installer software have been excluded. For our purposes we focused only on the first dataset (M_B), as it is balanced and with a constant number of samples of 100, enough to performing multi-class classification experiments.

After identifying the dataset source, our attention turned to filtering the retained reports to extract a subset consisting solely of file infectors. This process involved utilizing *AVclass2* [22]. AVclass2 is an automatic malware tagging tool that given the AV labels for a large number of samples, extracts for each input sample a clean set of tags that capture properties such as the malware class, family, file properties, and behaviors. AVclass2 provides different tags under each of the four default categories: behavior, class, file properties, and family, and it is able to identify strong relations between them. Thanks to the *Expansion phase* [22], AVclass2 outputs an *alias file* containing a list of tags and unknown tokens along with their confidence scores. The alias file is based on tags belonging to two different categories, and it is structured in different values: the tags, the number of samples each tag appears in the dataset being labeled, the number of samples where both tags appear, and the fraction of times that both tags appear in the same samples. From the output file, we selected the two categories as *family* and *class*. For the first set, called *"Likely viruses"*, we included all samples with the file infector class assigned as the top class. Additionally, we created another set named *"Maybe viruses"*, which comprised samples where the file infector class was not assigned as the top class, but where the associated confidence score for the file infector class within a family exceeded a specified threshold denoted as T_1. In our case, T_1 was set to 0.5, capturing all families with a probability greater than 50% of being classified as file infectors.

Dataset	**Samples**	**Families**
Likely viruses	4,000	40
Maybe viruses	3,000	30
All	7,000	70

Table 3.1: Summary of dataset.

In summary, as outlined in Table 3.1, we derived 70 families, each comprising 100 distinct samples, divided into two sets: "Likely viruses", encompassing all samples where the file infector class was assigned as the top class, and "Maybe viruses", comprising samples where the file infector class was not the top class, but had a probability greater than 50% of being classified as such.

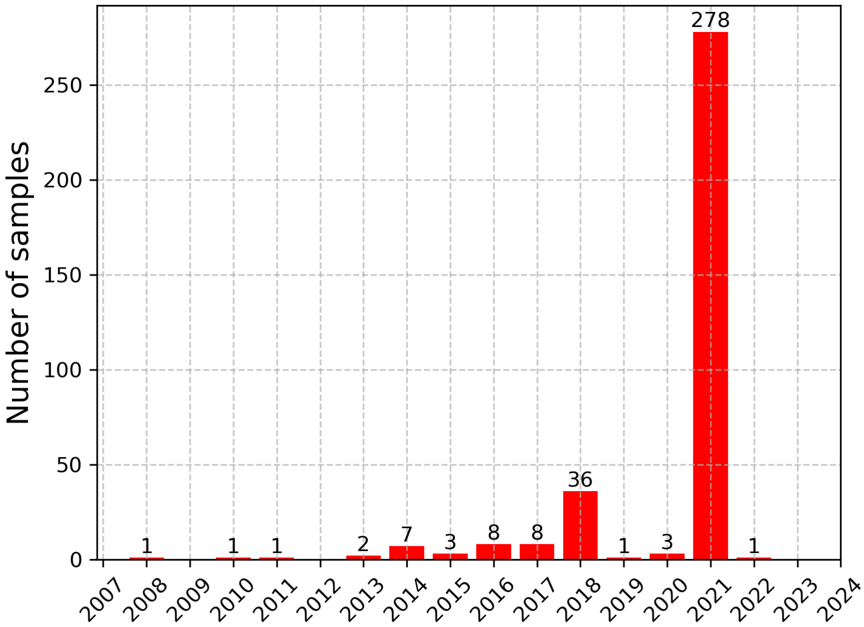


Figure 3.1: Number of samples first seen by VirusTotal per year.

To contextualize the timeline of our dataset samples, we chose to extract the *"first_seen"* field from previously collected VirusTotal reports. This helped us determine the year of first submission for these samples. It is important to note that due to resource constraints, we could only analyze a subset of 350 samples out of the total 7,000 in the dataset, as detailed in Chapter 5. Therefore, our temporal analysis will focus solely on these analyzed samples. As detailed in Figure 3.1, all samples are dated between 2008 and 2022, with a noticeable peak in 2021. Given that the VirusTotal feed was collected over a year of observation spanning from 2021 to 2022, these samples are relatively recent. This observation also suggests that file infectors are still actively infecting systems.

Chapter 4

Methodology

The goal of our study is to introduce a framework for the analysis of file infection malware. This framework aims to not only identify such malware but also to categorize them into distinct types based on their infection behavior. The framework, outlined in Figure 4.1, begins by taking a Portable Executable (PE) file as input. This file is dynamically executed in Cuckoo Sandbox, which generates a Cuckoo report and a disk image containing the altered file system resulting from the execution of the sample. These elements, along with the unaltered disk image prior to execution (the second input), are fed into the Image Differ module. This module compares the two file systems, identifying all modified executables. The presence of modified executables suggests potential infection, prompting further analysis. The modified executables, along with their original versions, serve as input for the PE Differ module. This module produces textual files for manual inspection of modifications and derives various features from similarity scores between executables. These features are then used by the Classifier to classify the executables into different types (e.g., Appender, Prepender). In the following sections all the conceptual steps for the analysis are explained in details.

4.1 Sandbox

Our analysis aims to identify file infectors by examining the differences between the permanently modified executables in the file system post-sample execution and their original counterparts. Dynamic analysis was employed to achieve this objective. We set up Cuckoo Sandbox for executing malware using the best practices recommended by previous works [23, 24, 25]. We configured a Windows 7 Pro 32-bit virtual machine (VM) with 2 CPUs and 2 GB of RAM. We deployed widely used applications and populated the file system with standard file formats to simulate the configuration of an authentic desktop workstation, and as recommended by Rossow et al. [25], the virtual machine runs

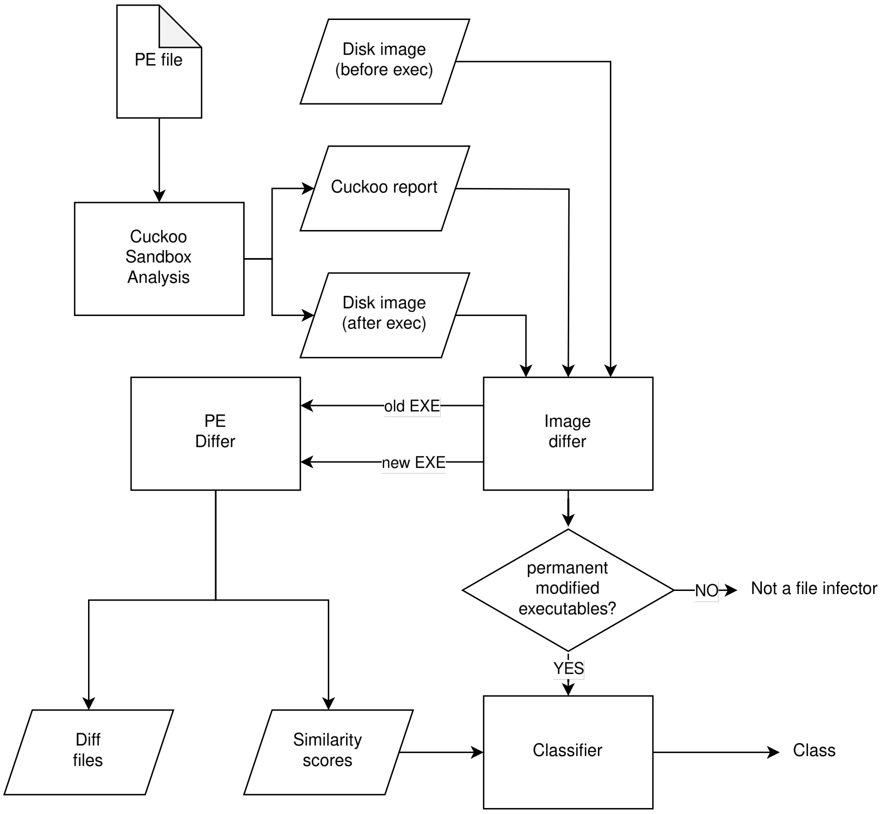


Figure 4.1: Framework for the analysis of file infection malware.

on its isolated local network. Ultimately, we validated our analytical environment using Al-Khaser [26] and Pafish [27] tools, to ascertain that our sandbox remained undetectable and potentially did not prompt evasive techniques from malware. We employed Cuckoo Sandbox, also referred to as Cuckoo, as our malware analysis system, which utilizes customizable components to monitor and document the activities of malicious processes while they operate within a virtual machine environment, which must be provided. Cuckoo works primarily on two essential concepts: the host, which is the operating system where Cuckoo is installed, housing configuration files and scripts for analysis, and the guest, an operating system utilized as a virtual machine by Cuckoo. The guest serves as the environment for executing malware and is restored after each iteration. During an analysis, control shifts from the host to the guest to execute the sample. Afterward, control is returned to the host, where all auxiliary and reporting modules are utilized to generate a textual report in JSON format. While Cuckoo Sandbox is a valuable malware analysis system, it is important to note that there are alternatives like Intel Pin [28] and Joe Sandbox [29], which are widely recognized and utilized by the scientific community. The decision to opt for Cuckoo is driven by two key factors: its user-friendly customization of auxiliary and reporting scripts, and the completeness of the JSON report it generates. To decide the execution time for each sample we relied on a previous study [3] and given the mentioned work, we adopted a similar cautious approach and executed each sample for a maximum of five minutes.

The input given to the Cuckoo Sandbox is an executable file, while its standard output is a JSON file. To facilitate the extraction of the file system image post-sample execution, we developed custom Python scripts. The disk's initial state was transformed into a disk image to serve as a reference for the original file system. Practically, immediately following the execution of the PE file and the transfer of control to the guest, the system captures a snapshot of the current disk state. Upon returning control to the host, this snapshot is restored, and a disk image is extracted in a raw format. This systematic process ensures that all files modified during execution are stored in an image file, facilitating easy access to them.

4.2 Image Differ

The Image Differ module takes disk images before and after sample execution, along with the Cuckoo report, as input. Its objective is to identify all permanently modified executables. To achieve this, the module calculates the SHA-256 hash of all files and identifies the subset with different hashes but the same filepath in both images, thanks to the information provided by the Cuckoo report. Following this, it filters for executable files and proceeds to extract them. To traverse the entire file system and extract executable files, we utilized a Python module called *The Sleuth Kit* [30], an open-source forensic toolkit for analyzing Microsoft and Unix file systems and disks. We exclusively concentrated on analyzing executables because they are a prime vector for triggering file infection and their ease of parsing provides convenience in understanding the modifications introduced by file infectors. The extracted executable files, both modified and original versions, along with their filepaths, are stored in a Python dictionary. This dictionary is serialized using the Python module *pickle* [31] to facilitate its use between Python scripts.

In Listing 4.1, a subset of executables extracted from a sample is displayed, each denoted by its filepath along with its old and new hashes resulting from modifications.

Listing 4.1: Example of extracted executables from a sample.

```
//Windows/assembly/NativeImages_v2.0.50727_64/MSBuild/1
    a154709cdfe214029ea88c51ab2b579
    321b2b1d180b4eb0bb5402fe6417f7f892a9dd68089419274f35dd555820cd35
    065b8ef72d98dd901b0199cf1a007c7696ee29279245f67393db4ea7f01f414f

//$Recycle.Bin/S-1-5-21-483214431-1722755210-2890981749-1001/$R7W2UHH/lib64/
    python3.10/site-packages/setuptools
    5c1af46c7300e87a73dacf6cf41ce397e3f05df6bd9c7e227b4ac59f85769160
    256ceba9c8e9e2303748398d2e08be9257756cbdcc7160924e4a787328334d58

```

```
//$Recycle.Bin/S-1-5-21-483214431-1722755210-2890981749-1001/$R7W2UHH/lib64/
    python3.10/site-packages/setuptools
    28b001bb9a72ae7a24242bfab248d767a1ac5dec981c672a3944f7a072375e9a
    639bb16efaf06be54434215a9eaadcea35aa3fb600a8359211bc3a5b05e3aa35

//Python27/Lib/site-packages/pip/_vendor/distlib
    34f60fa6decf22356a00112ed42cda6db0f21c7909a6ec3efea66aff8f07d23d
    b8e087f1b2166047bdb40c7335ddde85fa13e7118ad9831221edb9f309446c5a

//Program Files/LibreOffice/program
    a9e7d53b51e332a9a182e1cbb801ba243e98535aaf99991a53a4925865fdee1d
    12d70059c2d49067ca281a60542ea9cca660c0025825795a9d1dfe2822136e58

//$Recycle.Bin/S-1-5-21-483214431-1722755210-2890981749-1001/$R7W2UHH/lib64/
    python3.10/site-packages/setuptools
    75f12ea2f30d9c0d872dade345f30f562e6d93847b6a509ba53beec6d0b2c346
    d004eef460c27ec91580732e1b216dffcb43febb685ca60e8c8f215fc55ae32e

//Program Files/LibreOffice/program/python-core-3.8.18/lib/distutils/command
    3c978f7167f71538635c35864f2ac7862cb9ba8c57464b6f90f0c6185f258cb9
    fd0755f5170f9509a116f4097e0a7b923ce3deef4791d64d2d0bdf2684a6b534

//Program Files/LibreOffice/program/python-core-3.8.18/lib/distutils/command
    cf6cd3e0b085e89584061f5562f6206a98673251042e486ba0210d7d46817081
    6383564aca068de256d4130c26e17ab49c2e60314b3cd5a84a14e701f6f22cdf

//Program Files (x86)/Mozilla Maintenance Service
    1341040bb86be331284ddb95c93d4b157267c2c145d248c729eea6081e03fd19
    d580c1b66c4516cb53a3af521d3513d2439bdcdc9c9e7e92e8f79fd5f805b025

//Program Files (x86)/Adobe/Acrobat Reader DC/Reader
    1a5ea90eff67873ecba529096c005b641e025a8dd92a1732c532c500516de324
    4ada2b540ca9957917b6b6f93b18cf34531e4aec255442faf7a4f679df6a9c31
```

4.3 PE Structure

A Windows PE file follows a specific and fixed structure, illustrated in Figure 4.2, consisting of various components. Each component is further subdivided into fields that describe its specific attributes. Some components consist entirely of raw byte data, lacking distinct fields. In Figure 4.2, intentional empty spaces between components provide areas where malware could inject content without affecting functionality as well as components marked in red. The upcoming subsection will provide detailed explanations for each component.

In order to have a clear view of each component and its content, we implemented a Python script, based on the Python module *pefile* [32]. The software processes a 32-bit

Portable Executable (PE32) file by parsing each of its components. It extracts and presents details such as file offset, size, field name, and corresponding values for each field inside components. For components without fields, the program calculates the SHA-256 hash of the associated byte data. Additionally, the software identifies any bytes not attributed to specific components, dealing potential gaps in the analysis by computing the SHA-256 hash for these stray bytes. We intentionally omitted certain aspects of components and fields, such as header flags, as we did not consider them crucial for our work. This decision was made with the goal of creating a comprehensive mapping of every byte within the PE files. This approach facilitates the identification of the position of any modifications at the byte level.

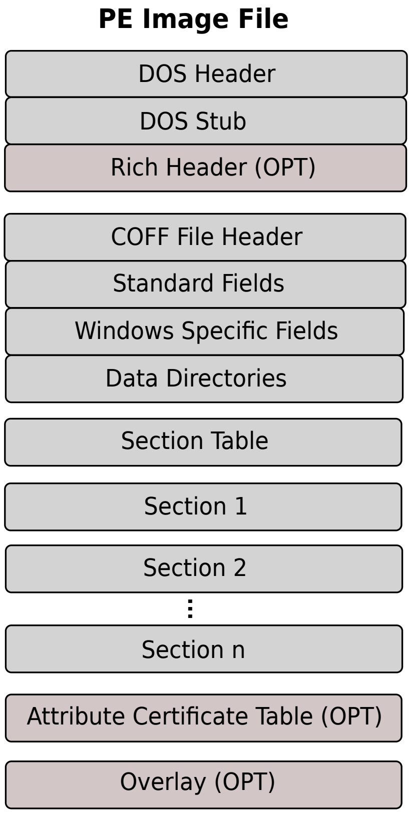


Figure 4.2: Portable Executable structure.

4.3.1 DOS Header

The DOS header, also known as the MS-DOS header, is a 64-byte structure located at the beginning of the PE file. It does not play a crucial role in the functionality of PE files on modern Windows systems and it exists for backward compatibility reasons. This header designates the file as an MS-DOS executable, therefore, when loaded on MS-DOS, the DOS stub is executed instead of the actual program. Omitting this header would result in the executable not loading on MS-DOS and producing a generic error. Two noteworthy fields are the following: *"e_magic"*, the initial member of the DOS Header, is a 2-byte-long entity commonly known as the "magic number". With a fixed value of *0x5A4D* or *MZ* in ASCII, it serves as a signature designating the file as an MS-DOS executable. The second field, *"e_lfanew"*, ends the DOS header structure, holding an offset to the start of the NT headers.

4.3.2 DOS Stub

The MS-DOS stub is a valid application that runs under MS-DOS. It is placed at the front of the PE image. The linker places a default stub here, which prints out the message "This program cannot be run in DOS mode" when the image is run in MS-DOS.

4.3.3 Rich Header

The Rich Header is an optional, undocumented field in the PE32 file format located between the MS-DOS and COFF Headers. It consists of a chunk of encrypted data followed by a signature *"Rich"*, from which it takes its name, and a 32-bit checksum value that is the encryption key. The encryption used in the Rich Header is a simple XOR function, thus, it is easy for attackers to take advantage of this implementation, but not equally smooth to detect for automated analysis. For this and other reasons, the Rich Header's potential in enabling the rapid triage of malicious samples has been demonstrated in Webster's work [33].

4.3.4 COFF File Header

At the beginning of an object file, or immediately after the signature of an image file, there is the COFF file header made of seven fixed fields. Notably, the field *"NumberOfSections"* is the only one worth mentioning, as it indicates, the number of sections present in the PE file.

4.3.5 Optional headers

The Optional headers starts right after the COFF File Header and it has three major parts. The first eight fields of the Optional headers, known as standard fields, are defined for every implementation of COFF. These fields contain general information that is useful for loading and running an executable file.

The subsequent 21 fields, categorized as Windows-specific fields, hold additional information necessary for the linker and loader in Windows. Among these, two fields are particularly noteworthy for our research: *"SizeOfCode"*, representing the sum of sizes of all text sections, and *"AddressOfEntryPoint"*, describing the starting address for execution.

The final segment of the Optional headers is the data directories. It is formed by different data directory entries are all loaded into memory so that the system can use them at run time. A data directory is an 8-byte field, consisting of 4-bytes Relative Virtual Address (RVA) of the table and 4-bytes of table size. The number of data directories is not fixed within a PE file and is determined by the *"NumberOfRvaAndSizes"* field in the Optional headers.

4.3.6 Sections

Section Table The Section Table contains details about the properties of PE data sections. Each section header is 40 bytes in size and includes information such as the section's name, size, virtual and physical addresses. This table is crucial in a PE executable as it describes the metadata of sections, rendering it susceptible to alteration during infection.

Data Sections Data sections in the PE executable format hold various types of data within the binary. These sections store information such as executable code, initialized data, uninitialized data, resources, and import/export tables. Each data section has its own characteristics, including a name, size, virtual and physical addresses, and flags indicating properties such as whether the section is executable, writable, or readable.

4.3.7 Attribute Certificate Table

The Certificate Table in PE files stores digital signatures that verify the integrity and authenticity of the executable. It holds certificates signed by trusted entities, enhancing the security and trustworthiness of the executable. The entry in the Certificate Table directs to a set of attribute certificates, distinctively not loaded into memory with the image. Due to its importance, malicious actors may leverage it to evade automated detection, a commonly observed tactic in malware, as outlined in a study by Kotzias et al. [34]. Additionally, in the same paper, researchers highlight a concern with Authenticode, located in the Certificate Table, where timestamped signed malware successfully validates even after the revocation of their code signing certificate.

4.3.8 Overlay

In a PE file, an overlay refers to additional data appended to the end of the legitimate PE file structure, this data is optional and not officially part of the PE file specifications. While the legitimate components of the PE file have specific offsets and sizes defined in the file header, the overlay is any data beyond the specified size of the last component. It is essential to note that although the Certificate Table, a component used to verify the authenticity and integrity of executable code, is technically part of the overlay as it is positioned after the last section, but we have excluded it from the overlay for the purpose of our work. Overlays can be utilized to conceal malicious code or attempt to evade detection.

4.4 PE Differ

The module responsible for identifying differences between executables is the PE Differ module. It accepts a pair of PE executable files as input: the original file and its infected version. It parses these files, breaking them down into PE components as detailed in the preceding section. The module then performs component-level diffing for each component, extracting the differences. The output produced by the module is a *diff file*, which explicitly highlights the alterations between the two executable files in a human-readable format. Unlike other tools such as BinDiff, which are capable of establishing differences between files through disassembly procedures, our tool does not involve disassembly processes. The module operates analogously to the Unix utility *diff*, part of *diffutils* [35] package. The Listing 4.2 provides a diff file example.

Listing 4.2: Example of diff file.

```
  file_path        //Python27/Lib/site-packages/pip/_vendor/distlib/w32.exe
- file_sha-256     34
    f60fa6decf22356a00112ed42cda6db0f21c7909a6ec3efea66aff8f07d23d
+ file_sha-256
    c4b5bc7fcd8b0e89e3bcc2dc3e6bea2e34fd393c86b268b9565fe20ef2407524
- file_tlsh
    T1E9938D11B291D076D05624305C6AC2B10ABEFC3395B9C54B7BC97B3E1F71381AA6BB27
?                     ^    ^^             ^                  -           ^
     ^ ^^   ^

+ file_tlsh
    T1ED938D55B291D076D05625305C6AC2B10ABEFC3354B9C54B7BC93B3E1F723C0AA6AB27
?                     ^    ^^             ^                   +          ^
     ^ ^^   ^

- file_ssdeep      1536:8Hg7DHaVCkO8XyUKdujCc66m5tvKbUOQNieF2FfHYTolz:8
    A7AFfCGReFwfHYTol
?                       ^^   ^                            ^^          ^^ ^
    ^^
```

```
+ file_ssdeep      1536:zxg7DkRaVCkO8XyUKdujCc66m5tvKbUOQNiml2FfHYTolzs:
    zO7ouFfCGRmlwfHYTolQ
?                       ^^   ^^                            ^^         + ^^ ^^
       ^^       +
- actual_size      0x15c00
?                     ^
+ actual_size      0x16c00
?                     ^
- expected_size    0x15c00
?                     ^
+ expected_size    0x16c00
?                     ^
  difference       0x0

  DOS_HEADER
  0x0        0x2        e_magic                        0x5a4d
  0x2        0x2        e_cblp                         0x0090
  0x4        0x2        e_cp                           0x0003
  0x6        0x2        e_crlc                         0x0000
  0x8        0x2        e_cparhdr                      0x0004
  0xa        0x2        e_minalloc                     0x0000
  0xc        0x2        e_maxalloc                     0xffff
  0xe        0x2        e_ss                           0x0000
  0x10       0x2        e_sp                           0x00b8
  0x12       0x2        e_csum                         0x0000
  0x14       0x2        e_ip                           0x0000
  0x16       0x2        e_cs                           0x0000
  0x18       0x2        e_lfarlc                       0x0040
  0x1a       0x2        e_ovno                         0x0000
  0x1c       0x8        e_res
  0x24       0x2        e_oemid                        0x0000
  0x26       0x2        e_oeminfo                      0x0000
  0x28       0x14       e_res2
  0x3c       0x4        e_lfanew                       0x000000f8

  DOS_STUB
  0x40       0x40       dos_stub                       sha256:7764
    e7022dcac1b5779d1f96fc05af5c1fee394aaff8a3a7e9a881e1a1b163a3
  0x80       0x60       rich_header                    sha256:0432
    e5a8f2acd97d3ec605b8e0bad8891b35f7fff83b952762100f67c2aa4a09

  0xe0       0x18       unknown                        sha256:9
    d908ecfb6b256def8b49a7c504e6c889c4b0e41fe6ce3e01863dd7b61a20aa0

  NT_HEADERS
  0xf8       0x4        Signature                      0x00004550

  FILE_HEADER
  0xfc       0x2        Machine                        0x014c
  0xfe       0x2        NumberOfSections               0x0005
- 0x100      0x4        TimeDateStamp                  0x5ad46bbb [Mon Apr 16
    09:24:11 2018 UTC]
?                                                        ^^ -----  ^^^  ^^ ^^
     ^ ^^  ^    ^
+ 0x100      0x4        TimeDateStamp                  0x4e4f4f4d [Sat Aug 20
    06:08:13 2011 UTC]
?                                                        ^^^^^^^   ^^^  ^^ ^^
     ^ ^^  ^    ^
  0x104      0x4        PointerToSymbolTable           0x00000000
  0x108      0x4        NumberOfSymbols                0x00000000
  0x10c      0x2        SizeOfOptionalHeader           0x00e0
  0x10e      0x2        Characteristics                0x0102

  OPTIONAL_HEADER
  0x110      0x2        Magic                          0x010b
  0x112      0x1        MajorLinkerVersion             0x0a
  0x113      0x1        MinorLinkerVersion             0x00
  0x114      0x4        SizeOfCode                     0x0000ba00
  0x118      0x4        SizeOfInitializedData          0x00009e00
  0x11c      0x4        SizeOfUninitializedData        0x00000000
  0x120      0x4        AddressOfEntryPoint            0x00002e1a
  0x124      0x4        BaseOfCode                     0x00001000
  0x128      0x4        BaseOfData                     0x0000d000
  0x12c      0x4        ImageBase                      0x00400000
  0x130      0x4        SectionAlignment               0x00001000
  0x134      0x4        FileAlignment                  0x00000200
  0x138      0x2        MajorOperatingSystemVersion    0x0005
  0x13a      0x2        MinorOperatingSystemVersion    0x0001
  0x13c      0x2        MajorImageVersion              0x0000
  0x13e      0x2        MinorImageVersion              0x0000
  0x140      0x2        MajorSubsystemVersion          0x0005
  0x142      0x2        MinorSubsystemVersion          0x0001
  0x144      0x4        Reserved1                      0x00000000
- 0x148      0x4        SizeOfImage                    0x0001b000
?                                                            ^
+ 0x148      0x4        SizeOfImage                    0x0001c000
?                                                            ^
  0x14c      0x4        SizeOfHeaders                  0x00000400
  0x150      0x4        CheckSum                       0x0001b9f7
  0x154      0x2        Subsystem                      0x0002
  0x156      0x2        DllCharacteristics             0x8140
  0x158      0x4        SizeOfStackReserve             0x00100000
  0x15c      0x4        SizeOfStackCommit              0x00001000
  0x160      0x4        SizeOfHeapReserve              0x00100000
  0x164      0x4        SizeOfHeapCommit               0x00001000
  0x168      0x4        LoaderFlags                    0x00000000
  0x16c      0x4        NumberOfRvaAndSizes            0x00000010

  DIRECTORIES_HEADER
  ENTRY_EXPORT
  0x170      0x4        rva                            0x00000000
  0x174      0x4        size                           0x00000000
  ENTRY_IMPORT
  0x178      0x4        rva                            0x0000f36c
  0x17c      0x4        size                           0x0000003c
  ENTRY_RESOURCE
  0x180      0x4        rva                            0x00014000
  0x184      0x4        size                           0x000050a4
  ENTRY_EXCEPTION
  0x188      0x4        rva                            0x00000000
  0x18c      0x4        size                           0x00000000
  ENTRY_SECURITY
  0x190      0x4        rva                            0x00000000
  0x194      0x4        size                           0x00000000
  ENTRY_BASERELOC
- 0x198      0x4        rva                            0x0001a000
?                                                           ^^
+ 0x198      0x4        rva                            0x00000000
?                                                           ^^
- 0x19c      0x4        size                           0x0000090c
?                                                             - ^
+ 0x19c      0x4        size                           0x00000000
?                                                              ^^
  ENTRY_DEBUG
  0x1a0      0x4        rva                            0x0000d190
  0x1a4      0x4        size                           0x0000001c
  ENTRY_COPYRIGHT
  0x1a8      0x4        rva                            0x00000000
  0x1ac      0x4        size                           0x00000000
  ENTRY_GLOBALPTR
  0x1b0      0x4        rva                            0x00000000
  0x1b4      0x4        size                           0x00000000
  ENTRY_TLS
  0x1b8      0x4        rva                            0x00000000
  0x1bc      0x4        size                           0x00000000
  ENTRY_LOAD_CONFIG
  0x1c0      0x4        rva                            0x0000eed0
  0x1c4      0x4        size                           0x00000040
  ENTRY_BOUND_IMPORT
  0x1c8      0x4        rva                            0x00000000
  0x1cc      0x4        size                           0x00000000
  ENTRY_IAT
  0x1d0      0x4        rva                            0x0000d000
  0x1d4      0x4        size                           0x00000154
  ENTRY_DELAY_IMPORT
  0x1d8      0x4        rva                            0x00000000
  0x1dc      0x4        size                           0x00000000
```

```
ENTRY_COM_DESCRIPTOR
0x1e0      0x4        rva                            0x00000000
0x1e4      0x4        size                           0x00000000
ENTRY_RESERVED
0x1e8      0x4        rva                            0x00000000
0x1ec      0x4        size                           0x00000000

SECTION_HEADER
0x1f0      0x8        Name                           .text
0x1f8      0x4        Misc                           0x0000b8da
0x1f8      0x4        Misc_PhysicalAddress           0x0000b8da
0x1f8      0x4        Misc_VirtualSize               0x0000b8da
0x1fc      0x4        VirtualAddress                 0x00001000
0x200      0x4        SizeOfRawData                  0x0000ba00
0x204      0x4        PointerToRawData               0x00000400
0x208      0x4        PointerToRelocations           0x00000000
0x20c      0x4        PointerToLinenumbers           0x00000000
0x210      0x2        NumberOfRelocations            0x0000
0x212      0x2        NumberOfLinenumbers            0x0000
0x214      0x4        Characteristics                0x60000020

SECTION_HEADER
0x218      0x8        Name                           .rdata
0x220      0x4        Misc                           0x00002b2c
0x220      0x4        Misc_PhysicalAddress           0x00002b2c
0x220      0x4        Misc_VirtualSize               0x00002b2c
0x224      0x4        VirtualAddress                 0x0000d000
0x228      0x4        SizeOfRawData                  0x00002c00
0x22c      0x4        PointerToRawData               0x0000be00
0x230      0x4        PointerToRelocations           0x00000000
0x234      0x4        PointerToLinenumbers           0x00000000
0x238      0x2        NumberOfRelocations            0x0000
0x23a      0x2        NumberOfLinenumbers            0x0000
0x23c      0x4        Characteristics                0x40000040

SECTION_HEADER
0x240      0x8        Name                           .data
0x248      0x4        Misc                           0x000036e4
0x248      0x4        Misc_PhysicalAddress           0x000036e4
0x248      0x4        Misc_VirtualSize               0x000036e4
0x24c      0x4        VirtualAddress                 0x00010000
0x250      0x4        SizeOfRawData                  0x00001000
0x254      0x4        PointerToRawData               0x0000ea00
0x258      0x4        PointerToRelocations           0x00000000
0x25c      0x4        PointerToLinenumbers           0x00000000
0x260      0x2        NumberOfRelocations            0x0000
0x262      0x2        NumberOfLinenumbers            0x0000
0x264      0x4        Characteristics                0xc0000040
```

```
  SECTION_HEADER
  0x268      0x8        Name                           .rsrc
  0x270      0x4        Misc                           0x000050a4
  0x270      0x4        Misc_PhysicalAddress           0x000050a4
  0x270      0x4        Misc_VirtualSize               0x000050a4
  0x274      0x4        VirtualAddress                 0x00014000
  0x278      0x4        SizeOfRawData                  0x00005200
  0x27c      0x4        PointerToRawData               0x0000fa00
  0x280      0x4        PointerToRelocations           0x00000000
  0x284      0x4        PointerToLinenumbers           0x00000000
  0x288      0x2        NumberOfRelocations            0x0000
  0x28a      0x2        NumberOfLinenumbers            0x0000
  0x28c      0x4        Characteristics                0x40000040

  SECTION_HEADER
  0x290      0x8        Name                           .reloc
- 0x298      0x4        Misc                           0x00000e8e
?                                                             ^^^
+ 0x298      0x4        Misc                           0x00002000
?                                                            + ^^
- 0x298      0x4        Misc_PhysicalAddress           0x00000e8e
?                                                             ^^^
+ 0x298      0x4        Misc_PhysicalAddress           0x00002000
?                                                            + ^^
- 0x298      0x4        Misc_VirtualSize               0x00000e8e
?                                                             ^^^
+ 0x298      0x4        Misc_VirtualSize               0x00002000
?                                                            + ^^
  0x29c      0x4        VirtualAddress                 0x0001a000
- 0x2a0      0x4        SizeOfRawData                  0x00001000
?                                                            ^
+ 0x2a0      0x4        SizeOfRawData                  0x00002000
?                                                            ^
  0x2a4      0x4        PointerToRawData               0x00014c00
  0x2a8      0x4        PointerToRelocations           0x00000000
  0x2ac      0x4        PointerToLinenumbers           0x00000000
  0x2b0      0x2        NumberOfRelocations            0x0000
  0x2b2      0x2        NumberOfLinenumbers            0x0000
  0x2b4      0x4        Characteristics                0x42000040

  0x2b8      0x148      unknown                        sha256:7b4499c3cc6e82a9da3100028f52af7f8c1e9ee60e33010a108e401989782962

  SECTION_DATA
- 0x400      0xba00     .text                          sha256:f63208c03a4e74b371ce17a5151f14acb1dff91c68bbb775bf53904106ede02f
+ 0x400      0xba00     .text                          sha256:bef4d87efc208ad5a86e5fdefef089baff51b919f76fba05049df2853f68ba6b

```

```
  SECTION_DATA
- 0xbe00     0x2c00     .rdata                         sha256:82
   e7bdf9a42713d94b225e1da4a204b3797898e9bebc30fe3b2ff38b3d86385c
+ 0xbe00     0x2c00     .rdata                         sha256:
   c9f7496ff730bd5b9934c9d170a9731cd07f4e34e834da3da560e6320fba8f44

  SECTION_DATA
  0xea00     0x1000     .data                          sha256:1
   f030beb042fa9a2eada0dee901747f50b0b6a006d0dc4d086b23a607d4244f4

  SECTION_DATA
  0xfa00     0x5200     .rsrc                          sha256:
   dde3adb5a734a39fe8fccefcd3f8a315bfa883a6466b1ab7b20cc2670c9acff0

  SECTION_DATA
- 0x14c00    0x1000     .reloc                         sha256:
   d175784bb45286eb24f579f824c682d105e2edc10991848ac1e0e3d52922b961
+ 0x14c00    0x2000     .reloc                         sha256:
   f3f3668a7ca7eb4701e6a303f8fb59b86e0e1eee9243430571c4ad3919669028
```

After establishing the process for highlighting differences between the analyzed input files, the need arose to define a distinct measure for file similarity. To quantify the differences between files and express them numerically, we employed fuzzy hashes. Fuzzy hashes, also known as similarity hashes [36], serve the purpose of establishing a distance metric between files. In this study, we applied fuzzy hashes as byte-wise approximate matching algorithms. This approach involves two phases: the first phase calculates a string that serves as a representation for the specific file. This string captures all aspects of the file by encoding them into a fixed sequence of characters, commonly known as a fingerprint. The second phase of the process is dedicated to generating similarity scores. We employed two distinct fuzzy hashes, SSDeep [9] and TLSH [11]. These hashes were implemented using their respective Python modules, enhancing the efficiency of both phases of the process. The similarity scores indicate the degree of similarity between two files, ranging from 0 (indicating no similarity) to 100 (indicating high similarity). Regarding TLSH, the Python library used allowed only for the computation of distance scores. To convert these distance scores into similarity scores, we relied on the work of Upchurch and Zhou [37], which enabled us to invert and normalize the algorithm. To determine the similarity scores for each pair of files, we considered the maximum similarity scores obtained from both SSDeep and TLSH.

The PE executable differ tool utilized by the PE Differ module has been made available as open-source software, promoting accessibility and collaboration within the research community. The source code and documentation for the tool are available at: `https://github.com/f4ncyz4nz4/pe_differ`.

4.5 Classifier

The final component of the proposed framework is the Classifier. Its primary function is to receive similarity scores from the PE Differ module as input, extract features by combining the similarity scores of permanently modified executables, and subsequently produce a label to classify the sample. For each sample, six similarity-based features are established to feed into the properly trained classifier. The selection of these features is grounded in the characterization of file infectors outlined in Section 5.2. These features encompass six key aspects:

- Average similarity scores between original and modified infected executables, they serve to quantify the degree of similarity between original and infected executables.
- Average similarity scores between original and modified infected files truncated according to the original size, they are particularly useful for samples that append a significant amount of bytes in the last sections, thereby altering normal similarity scores.
- Average similarity scores between the original and just the overlay of the modified infected files, they are particularly useful for samples that store the original file in the overlay, possibly with some modifications.
- Average similarity scores between original and modified infected files with the overlay removed, they are particularly useful for samples that append a significant amount of bytes to the overlay, rendering a normal similarity scores meaningless.
- Average similarity scores between modified infected files, they serve to quantify the degree of similarity between infected executables.
- Number of infected files appended with an overlay, it measures how many infected executables present overlays compared to their original version.

We utilized a Random Forest classifier to handle the classification task, which involved distinguishing among four distinct labels. These labels used by the Classifier are: Appender, Prepender, Impersonator, and *Unknown*. The label “Unknown” serves as a pseudo-label assigned to all file infectors exhibiting unusual behavior, which may include behaviors that are non-malicious or different from typical behaviors observed within the same family. Random Forest is an ensemble learning method widely used for both classification and regression tasks. It operates by constructing a multitude of decision trees during the training phase and outputting the mode of the classes for classification or the mean prediction for regression. Each decision tree in the ensemble is trained on a bootstrapped subset of the training data and makes predictions independently. The final prediction is determined by aggregating the predictions of all trees in the forest. Random Forest is renowned for its robustness against overfitting, high accuracy, and ability to handle large datasets with high dimensionality. By utilizing Random Forest, we aimed to leverage its strengths in handling multi-class classification tasks while ensuring reliable and interpretable predictions.

Chapter 5

Results

The upcoming sections will explain the outcomes derived from various analysis modules of the framework proposed and outlined in Chapter 4.

5.1 Dynamic Analysis

In this section, we delve into the findings from the dynamic analysis conducted within the controlled environment of Cuckoo Sandbox, following the analytical framework outlined in Figure 4.1. We had a dataset consisting of 7,000 samples spread across 70 distinct families, as shown in Table 3.1. From this dataset, we analyzed a total of 350 samples, with 5 samples randomly chosen from each family.

Table 5.1 provides a summary of the relevant information extracted from the analysis of each sample. Each row represents a different family, with the columns indicating the total data aggregated from all samples within that family. The columns, from left to right, detail the family name, the total number of executed samples per family, the count of files opened, written, and deleted, the total number of permanently modified executables (with the number of samples that permanently modified those executables in brackets if greater than zero), and finally, the total count of Windows API calls. We observed that only 2 out of the 70 families, namely *contenedor* and *stone*, did not exhibit any Windows API calls. This finding suggests that behavioral observations were possible for the vast majority of our samples. To further refine our analysis, we referred to a previous study [3], which proposes a threshold of at least 50 Windows API calls to ascertain sample detonation. Detonation refers to the process of triggering a sample to exhibit its malicious behavior. After applying this threshold, we determined that the samples meeting this criterion comprise 94.3% of the total. Finally, among all executed samples, only 94 samples from 22 families demonstrated permanent modifications to executables, thereby being classified as file infectors. The families *lamer* and *xolxo* stand out as exceptions because they do not exhibit any Windows API calls, but they do show executables permanently modified, indicating successful detonation and file infection. Consequently, these families are classified among the file infector families.

Table 5.1: Summary of modification of executed samples per family.

family	**exec**	**file open**	**file writ**	**file del**	**permanent exe mod**	**win api**
adgazelle	5	159	41	41	0	66811
airinstaller	5	15	3	2	0	21166
alman	5	0	0	1	0	383
atcpa	5	20	5	0	0	138878
babar	5	50	8	6	0	14122
benjamin	5	1010	1005	0	0	142272
catalina	5	700	340	15	0	48723
chir	5	16	6	0	0	39536
contenedor	5	0	0	0	0	0
copidmbe	5	3	1	0	0	2820
cosmu	5	15253	85	1	0	1761724
dealply	5	34	2	0	0	1064597
detroie	5	122	33	15	8(5)	30186
egroupdial	5	66	75	21	0	17974
expiro	5	223	133	111	209(5)	109292
fesber	5	95	315	2	0	344515
fickerstealer	5	3	3	1	0	67397
firseria	5	225	10	6	0	567171
fujacks	5	109	65	14	0	737272
geegly	5	5	5	3	0	1430798
getnow	5	107	56	81	0	132445
gogo	5	88	45	36	25(4)	281841
grenam	5	0	0	0	276(4)	2
hematite	5	6	2	0	0	273
imali	5	224	39	20	0	66246
induc	5	152	75	86	24(3)	1166071
installbrain	5	58	93	10	0	1037398
installcore	5	43	210	6	0	1335920
jeefo	5	55	113	4	0	40623
klez	5	4	4	0	0	2179
kolabc	5	0	0	0	0	884
lamer	5	0	0	0	109(5)	0
lassorm	5	56	54	11	0	907889
lmir	5	40	25	15	5(5)	20688

family	**exec**	**file open**	**file writ**	**file del**	**permanent exe mod**	**win api**
mabezat	5	114	12	9	0	12055
memery	5	8828	160	0	105(5)	907040
mepaow	5	28	4	0	0	46302
mywebsearch	5	100	105	10	0	31931
neshta	5	794	645	40	440(5)	476875
parite	5	1706	1703	0	0	362295
pidgeon	5	3	1	0	412(5)	237
pioneer	5	360	21	14	0	33101
plemood	5	0	0	0	0	0
ramnit	5	7	2	2	0	603541
rungbu	5	659	51	66	0	410868
sality	5	4	3	2	0	4035
shodi	5	276	97	88	49(4)	285574
sinau	5	277	33	4033	8(4)	230001
sivis	5	18653	17799	0	18(5)	369973
slugin	5	30	8	0	0	563146
soulclose	5	1240	935	1224	658(5)	2109916
stihat	5	43	12	6	24(3)	51614
stone	5	0	0	0	0	0
tempedreve	5	15	15	0	0	3276
tenga	5	24	0	0	0	4682
triusor	5	884	40	0	28(4)	341681
tufik	5	391	1589	143	0	1052300
tupym	5	120	10	0	0	2838596
ursnif	5	25	0	0	0	24819
usteal	5	26	16	5	0	14407
viking	5	38	34	15	0	21525
virfire	5	6	2	0	0	2024802
virlock	5	750	796	761	58(4)	435165
virut	5	0	0	0	0	61
vitro	5	34	22	2	0	82926
wapomi	5	805	103	6	27(5)	1441027
wlksm	5	4353	72	0	36(3)	399018
xiaobaminer	5	11812	1671	0	172(5)	2171895
xolxo	5	0	0	0	84(5)	0
xorer	5	999	736	20	13(1)	153833

5.2 File Infector Characterization

The preview section provides an overview of the 350 analyzed samples. Among them, we identified 94 file infectors belonging to 22 families. Moving forward, our analysis will exclusively focus on these samples. We conducted manual analysis to establish ground truth regarding the types of file infectors. As depicted in Table 5.2, we identified three potential file infector types: Appender, Prepender, and *Impersonator*. The Impersonator is a classification of file infector type introduced by our study to delineate a specific infection behavior. Each type manifests distinct behavior patterns, facilitating its characterization. Detailed explanations of these file infector types will be provided in the subsequent sections.

The Table is organized into eight columns. Initially, the *"family"* column delineates the file infector families, while the *"type"* column categorizes them by their file infector types. Following this are three columns detailing modifications to the section table: *"st added"*, *"st extend"*, and *"st remov"*, indicating the average number of sections added, extended, or removed, respectively. The sixth column, *"es mod"*, is a boolean value denoting whether modifications were made to the entry section, the first section executed, as pointed by the entry point. Similarly, the seventh column, *"ep mod"*, is a boolean value indicating modifications to the entry point. Finally, the last column, *"overlay ratio"*, quantifies the average ratio between the size of the overlay and the original file size for each pair of original and infected files. This ratio ranges from zero to infinity.

5.2.1 Appenders

An Appender is a type of file infector that injects malicious code into executables by appending it to the end of the file. Additionally, they typically modify either the entry point or the section that contains the entry point (i.e., the entry section) to facilitate the execution of the appended malicious code. It is important to note that Appenders do not exhibit an overlay.

In our analysis of Appenders, we identified two primary behaviors. Firstly, *expiro* and *wlksm* modify the section table to extend the final section and append malicious code to it. They do not modify the entry point itself; instead, they modify the entry section. The family *wlksm* typically modifies the entry section by a small number of bytes, which include the invocation of the *VirtualAlloc* Windows function to allocate memory. In contrast, *expiro* modifies a significantly larger portion of the entry section. The second behavior is shared by *triusor* and *wapomi*. Instead of extending the last section, they modify the section table to add one or more sections for storing the malicious code. Unlike the first behavior, they do not alter the entry section but directly modify the entry point address within the headers. This modification directs the execution flow to the newly created section containing the malicious code.

Table 5.2: Summary of modifications performed by identified file infectors per family.

family	**type**	**st added**	**st extend**	**st remov**	**es mod**	**ep mod**	**overlay ratio**
expiro	A	0	1	0	✓	✗	✗
triusor	A	4	0	1	✗	✓	✗
wapomi	A	1	0	0	✗	✓	✗
wlksm	A	0	1	0	✓	✗	✗
lamer	P	3	0	all	-	-	637.2
induc	P	3	0	all	-	-	1.0
neshta	P	8	0	all	-	-	1.0
shodi	P	4	0	all	-	-	6.4
sinau	P	8	0	all	-	-	1.0
sivis	P	3	0	all	-	-	4.8
soulclose	P	3	0	all	-	-	1.0
xiaobaminer	P	7	0	all	-	-	53.3
memery	P	4	0	all	-	-	55.9
pidgeon	P	26	0	all	-	-	0.8
detroie	P	8	0	all	-	-	20.3
gogo	P	3	0	all	-	-	33.1
lmir	P	8	0	all	-	-	0.1
stihat	P	18	0	all	-	-	13.4
xolxo	P	70	0	all	-	-	82.6
xorer	P	3	0	all	-	-	1.8
virlock	I	2	0	all	-	-	✗
grenam	I	10	0	all	-	-	✗

5.2.2 Prependers

A Prepender represents a specific type of file infector that injects malicious code into executables by adding it to the beginning of the file. These entities significantly alter the infected file by adding themselves at the file's outset and modifying almost all the aspects from the executable's headers to its data sections. In addition, Prependers exhibit an overlay, a section of additional data appended to the end of the PE file beyond the last defined section. This overlay remains unnoticed during execution and can contains the original executable file, sometimes with alterations.

Among the 16 families identified as Prependers, we observed a consistent pattern of behavior. As indicated in Table 5.2, these families completely modify the section table by removing the original sections and inserting their own, without extending any section. The table does not provide specific information for the *"es mod"* or *"ep mod"* columns. This omission is due to the fact that since these Prependers entirely replace the original content of infected executables, both the entry point and the entry section must consequently undergo modification. An interesting observation arises from the *"overlay ratio"* column in

Table 5.2, which quantifies the extent of modifications to the original file potentially stored in the overlay. For instance, some families, like *nestha*, store the original version entirely unmodified, and extracting only this overlay would yield a new PE executable. Conversely, other families, such as *xiaobaminer*, completely alter the original executable, resulting in a meaningless blob of bytes or sometimes nested executables. When referring to "nested executable", we mean that certain families, like *memery*, exhibit a layered structure in their overlay. This structure allows for the recursive extraction of various PE executable versions from the overlay of the previously extracted one. One potential explanation for this phenomenon could be related to an infection error resulting from uncontrolled infections of already infected executables, as discussed in a previous study [38].

5.2.3 Impersonators

An Impersonator is a distinct type of file infector that injects malicious code by completely overwriting the infected file. These infectors entirely transform the infected file by modifying all its aspects, from the executable's headers to its data sections. Unlike other types of infectors, Impersonators do not have an overlay; they do not retain the original file in any form. Instead, they directly replace the original executable with their modified version.

From the Table 5.2, we observe that only two families have been classified as Impersonators. Similar to Prependers, these families manipulate the section table by replacing the original sections with their own, without extending any section. Consequently, the Table 5.2 does not provide specific information for the *"es mod"* or *"ep mod"* columns, as for the previous section.

After identifying the file infectors, it is now possible to temporally locate them and assess whether they remain a contemporary concern. Employing a similar approach as outlined in Chapter 3, we observed that all the file infectors are temporally confined between 2016 and 2022, as depicted in Figure 5.1, with a peak in 2021. This observation indicates that they were recently detected by VirusTotal for the first time. It is important to note that these results do not ascertain the age of specific samples or their respective families, as they may have been previously infected by other malicious files rather than being the original file infector. However, it is evident that file infectors continue to propagate in contemporary times.

5.3 Similarity Scores

We concentrated our similarity analysis on the 94 samples of the 22 families classified as file infectors, which exhibited permanent modifications to executables. The outcomes, categorized by type, are outlined in Table 5.3.

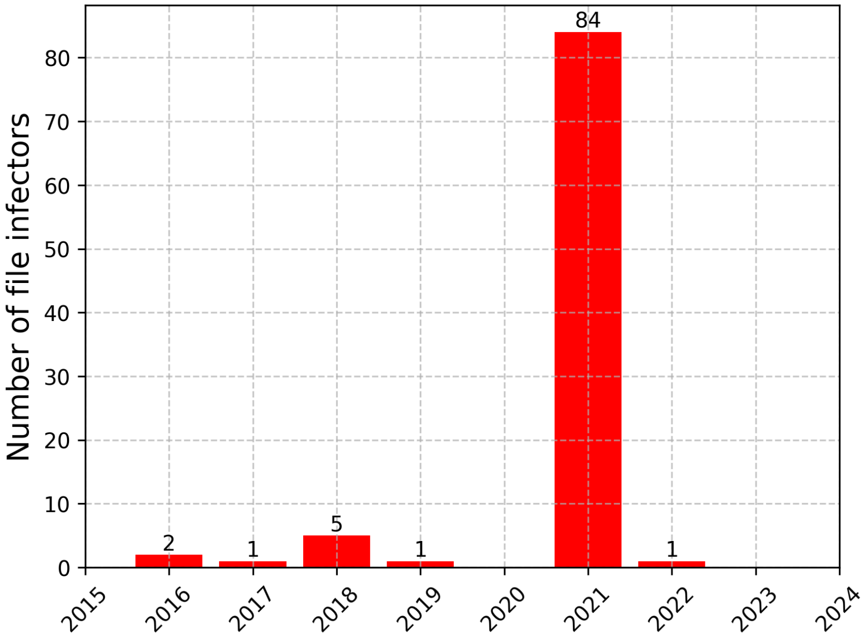


Figure 5.1: Number of file infectors first seen by VirusTotal per year.

The Table is structured into eight columns. Firstly, the *"family"* column represents the file infector families, while the *"type"* column categorizes them based on their file infector types. Moving forward, the *"orig mod"* column presents the average similarity score computed between each original executable version and its infected counterpart. Subsequently, the *"orig mod sized"* column displays the average similarity score computed between each original executable version and its infected counterpart, but truncated to the original file size. Following that, the *"orig mod just over"* column shows the average similarity score computed between each original executable version and just the overlay of its infected counterpart, if present. Further, the *"mod mod"* column displays the average similarity score calculated among all possible pairs of infected executables. Subsequently, the *"mod mod no over"* column shows the average similarity score among all possible pairs of infected executables, excluding their overlays, if present. Finally, the *"over occur"* column presents the occurrence of overlays in the infected files compared with their original versions.

Starting with Appenders, we can identify common characteristics. These infectors inject malicious code at the end of the file, typically altering only a small fraction of the entire executable. This behavior is reflected in the high average similarity score between the original and truncated infected executable versions. Similarly, since the modifications do not impact the entire file, the average similarity scores between modified executables are consistently lower. As there is no overlay present, the fifth column remains empty. The family *wapomi* is an exception, as some infected files exhibit an overlay. This behavior is likely a result of an infection error, as aside from the creation of the overlay, no other

Table 5.3: Summary of similarity results performed on identified file infectors per family.

family	**type**	**orig mod**	**orig mod sized**	**orig mod just over**	**mod mod**	**mod mod no over**	**over occur**
expiro	A	36.8	95.4	✗	17.4	✗	✗
triusor	A	73.0	97.0	✗	41.5	✗	✗
wapomi	A	72.2	100.0	3.0	71.6	56.2	5.2
wlksm	A	50.0	98.0	✗	37.7	✗	✗
detroie	P	0.0	4.8	16.0	96.0	99.0	1.0
gogo	P	5.5	42.8	3.5	73.0	100.0	6.2
induc	P	42.0	7.3	100.0	73.3	95.0	8.0
lamer	P	15.6	23.2	12.2	71.8	92.8	20.4
lmir	P	50.6	50.6	0.0	✗	✗	1.0
memery	P	0.0	51.2	0.0	98.8	100.0	16.0
neshta	P	82.0	83.0	99.0	14.0	84.4	72.0
pidgeon	P	29.8	28.0	78.2	13.2	26.8	63.8
shodi	P	40.8	49.8	37.2	48.5	98.2	11.5
sinau	P	34.5	15.0	98.0	57.8	96.0	1.0
sivis	P	86.8	88.8	24.0	41.4	100.0	18.0
soulclose	P	72.2	60.2	95.2	20.8	98.4	112.4
stihat	P	1.3	0.0	14.0	96.3	100.0	8.0
xiaobaminer	P	10.4	33.0	4.4	77.2	100.0	148.6
xolxo	P	4.8	20.2	8.0	86.2	100.0	15.8
xorer	P	42.0	42.0	44.0	21.0	91.0	11.0
grenam	I	6.0	17.2	✗	99.2	✗	✗
virlock	I	8.0	12.8	✗	14.0	✗	✗

parts are modified, rendering the overlay useless.

Prependers demonstrate a consistent behavior pattern. Since they extensively modify most, if not all, of the target executables, the average similarity score between the original and infected versions tends to be notably low, occasionally even reaching zero for specific families. However, the most crucial aspect among these infected files lies in the overlay, which may contain their original versions. Removing this overlay yields higher similarity scores, effectively delineating this type of file infector. The family *lmir* represents an exception because all the analyzed samples infected only one file, making it impossible to calculate the metrics between infected files.

Impersonators, much like Prependers, extensively modify the target executables. This behavior is reflected in the similarity scores, as the average similarity score between the original and infected executables tends to be quite low, as anticipated. Conversely, the average similarity scores between modified executables are higher. Similar to Appenders, Impersonators do not present an overlay.

5.4 Classifier

In this section, we present the outcomes of the multiclass classification conducted with a Random Forest classifier. The dataset was statically partitioned into a Training set, encompassing 70% of the samples, and a Testing set, encompassing 30% of the samples. The classification results are summarized in Table 5.4. The obtained results indicate remarkably high precision values, with all precision scores being 1.0. While these results may seem promising, it is essential to note that they could be indicative of overfitting, especially given the relatively small size of the dataset consisting of only 94 file infectors. Overfitting occurs when a model learns the training data too well, capturing noise in the data rather than underlying patterns. Therefore, further investigation is necessary to validate the robustness of the classification model and ensure its generalizability to unseen data.

Table 5.4: Overall of classification results using Random Forest.

	Accuracy	**Macro Avg**	**Weighted Avg**	**A**	**P**	**I**	**U**
Precision	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Recall	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F1-score	1.0	1.0	1.0	1.0	1.0	1.0	1.0

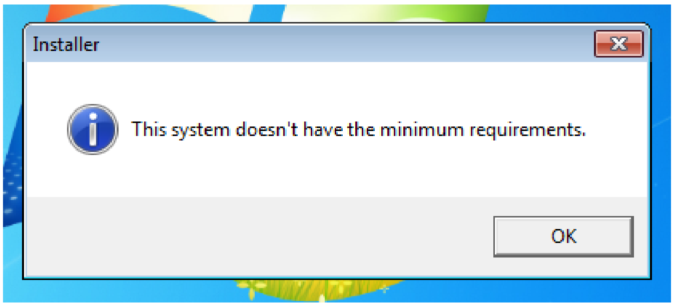
Chapter 6

Conclusions

In this thesis, we presented a novel framework for the analysis of file infection malware, able to detect file infectors and classify them into different types based on their infection behaviors. Through the analysis of a dataset comprising 350 samples, we demonstrated the effectiveness of our framework in accurately classifying 94 file infectors, belonging to 22 families, into distinct types (e.g., Appender, Prepender, Impersonator). Despite the limitations associated with the analyzed dataset size and the potential risk of overfitting in the classification model, our framework shows promise in improving the understanding and characterization of file infectors.

6.1 Limitations

A limitation intrinsic to dynamic analysis is that a fraction of samples may not detonate due to various reasons. Samples might fail to detect components, encounter incompatible versions, or detect the sandbox environment, thereby not exhibiting their malicious behavior. Figure 6.1 illustrates one of the many error message windows displayed when a sample fails to detect components or encounters incompatible versions during manual execution. As a result, the system may determine that a sample is not a file infector simply because it was not able to capture sample detonation and thus observe permanently modified executables. Another limitation is that our dataset, extracted from the original dataset by Dambra et al. using the AVClass2 tool, may not include all file infectors present within it. We did not discover different types of file infectors beyond those discussed in our work, such as Cavity Infectors [4], despite their theoretical mention. Another drawback of this study is the relatively small number of samples analyzed. Due to resource constraints and time limitations, we were only able to analyze a limited dataset comprising 350 samples. This restricted sample size may not fully capture the diversity and complexity of file infectors in the wild, potentially limiting the generalizability of our findings. Another limitation is the


Installer

This system doesn't have the minimum requirements.

Figure 6.1: Example of error message window.

risk of overfitting in the Random Forest classifier. Despite its high precision and accuracy, it may have memorized the training data too well, resulting in poor generalization on unseen data. Overfitting can happen when the model is overly complex or the training dataset is too small. These limitations underscore the need for caution when interpreting the results of this study and highlight areas for future research and improvement.

6.2 Future Work

As the thesis concludes, it is imperative to outline potential avenues for future research endeavors that can build upon the foundations laid in this study. The following themes represent promising directions for advancing the fields of file infector analysis and characterization. One crucial aspect of future research involves the analysis of a more extensive and diverse dataset of samples. While our study was limited to 350 samples due to resource constraints, future research endeavors should prioritize the analysis of a more extensive sample pool. By expanding the sample size, researchers can gain deeper insights into the behavior of file infectors across various contexts and environments, enhancing the robustness and generalizability of the proposed framework. In addition to refining current methodologies, future research initiatives should investigate alternative features for file infector classification. Although similarity scores derived from similarity analysis served as the primary feature in our study, there is potential to incorporate additional features beyond similarity scores. Researchers could explore the integration of supplementary features, such as distinct disparities in specific section table entries, to augment their analysis and develop a more robust classifier. Improving the capabilities of PE executable differ tool represents another area for future exploration. Addressing challenges such as the hierarchical alignment problem is essential for enhancing the accuracy and reliability of this tool in identifying and comparing modifications between PE files.

Bibliography

[1] John E Sawyer, Mary C Kernan, Donald E Conlon, and Howard Garland. «Responses to the Michelangelo Computer Virus Threat: The Role of Information Sources and Risk Homeostasis Theory 1». In: *Journal of Applied Social Psychology* 29.1 (1999), pp. 23–51.

[2] Berni Dwan. «The Computer Virus—From There to Here.: An Historical Perspective.» In: *Computer Fraud & Security* 2000.12 (2000), pp. 13–16.

[3] Savino Dambra, Yufei Han, Simone Aonzo, Platon Kotzias, Antonino Vitale, Juan Caballero, Davide Balzarotti, and Leyla Bilge. «Decoding the Secrets of Machine Learning in Malware Classification: A Deep Dive into Datasets, Feature Extraction, and Model Performance». In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. <conf-loc>, <city>Copenhagen</city>, <country>Denmark</country>, </conf-loc>: Association for Computing Machinery, 2023, pp. 60–74. ISBN: 9798400700507. DOI: `10.1145/3576915.3616589`. URL: `https://doi.org/10.1145/3576915.3616589`.

[4] Eric Filiol. *Computer viruses: from theory to applications*. Springer Science & Business Media, 2006.

[5] Fred Cohen. «Computer viruses: Theory and experiments». In: *Computers & Security* 6.1 (1987), pp. 22–35. ISSN: 0167-4048. DOI: `https://doi.org/10.1016/0167-4048(87)90122-2`. URL: `https://www.sciencedirect.com/science/article/pii/0167404887901222`.

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

[7] Jerome H Saltzer and Michael D Schroeder. «The protection of information in computer systems». In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308.

[8] Ed Skoudis and Lenny Zeltser. *Malware: Fighting malicious code*. Prentice Hall Professional, 2004.

[9] Jesse Kornblum. «Identifying almost identical files using context triggered piecewise hashing». In: *Digital Investigation* 3 (2006). The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06), pp. 91–97. ISSN: 1742-2876.

DOI: `https://doi.org/10.1016/j.diin.2006.06.015`. URL: `https://www.sciencedirect.com/science/article/pii/S1742287606000764`.

[10] Vassil Roussev. «An evaluation of forensic similarity hashes». In: *digital investigation* 8 (2011), S34–S41.

[11] Jonathan Oliver, Chun Cheng, and Yanggui Chen. «TLSH – A Locality Sensitive Hash». In: *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. 2013, pp. 7–13. DOI: `10.1109/CTC.2013.9`.

[12] Jason Upchurch and Xiaobo Zhou. «Variant: a malware similarity testing framework». In: *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE. 2015, pp. 31–39.

[13] Ahmad Azab, Robert Layton, Mamoun Alazab, and Jonathan Oliver. «Mining malware to detect variants». In: *2014 fifth cybercrime and trustworthy computing conference*. IEEE. 2014, pp. 44–53.

[14] Fabio Pagani, Matteo Dell'Amico, and Davide Balzarotti. «Beyond Precision and Recall: Understanding Uses (and Misuses) of Similarity Hashes in Binary Analysis». In: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. CODASPY '18. Tempe, AZ, USA: Association for Computing Machinery, 2018, pp. 354–365. ISBN: 9781450356329. DOI: `10.1145/3176258.3176306`. URL: `https://doi.org/10.1145/3176258.3176306`.

[15] Halvar Flake. «Structural comparison of executable objects». In: *DIMVA 2004, July 6-7, Dortmund, Germany* (2004).

[16] Using SABRE BinDiff. *v1. 6 for Malware analysis.*

[17] Mahmoud Kalash, Mrigank Rochan, Noman Mohammed, Neil D. B. Bruce, Yang Wang, and Farkhund Iqbal. «Malware Classification with Deep Convolutional Neural Networks». In: *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 2018, pp. 1–5. DOI: `10.1109/NTMS.2018.8328749`.

[18] Yanfang Ye, Tao Li, Donald Adjeroh, and S. Sitharama Iyengar. «A Survey on Malware Detection Using Data Mining Techniques». In: *ACM Comput. Surv.* 50.3 (June 2017). ISSN: 0360-0300. DOI: `10.1145/3073559`. URL: `https://doi.org/10.1145/3073559`.

[19] Ömer Aslan Aslan and Refik Samet. «A Comprehensive Review on Malware Detection Approaches». In: *IEEE Access* 8 (2020), pp. 6249–6271. DOI: `10.1109/ACCESS.2019.2963724`.

[20] Razvan Pascanu, Jack W. Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. «Malware classification with recurrent networks». In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2015, pp. 1916–1920. DOI: `10.1109/ICASSP.2015.7178304`.

[21] *VirusTotal*. Accessed: 11/2023. URL: `https://www.virustotal.com`.

[22] Silvia Sebastián and Juan Caballero. «AVclass2: Massive Malware Tag Extraction from AV Labels». In: *Proceedings of the 36th Annual Computer Security Applications Conference*. ACSAC '20. <conf-loc>, <city>Austin</city>, <country>USA</country>, </conf-loc>: Association for Computing Machinery, 2020, pp. 42–53. ISBN: 9781450388580. DOI: `10.1145/3427228.3427261`. URL: `https://doi.org/10.1145/3427228.3427261`.

[23] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. «Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts». In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 1009–1024. DOI: `10.1109/SP.2017.42`.

[24] Lorenzo Maffia, Dario Nisi, Platon Kotzias, Giovanni Lagorio, Simone Aonzo, and Davide Balzarotti. *Longitudinal Study of the Prevalence of Malware Evasive Techniques*. 2021. arXiv: `2112.11289 [cs.CR]`.

[25] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. «Prudent Practices for Designing Malware Experiments: Status Quo and Outlook». In: *2012 IEEE Symposium on Security and Privacy*. 2012, pp. 65–79. DOI: `10.1109/SP.2012.14`.

[26] LordNoteworthy. *Al-Khaser*. Accessed: 01/2024. 2021. URL: `https://github.com/LordNoteworthy/al-khaser`.

[27] Alberto Ortega. *Pafish*. Accessed: 01/2024. 2021. URL: `https://github.com/a0rtega/pafish`.

[28] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. «Pinpointing Representative Portions of Large Intel ® Itanium ® Programs with Dynamic Instrumentation». In: *37th International Symposium on Microarchitecture (MICRO-37'04)*. 2004, pp. 81–92. DOI: `10.1109/MICRO.2004.28`.

[29] *Joe Sandbox*. Accessed: 02/2024. URL: `https://www.joesecurity.org`.

[30] sleuthkit. *The Sleuth Kit*. Accessed: 01/2024. URL: `https://github.com/sleuthkit/sleuthkit`.

[31] *pickle*. Accessed: 01/2024. URL: `https://docs.python.org/3/library/pickle.html`.

[32] Ero Carrera. *pefile*. Accessed: 10/2023. 2023. URL: `https://github.com/erocarrera/pefile`.

[33] George Webster, Bojan Kolosnjaji, Christian Pentz, Julian Kirsch, Zachary Hanif, Apostolis Zarras, and Claudia Eckert. «Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage». In: July 2017, pp. 119–138. ISBN: 978-3-319-60875-4. DOI: `10.1007/978-3-319-60876-1_6`.

[34] Platon Kotzias, Srdjan Matic, Richard Rivera, and Juan Caballero. «Certified PUP: Abuse in Authenticode Code Signing». In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 465–478. ISBN: 9781450338325. DOI: `10.1145/2810103.2813665`. URL: `https://doi.org/10.1145/2810103.2813665`.

[35] David MacKenzie, Paul Eggert, and Richard Stallman. *GNU Diffutils Reference Manual.* London, GBR: Samurai Media Limited, 2015. ISBN: 9789888381548.

[36] Frank Breitinger, F Breitinger, D White, B Guttman, M McCarrin, and V Roussev. *Approximate matching: definition and terminology.* US Department of Commerce, National Institute of Standards and Technology, 2014.

[37] Jason Upchurch and Xiaobo Zhou. «Variant: a malware similarity testing framework». In: *2015 10th International Conference on Malicious and Unwanted Software (MALWARE).* 2015, pp. 31–39. DOI: 10.1109/MALWARE.2015.7413682.

[38] *virusbulletin.* Accessed: 01/2024. 2007. URL: https://www.virusbulletin.com/virusbulletin/2007/01/great-prepender-w32-nubys.