Dissertation

# Large-Scale Dynamic Malware Analysis

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften unter der Leitung von


Priv.-Doz. Dipl.-Ing. Dr. Engin Kirda
und
Priv.-Doz. Dipl.-Ing. Dr. Christopher Kruegel

Institut für Rechnergestützte Automation
Arbeitsgruppe Automatisierungssysteme (E183-1)


eingereicht an der Technischen Universität Wien,
Fakultät für Informatik

von


Dipl.-Ing. Ulrich Bayer

Ludwig Hinnerthstrasse 5
3021 Pressbaum
9926630


Wien, 13. Dezember 2009

# Kurzfassung

Malware (Computerviren, Trojaner, etc.) stellen heutzutage eines der größten Sicherheitsprobleme im Internet dar. Antiviren-Firmen bekommen typischerweise zehntausende neue Malware Dateien pro Tag. Um mit diesen großen Datenmengen zurecht zukommen, haben Forschung und Industrie gleichermaßen an der Entwicklung von automatisierten, dynamischen Malware-Analyse-Systemen gearbeitet. Solche Systeme führen ein zu untersuchendes Programm in einer speziell präparierten Umgebung aus und erstellen einen Bericht, der das Verhalten des Programms wiedergibt. Anubis [1, 28], ein Programm, das größtenteils vom Autor entwickelt wurde, ist ein Beispiel solch eines automatisierten, dynamischen Analyse-Systems.

Um eine Analyse durchzuführen, überwacht Anubis den Aufruf von wichtigen Windows-API- Funktionen und System-Calls, zeichnet Netzwerkverkehr auf, und verfolgt Datenströme. Für jede zu untersuchende Datei wird ein Bericht erstellt, der die Aktivitäten (u.a. welche Dateien, Windows-Registry-Keys, Windows-Services erzeugt worden sind) des Programms beinhaltet. Anubis erhält Dateien durch ein öffentliches Web-Interface [1] und durch eine Anzahl von Kooperationsabkommen mit Sicherheitsfirmen. Da die Dateien von einer Vielzahl an Benutzern stammen, stellen die von Anubis gesammelten Dateien eine umfassende und repräsentative Auswahl der gängigen Viren dar.

In dieser Dissertation stellen wir neuartige Techniken vor, um die Anwendung von automatisierter, dynamischer Analyse im großen Stil zu verbessern:

**Typisches Malware-Verhalten.** Wir versuchen typisches Malware-Verhalten zu ergründen. Dafür werten wir die Anubis-Analyse-Resultate von fast einer Million Dateien aus, untersuchen Trends und Evolutionen von bösartigem Verhalten über einen Zeitraum von ungefähr zwei Jahren und untersuchen den Einfluss von Polymorphismus auf Malware-Statistiken.

**Skalierbares, verhaltensbasiertes Malware Clustering.** Automatisierte, dynamische Analyse-Systeme erlauben die Analyse tausender bösartiger Programme pro Tag. Jedes Analyseresultat liegt in der Form eines Anubis-Reports vor, der die Aktionen eines Programms zusammenfasst. Jetzt ist man allerdings mit dem Problem konfrontiert, tausende Analyseresultate manuell bearbeiten zu müssen. Vor kurzem haben Forscher mit der Entwicklung von automatisierten Clustering-Systemen zur Identifizierung von Dateien mit ähnlichem Verhalten begonnen. Dadurch hat ein Analyst einer A/V-Firma die Möglichkeit, sich auf interessante, neue Bedrohungen zu konzentrieren, während er Reporte von bereits bekannten Malware-Programmen ignorieren kann. Leider skalieren bisherige Clustering-Systeme schlecht und haben noch Probleme damit, beobachtetes Verhalten gut genug zu generalisieren, um wirklich Malware-Familien zu erkennen.

Wir präsentieren in dieser Dissertation ein skalierbares Clustering-System zur Identifizierung und Gruppierung von Malware-Programmen mit ähnlichem Verhalten. Zu diesem Zweck führen wir zuerst eine dynamische Analyse durch, um eine Aufzeichnung der Funktionsaufrufe zu erhalten. Danach generalisieren wir diese Aufzeichnungen zu sogenannten Verhaltensprofilen. Ein Verhaltensprofil charakterisiert das Verhalten eines Malware-Programms auf einer höheren Abstraktionsebene. Diese Verhaltensprofile dienen als Eingabe für unseren effizienten Clustering-Algorithmus, der es uns erlaubt Malware-Programme um eine Größenordnung schneller zu clustern als bisherige Techniken. Weiters haben wir unser System mit Malware-Sammlungen aus der realen Welt getestet. Die Resultate demonstrieren, dass unsere Technik in der Lage ist Ähnlichkeiten zwischen Malware-Programmen zu erkennen und entsprechende Gruppen zu bilden. Dabei erzielen wir genauere Resultate als bisherige Techniken. Um die Skalierbarkeit des Systems zu zeigen, haben wir für eine Menge von fünfundsiebzigtausend Malware-Programmen in weniger als drei Stunden erfolgreich ein Clustering erstellt.

**Effizienzsteigerung für die dynamische Malware Analyse.** Innerhalb der letzten drei Jahre stieg die Anzahl an Malware-Programmen, die sich im Umlauf befinden, um den Faktor 10. Leider ist anzunehmen, dass die Anzahl an neuen Malware-Dateien in Zukunft noch weiter zunehmen wird. Damit die automatisierte, dynamische Analyse mit dieser Entwicklung Schritt halten kann - ohne weitere Kosten für neue Hardware zu verursachen - haben wir eine Technik entwickelt, die die Zeit für die Analyse einer bestimmten Menge von Programmen drastisch reduziert. Unsere Lösung basiert auf der Erkenntnis, dass die große Anzahl an neuen Malware-Programmen vor allem durch einfache, teilweise automatisierte Mutationen einiger weniger Malware Programme entsteht. Um Analysezeit zu sparen, schlagen wir eine Technik vor, die eine mehrmalige, vollständige Analyse von polymorphen Programmen vermeidet. In einem Experiment mit unserem Prototypen gelang es uns, bei einer Auswahl von 10.922 Dateien in 25,25 Prozent der Fälle eine vollständige Analyse zu vermeiden.

# Abstract

Malicious software (or malware) is one of the most pressing and major security threats facing the Internet today. Anti-virus companies typically have to deal with tens of thousands of new malware samples every day. To cope with these large quantities, researchers and practitioners alike have developed automated, dynamic malware analysis systems. These systems automatically execute a program in a controlled environment and produce a report describing the program's behavior. Anubis [1, 28], a program mainly developed by the author, is an example of such a system.

To perform the analysis, Anubis monitors the invocation of important Windows API calls and system services, it records the network traffic, and it tracks data flows. For each submission, reports are generated that provide comprehensive reports about the activities of the binary under analysis. Anubis receives malware samples through a public web interface and a number of feeds from security organizations and anti-malware companies. Because the samples are collected from a wide range of users, the collected samples represent a comprehensive and diverse mix of malware found in the wild.

This thesis presents novel approaches for performing large-scale dynamic malware analysis:

**A View on Current Malware Behaviors.** We aim to shed light on common malware behaviors. To this end, we evaluate the Anubis analysis results for almost one million malware samples, study trends and evolution of malicious behaviors over a period of almost two years, and examine the influence of code polymorphism on malware statistics.

**Scalable, Behavior-Based Malware Clustering.** Automated, dynamic analysis systems permit the analysis of thousands of malicious binaries per day. Each analysis results in the creation of an analysis report summarizing a program's actions. Of course, the problem of analyzing the reports still remains. Recently, researchers have started to explore automated clustering techniques that help to identify samples that exhibit similar behavior. This allows an analyst to discard reports of samples that have been seen before, while focusing on novel, interesting threats. Unfortunately, previous techniques do not scale well and frequently fail to generalize the observed activity well enough to recognize related malware.

In this thesis, we propose a scalable clustering approach to identify and group malware samples that exhibit similar behavior. For this, we first perform dynamic analysis to obtain the execution traces of malware programs. These execution traces are then generalized into behavioral profiles, which characterize the activity of a program in more abstract terms. The profiles serve as input to an efficient clustering algorithm that allows us to handle sam-

ple sets that are an order of magnitude larger than previous approaches. We have applied our system to real-world malware collections. The results demonstrate that our technique is able to recognize and group malware programs that behave similarly, achieving a better precision than previous approaches. To underline the scalability of the system, we clustered a set of more than 75 thousand samples in less than three hours.

**Improving the Efficiency of Dynamic Malware Analysis.** During the last three years, the number of malware programs appearing each day has increased by a factor of ten, and this number is expected to continue to grow. To keep pace with these developments without causing even more hardware costs for operating dynamic analysis systems, we have developed a technique that drastically reduces the overall analysis time. Our solution is based on the insight that the huge number of new malicious files is due to mutations of only a few malware programs. To save analysis time, we suggest a technique that avoids performing a full analysis of the same polymorphic file multiple times. In an experiment conducted on a set of 10,922 randomly chosen executable files, our prototype implementation was able to avoid a full dynamic analysis in 25.25 percent of the cases.

# Acknowledgments

First of all, I would like to thank my advisors Prof. Dr. Christopher Kruegel and Prof. Dr. Engin Kirda for making this PhD thesis possible. They gave me the chance to work on a very interesting subject at the forefront of scientific research, provided me with financial support, inspired me, and showed me how to conduct scientific research. By taking care of many of the administration issues, they allowed me to fully concentrate on my research as well as the Anubis project. I'm grateful to Engin Kirda for giving me the opportunity to spend part of my time as a PhD student abroad at Eurecom, in France.

Special thanks go to my colleagues at the Vienna University of Technology and at Eurecom for fruitful discussions, interesting collaborations and a good working environment. In particular, I would like to thank all the persons that were involved in the Anubis project. These are in alphabetical order: Constantin Claudiu Gavrilete, Valentin Habsburg, Clemens Hlauschek, Sylvester Keil, Engin Kirda, M. Levent Koc, Clemens Kolbitsch, Christopher Kruegel Florian Lukavsky, Paolo Milani Comparetti, Andreas Moser, Florian Nentwich, Matthias Neugschwandtner, Martin Schenk, Christoph Schwarz, Michael Weissbacher. Without their contributions, Anubis would not exist as it does today.

In addition, I'm grateful for the collaboration with Ikarus Software GmbH, especially with Josef Pichlmayr and Thomas Mandl. This collaboration provided me with the opportunity to get insights into the real-word aspects of the malware problem. Moreover, the Anubis project would not exist without their constant support through all these years.

I owe thanks to Secure Business Austria, in particular to Prof. Dr. Edgar Weippl and Mag. Markus Klemen, for financially supporting me during this thesis and for providing me with an environment that allowed me to fully focus on my research activities.

Last but not least, I would like thank my parents Gerhild and Josef for their life-long support. Without their assistance through all these years, this work would not have been possible.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

One of the major threats on the Internet today is malicious software, often referred to as malware. In fact, most Internet security problems have malware as their underlying root cause. For example, botnets are commonly used to send spam and host phishing web sites that are more difficult to track down and blacklist. Malware comes in a wide range of forms and variations, such as viruses, worms, botnets, rootkits, Trojan horses, and denial of service tools. To spread, malware exploits software vulnerabilities in browsers and operating systems, or uses social engineering techniques to trick users into running the malicious code.

Whereas early malware was not focused on making a financial profit, unfortunately, this is not the case anymore. The success of the web and the lack of technical sophistication and understanding of many web users has attracted criminals, who are well-organized and who aim to make easy money. As a result, the number of malware instances discovered in the wild continues to increase at an alarming rate [14]. To defend against the flood of malware samples, the anti-malware industry has to invest significant effort to develop effective signatures. Unfortunately, the problem of maintaining a signature database and keeping it up-to-date is not trivial.

An anti-malware company typically receives thousands of new malware samples every day. These samples are submitted by users who have found suspicious code on their systems, by other anti-malware companies that share their samples, and by organizations (e.g., MWCollect [15], ShadowServer [17], VirusTotal [20]) that use technologies such as honeypots [70] to collect malware. For each sample, it is important to understand the actions that this program can perform. This is necessary to determine the type and severity of the threat that the malware constitutes. Also, this information is valuable to create detection signatures and removal procedures. In some cases, the sample may turn out to be harmless. Furthermore, in many cases, the malware may turn out to be a variant of a well-known malware instance. In fact, although the malware may remain the same, its signature might change just because the malware author is using a simple obfuscation or polymorphism technique [55, 61, 72]. Hence, for anti-malware organizations, it is typically

easy to obtain many malware samples in the wild, but difficult to analyze their functionality.

Because of the growing need for automated techniques to examine malware, dynamic malware analysis tools such as CWSandbox [5], Norman Sandbox [16], ThreatExpert [19] and ANUBIS [1, 28] have increased in popularity. These systems execute the malware sample in a controlled environment and monitor its actions. Based on the execution traces, reports are generated that aim to support an analyst in reaching a conclusion about the type and severity of the threat imposed by a malware sample. For example, when observing a sample that modifies the autorun registry entry and opens a connection to a notorious IRC server (which is often used for botnet command and control), the analyst can quickly conclude that the sample is an IRC bot. Even when the analyst is not able to reach a detailed conclusion about a sample, the automated analysis is beneficial to help separate interesting samples from those that are less relevant. Clearly, if required, the interesting samples can further be analyzed manually (e.g., by disassembling and debugging the program using reverse engineering tools such as IDA Pro [12]).

Automated analysis systems are nothing but a first step to solve the challenges posed by the large quantities of new malware samples appearing each day. The question arises how we can master the large number of analysis reports that we have created in response to the large number of incoming malicious programs. To address this problem, we present a technique that allows the grouping of samples based on the similarity of their behavior exhibited while running in Anubis. Moreover, we show how to leverage the large body of analysis reports for drawing conclusions about common malware behaviors. Last, we demonstrate that it is possible to improve the efficiency of dynamic malware analysis by making use of the fact that many of today's malware programs are due to mutations of only a few malware programs [45]. This permits us to analyze more samples in the same time without requiring more hardware resources. We will give a more detailed overview of these three solutions in the following paragraphs.

## 1.1 Anubis

Anubis is a project that focuses on automatic malware analysis. By dynamically analyzing the behavior of Windows executables, it addresses the demand for automatic binary analysis that arises from the huge amounts of new malware samples appearing each day. Apart from analyzing individual malware samples, Anubis is capable of providing a high-level view of the malware landscape by clustering samples into larger malware families and by offering various

statistics on common malicious behavior. Anubis offers users the possibility to upload suspicious samples for analysis via a public website [1]. In addition, Anubis receives malware samples through a number of feeds from security organizations and anti-malware companies. At the time of writing this thesis, Anubis analyzes several thousand binaries per day.

Anubis evolved from TTAnalyze [28], a tool for dynamically analyzing malware samples, written by the author during his master's thesis. While TTAnalyze was a command-line tool that analyzes a suspicious Windows executable and subsequently outputs the results into an HTML file, Anubis is a system designed for large-scale malware analysis that consists of many programs, scripts, databases as well as a powerful hardware infrastructure.

Anubis began as a single-man project and over time grew to a project that involved many students and colleagues. Since its beginning the following people contributed to the development of Anubis: Ulrich Bayer, Florian Nentwich, Paolo Milani Comparetti, Constantin Claudiu Gavrilete, Clemens Hlauschek, Clemens Kolbitsch, Florian Lukavsky, Matthias Neugschwandtner, Martin Schenk, Michael Weissbacher, Engin Kirda, Christopher Kruegel, Andreas Moser, Sylvester Keil, M. Levent Koc, Valentin Habsburg, Christoph Schwarz. Without their help Anubis would not exist as it does today.

In this section, we are going to provide a technical overview of the Anubis system. We speak about the practical aspects of designing and operating a large-scale dynamic malware analysis system.

### 1.1.1 Architecture

Anubis has grown from a simple command-line tool for analyzing malware to a full-fledged system for large-scale malware analysis consisting of many components. We list the main components in Figure 1.1.

- *Web/DB Server.* This component provides the public web interface that permits users to submit malware samples to Anubis. At the same time, this server hosts a relational database that we leverage for storing analysis tasks and its results. In particular, we store the produced analysis reports, which exist in the form of XML files, also in the database.

- *Malware Sample Storage.* We store all malware files on the filesystem.

- *Report Storage.* The storage of analysis results takes place on the file system. In addition to the XML report file, we store the network traffic for each analysis as a PCAP file.

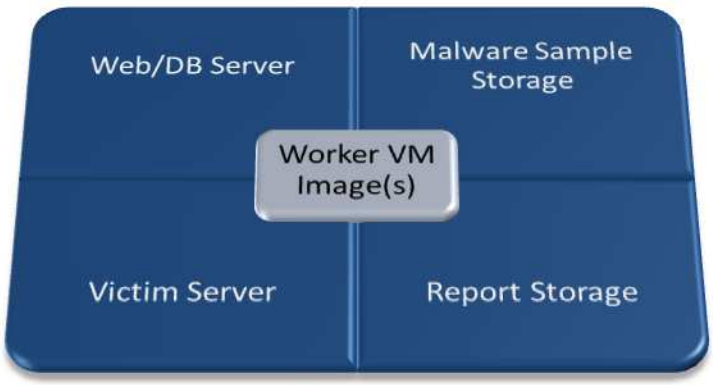


Figure 1.1: Anubis Building Blocks.

- *Worker.* The workers are the servers that perform the actual work of dynamically analyzing a Windows executable. We employ several workers in parallel for improving the analysis throughput. To analyze a malware sample, the binary is run in an emulated operating system environment and its (security-relevant) actions are monitored. In particular, we record the Windows native system calls and Windows API functions that the program invokes. One important feature of our system is that it does not modify the program that it executes (e.g., through API call hooking or breakpoints), making it more difficult to detect by malicious code. Also, our tool runs binaries in an unmodified Windows environment, which leads to excellent emulation accuracy.

- *Victim Server.* When analyzing malware, it is essential to avoid sending malicious traffic out of the analysis environment. For this reason, we redirect the majority of network traffic generated by files under analysis to the so-called *victim server*. This server has been configured to accept incoming connections on a number of ports that are frequently used by malware programs. For example, the victim machine runs its own SMTP server that answers all SMTP requests (but does not deliver any emails). Moreover, we have set up nepenthes [22] - a honeypot system that emulates known vulnerabilities of popular services. Of course, we are not using the nepenthes server as a honeypot system in the usual sense, i.e., as a way to gain new malware samples. Instead, we have deployed nepenthes only for having a basic service listening on ports that are frequently used for spreading (such as the Windows Samba

ports)

**Submission Process.** To clarify the interaction between these components, let us have a look at the submission process. First, a user submits a sample to the Anubis webserver which registers the submission in the database and stores the submitted file in the sample store. In a next step, the Anubis workers process outstanding tasks and store the analysis results in the report store as well as the database.

### 1.1.2 Data Repository

Operating an analysis system that keeps up with the huge number of new malware samples appearing each day means generating several thousands of new results each day. This quickly leads to big amounts of data that need to stored. We designed Anubis to store the raw data (such as submitted files, behavioral reports) on the file system. However, we extract the important information from these files and load them into a relational database. Having this data in a relational database enables us to draw conclusions on global malicious behavior.

The data repository fulfills three different requirements:

- *Task management.* Submissions to Anubis are represented as analysis tasks in the Anubis environment. These tasks, as well as their state (outstanding, completed, currently processed) are managed in the database.

- *Storage of behavioral information.* We load important information from the behavioral summaries (XML files) into the database.

- *Clustering management.* Similar to the task management capabilities, clustering tasks are kept in the database. Moreover, behavioral profiles (a more machine-readable representation of a program's behavior) are stored in the database.

## 1.2 A View on Current Malware Behaviors

In this thesis, we set out to provide insights into common malware behaviors. Our analysis and experiences are based on the malicious code samples that were collected by Anubis [1, 28], our dynamic malware analysis platform. When it receives a sample, Anubis executes the binary and monitors the invocation of important system and Windows API calls, records the network

traffic, and tracks data flows. This provides a comprehensive view of malicious activity that is typically not possible when monitoring network traffic alone.

Anubis receives malware samples through a public web interface and a number of feeds from security organizations and anti-malware companies. These samples are collected by honeypots, web crawlers, spam traps, and by security analysts from infected machines. Thus, they represent a comprehensive and diverse mix of malware found in the wild. Our system has been live for a period of about two years. During this time, Anubis has analyzed almost one million unique binaries (based on their MD5 file hashes). Given that processing each malware program is a time consuming task that can take up to several minutes, this amounts to more than twelve CPU years worth of analysis.

When compiling statistics about the behaviors of malicious code, one has to consider that certain malware families make use of polymorphism. Since samples are identified based on their MD5 file hashes, this means that any malware collection typically contains more samples of polymorphic malware programs than of non-polymorphic families. Unfortunately, this might skew the results so that the behavior (or certain actions) of a single, polymorphic family can completely dominate the statistics. To compensate for this, we analyze behaviors not only based on individual samples in our database but also based on malware families (clusters).

For this thesis, we performed an analysis of almost one million malware samples. The main contribution are statistics about and insights into malicious behaviors that are common among a diverse range of malware programs. We also consider the influence of code polymorphism on malware statistics. To this end, we compare analysis results based on individual samples to results based on malware families. We will present details in Chapter 2.

## 1.3 Clustering Malware

In recent years, there has been progress on automated malware analysis [31, 41, 60, 73, 74]. However, while automating the analysis of the behavior of a single malware sample is a first step, it is not sufficient. The reason is that the analyst is now facing thousands of reports every day that need to be examined. Thus, there is a need to prioritize these reports and guide an analyst in the selection of those samples that require most attention. One approach to process reports is to cluster them into sets of malware that exhibit similar behavior. The ability to automatically and effectively cluster analyzed malware samples into families with similar characteristics is beneficial for the following reasons: First, every time a new malware sample is found in the

wild, an analyst can quickly determine whether it is a new malware instance or a variant of a well-known family. Moreover, given sets of malware samples that belong to different malware families, it becomes significantly easier to derive generalized signatures, implement removal procedures, and create new mitigation strategies that work for a whole class of programs.

Grouping individual malware samples into malware families is not a new idea, and clustering and classification methods have already been proposed previously [24, 43, 50, 52, 47]. These approaches, however, generally do not scale well and are too slow for the size of malware sets that anti-malware companies are confronted with. Moreover, these techniques are imprecise, either because their notion of similarity is not tied to a program's actual behavior or because it does not capture a program's behavior well enough. Imprecise in this context either means putting samples of different types into the same group or failing to recognize similar malware programs. We will present our scalable, behavior-based solution in Chapter 3.

## 1.4 Improving the Efficiency of Dynamic Malware Analysis

The main thing to note about dynamic analysis systems is that they are indeed executing the binary for a limited amount of time. Since, typically, malicious programs do not reveal their behavior when only executed for several seconds, dynamic systems are required to monitor the binary's execution for a longer time. This is why dynamic analysis is resource-intensive in terms of necessary hardware and time. Moreover, the sheer number of malware programs appearing each day became high enough to not only challenge manual analysis but also automated, dynamic malware analysis. One needs costly server farms running the dynamic analysis systems to cope with the ever-increasing load (i.e., amount of binaries to be analyzed).

In this thesis, we present a novel and practical approach for improving the efficiency of dynamic malware analysis systems. Our approach is based on the insight that the huge number of new malicious files appearing each day is due to mutations of only a few malware programs [45]. More precisely, malware authors write programs that reproduce polymorphically [72] or employ run-time packing algorithms to create new malware instances that differ on the file level, but exhibit the same behavior. We propose a system that avoids analyzing malware binaries that merely constitute slightly mutated instances of already analyzed polymorphic malware. To detect polymorphic binaries, we have extended our dynamic analysis system to check—after executing the

malware program for only a short time—whether our database of existing analysis reports contains a behaviorally almost identical (for the time frame in question) analysis report. If this is the case, we stop the analysis process and instead, return the existing analysis result.

We will describe this solution in more detail in Chapter 4.

## 1.5 Contributions

In this thesis, we address several of the challenges created by the need to analyze the large quantities of malicious programs currently circulating in the wild. We propose three novel approaches–each one providing a practical solution to one aspect in the field of large-scale automated, dynamic malware analysis. To evaluate our techniques, we have built and successfully run prototype systems. First, we have built and operated Anubis as a public system that allows Internet users to submit suspicious binaries to our analysis. We make use of these results to provide a view on current malware behaviors. Furthermore, we have built a prototype clustering system to assess the quality and performance metrics of our novel clustering technique. To evaluate our approach for improving the efficiency of dynamic malware analysis, we have also built a prototype system and conducted experiments. Summing up, we make the following contributions in this thesis:

- We provide a view on the currently common malicious behavior by analyzing almost one million malware samples. We present statistics about and insights into malicious behaviors that are common among a diverse range of malware programs. We also consider the influence of code polymorphism on malware statistics. To compile these statistics, we evaluate the Anubis results collected by operating Anubis during a time period of approximately two years and analyzing almost one million malware sample.

- We present a novel clustering technique that scales well and produces more precise results than previous approaches. Our technique is based on a dynamic analysis system that monitors the execution of a malware sample in a controlled environment. Unlike many previous systems that operate directly on low-level data such as system call traces, we enrich and generalize the collected data and summarize the behavior of a malware sample in a behavioral profile. These profiles express malware behavior in terms of operating system (OS) objects and OS operations. Moreover, profiles capture a more detailed view of network activity and the ways in which a malware program uses input from the environment.

This allows our system to recognize similar behaviors among samples whose low-level traces appear very different. Finally, we cluster the analyzed samples according to their behavioral profile. We employ a scalable clustering algorithm that avoids calculating $n^2$ distances between all pairs of $n$ samples, and thus, is suitable for clustering large, real-world malware collections.

- We present a novel and practical approach for improving the efficiency of dynamic malware analysis systems. Our approach is based on the insight that the huge number of new malicious files appearing each day is due to mutations of only a few malware programs [45]. To save analysis time, we suggest a technique that avoids performing a full analysis of the same polymorphic file multiple times. In an experiment conducted on a set of 10,922 randomly chosen executable files, our prototype implementation was able to avoid a full dynamic analysis in 25.25 percent of the cases.

## 1.6 List of Publications

This thesis is based on material that was published at academic, peer-reviewed conferences. Chapter 2 presents the results of a study of current malware behavior that was undertaken together with Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel and published at LEET 2009 [26]. Chapter 3's work on clustering malware is based on joint work with Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel and Engin Kirda. It was published at NDSS 2009 [25]. Chapter 4 features work that I did in collaboration with Christopher Kruegel and Engin Kirda for improving the efficiency of dynamic malware analysis. It will be presented at SAC 2010 [27].

# Chapter 2

# A View on Current Malware Behaviors

## 2.1 Introduction

Anubis receives malware samples through a public web interface and a number of feeds from security organizations and anti-malware companies. Because the samples are collected from a wide range of users, the collected samples represent a comprehensive and diverse mix of malware found in the wild. In this chapter, we aim to shed light on common malware behaviors. To this end, we evaluate the Anubis analysis results for almost one million malware samples, study trends and evolution of malicious behaviors over a period of almost two years, and examine the influence of code polymorphism on malware statistics.

## 2.2 Dataset

In this section, we give a brief overview of the data that Anubis collects. As mentioned previously, a binary under analysis is run in an emulated operating system environment (a modified version of Qemu [29]) and its (security-relevant) actions are monitored. In particular, we record the Windows native system calls and Windows API functions that the program invokes. One important feature of our system is that it does not modify the program that it executes (e.g., through API call hooking or breakpoints), making it more difficult to detect by malicious code. Also, our tool runs binaries in an unmodified Windows environment, which leads to good emulation accuracy. Each sample is run until all processes are terminated or a timeout of four minutes expires. Once the analysis is finished, the observed actions are compiled in a report and saved to a database.

Our dataset covers the analysis reports of all files that were submitted to Anubis in the period from February 7th 2007 to December 31st 2008, and that were subsequently analyzed by our dynamic analysis system in the time period between February 7th 2007 and January 14th 2009. This dataset contains 901,294 unique samples (based on their MD5 hashes) and covers a total of

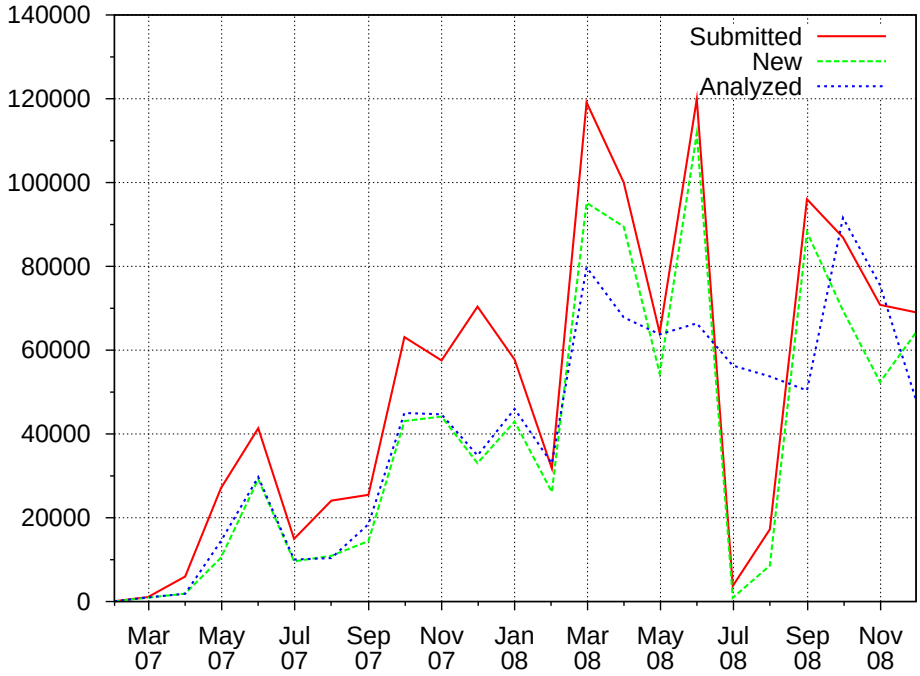


Figure 2.1: Anubis submission statistics.

1,167,542 submissions. Typically, a given sample is only analyzed once by our analysis system. That is, when a sample is submitted again, we return the already existing analysis report without doing an actual analysis.

Figure 2.1 shows the number of total samples, the number of new samples, and the number of actually analyzed samples submitted to Anubis, grouped by months. We consider a file as being *new* when, at the time of its submission, we do not have a file with the same MD5 hash in our repository. As one can see, we have analyzed about fifty thousand samples on average per month in the year 2008. When we first launched the Anubis online analysis service, we received only few samples. However, as the popularity of Anubis increased, it was soon the computing power that became the bottleneck. In fact, in July and August 2008, we had to temporarily stop some automatic batch processing to allow our system to handle the backlog of samples.

Naturally, the Anubis tool has evolved over time. We fixed bugs in later versions or added new features. Given that there is a constant stream of new malware samples arriving and the analysis process is costly, we typically do not rerun old samples with each new Anubis version. Unfortunately, this makes it a bit more difficult to combine analysis results that were produced by different versions of Anubis into consolidated statistics. In some cases, it is possible to

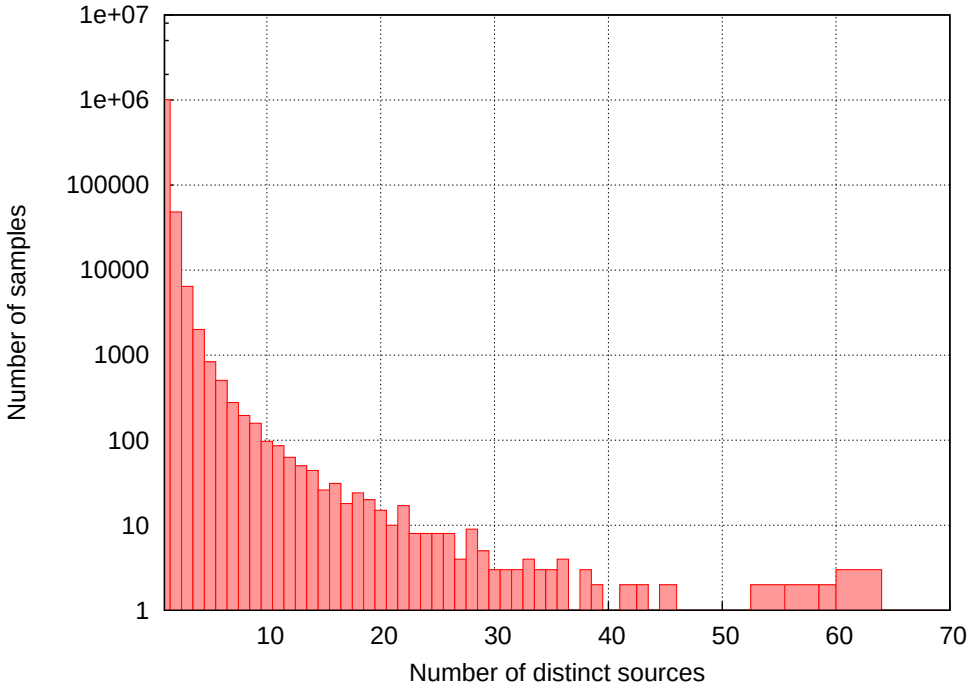


Figure 2.2: Number of distinct sources for each sample.

work around such differences. In other cases (in particular, for the analysis of anti-sandbox techniques presented Section 2.3.6), we had to confine ourselves to results for a smaller subset of 330,088 analyzed PE files. The reason is that necessary information was not present in older reports.

### 2.2.1 Submissions

Figure 2.2 shows the number of different sources that submit a particular sample to Anubis. The graph illustrates that most of the samples we receive are submitted by one source only. Even though the curve decreases quickly, there is still a significant number of samples that are submitted by 10 to 30 different sources.

We have made the experience that measuring the number of sources that submit a certain sample tends to indicate how widespread a certain malware sample is in the wild. In fact, this premise is supported by the results of the anti-virus scans that we run on each sample that we receive. For example, if we consider the samples submitted by one source, 73% of the submissions are categorized by two out of five anti-virus scanners as being malicious. In comparison, 81% of the submissions that are submitted by at least 3 different

sources are identified as being malicious by anti-virus software. Furthermore, among the samples that are submitted by 10 or more sources, 91% are identified as being malicious.

### 2.2.2 Submitted file types

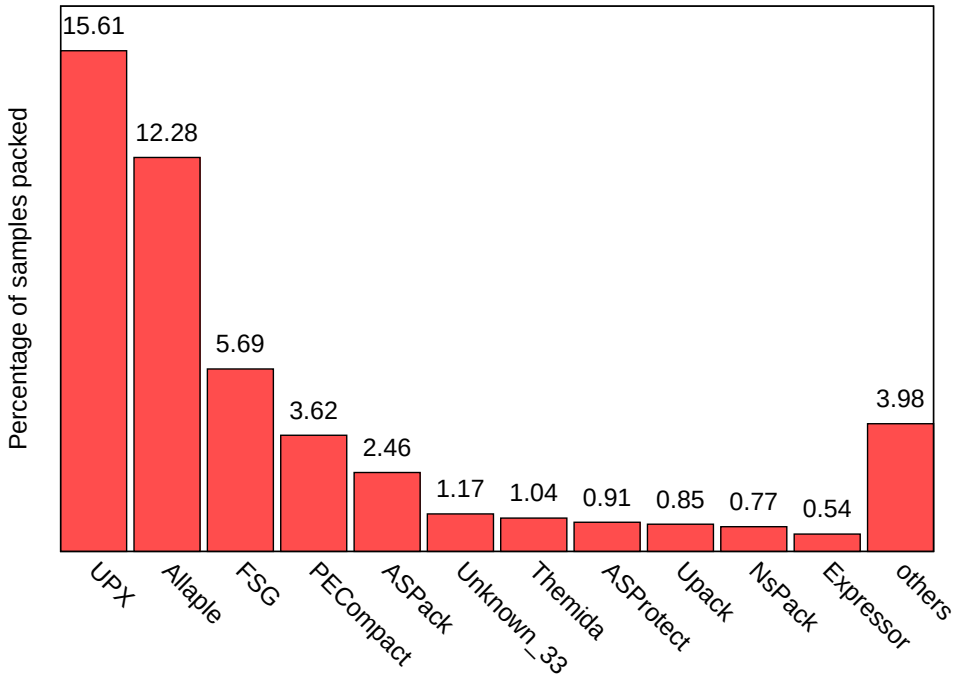


Figure 2.3: Overview of used packers

One problem with running an online, public malware analysis service is that one can receive all sorts of data, not only malware. In fact, users might even try to submit applications such as Microsoft Word or Microsoft Internet Explorer just to see how the system reacts. Furthermore, unfortunately, not all the submitted samples are valid Windows PE executables [66] (around 14% are not). Table 2.1 shows a breakdown of the different file types submitted to Anubis. As can be seen from this table, fortunately for us, most of the files that are sent to Anubis can be analyzed. The category of non PE files includes mostly different archive formats (ZIP and RAR archives) and MS Office documents (such as Word and Excel), but also a small number of shell scripts and executables for different operating systems (such as DOS, Linux). According to SigBuster, a signature-based scanner for packers, 40.64% of the

| | |
|---|---|
| PE files (770,960) | DLL files (75,505)<br>Drivers (4,298)<br>Executables (691,057) |
| Non PE files (130, 334) | ZIP archives (17,059)<br>RAR archives (25,127)<br>HTML files (27,813)<br>Other (60,335) |

Table 2.1: File types submitted to Anubis.

analyzed PE files are packed. Figure 2.3 provides an overview of the most common packers.

### 2.2.3 Submission sources

Over the two-year time period that we have provided the service, Anubis received samples from more than 120 different countries. Depending on the number of samples submitted, we have grouped the Anubis submitters into four different categories: large, medium, small, single. We define a large submitter as an entity (i.e., a person, an organization) that has submitted more than one thousand different (per MD5 hash) samples. A medium submitter is an entity that has submitted between 100 and 1,000 different samples. A small submitter is an entity that has submitted between 10 and 100 different samples, and finally, a single submitter is an entity that has submitted less than 10 samples. Table 2.3 summarizes our findings.

Note that there are 20 *large* submitters (with more than one thousand different samples submitted) who account for almost 90% of the Anubis submissions. Interestingly, the number of *single* submitters is very high. However, these users are only responsible for about 5% of the total submissions. According to anti-virus results that we run on every submitted sample, the medium submitters (probably represented by malware analysts) are more reliable in submitting malicious samples (i.e., 75% of their submissions are flagged as being malicious). In comparison, only 50% of the samples submitted by single submitters are identified as being malicious, suggesting that single individuals are probably more likely to submit random files, possibly to test the Anubis system.

| Observed Behavior | Percentage of Samples | Percentage of Clusters |
|---|---|---|
| Installation of a Windows kernel driver: | 3.34% | 4.24% |
| Installation of a Windows service: | 12.12% | 7.96% |
| Modifying the hosts file: | 1.97% | 2.47% |
| Creating a file: | 70.78% | 69.90% |
| Deleting a file: | 42.57% | 43.43% |
| Modifying a file: | 79.87% | 75.62% |
| Installation of an IE BHO: | 1.72% | 1.75% |
| Installation of an IE Toolbar: | 0.07% | 0.18% |
| Display a GUI window: | 33.26% | 42.54% |
| Network Traffic: | 55.18% | 45.12% |
| Writing to stderr: | 0.78% | 0.37% |
| Writing to stdout: | 1.09% | 1.04% |
| Modifying a registry value: | 74.59% | 69.92% |
| Creating a registry key: | 64.71% | 52.25% |
| Creating a process: | 52.19% | 50.64% |

Table 2.2: Overview of observed behavior.

| Submitter Category | Category Members | % of total tasks submitted |
|---|---|---|
| Large (1000-*) | 20 | 89.1% |
| Medium (100-1000) | 112 | 3.8% |
| Small (10-100) | 1279 | 2.5% |
| Single (1-10) | 30944 | 4.5% |

Table 2.3: Submission sources.

## 2.3 Observed Malicious Behavior

In this section, we present detailed discussions on the file, registry, network, and botnet activity that we observed when analyzing the Anubis submissions. The goal is to provide insights into malicious behaviors that are common among a diverse range of malware programs. An overview of interesting activity is shown in Table 2.2. In this table, we show the fraction of samples that perform certain high-level activity. We also provide the behavioral information with respect to the number of malware families, approximated as clusters of samples that exhibit similar behaviors. We describe the clustering technique in more detail in Chapter 3.It is interesting to observe that the differences are often not very pronounced. One reason is that the clustering process was using a tight threshold. That is, samples are only grouped when they exhibit very similar activity, resulting in a large number of clusters. Another reason is that the activity in Table 2.2 is quite generic, and there is not much difference at this level between individual samples and families. The situation changes when looking at activity at a level where individual resources (such as files, registry keys) are considered. For example, 4.49% of all samples create the file `C:\WINDOWS\system32\urdvxc.exe`, but this is true for only 0.54% of all clusters. This file is created by the well-known, polymorphic `allaple` worm, and many of its instances are grouped in a few clusters. Another example can be seen in Table 2.4 and in Table 2.5. Here, 17.53% of all samples use a specific registry key for making the malware persistent. When looking at the granularity of clusters (families), this number drops to 11.67%. Again, the drop is due to the way in which `allaple` operates. It also demonstrates that using statistics based on malware clusters is more robust when large clusters of polymorphic malware samples are present in the dataset.

### 2.3.1 File system activity

Looking at Table 2.2, we can see that, unsurprisingly, the execution of a large number of malware samples (70.8% of all binaries) lead to changes on the file system. That is, new files are created and existing files are modified.

When analyzing the *created* files in more detail, we observe that they mostly belong to two main groups: One group contains executable files, typically because the malware copies or moves its binary to a known location (such as the Windows system folder). Often, this binary is a new polymorphic variant. In total, 37.2% of all samples create at least one executable file. Interestingly, however, only 23.2% of all samples (or 62% of those that drop an executable) choose the `Windows` directory or one of its sub-folders as the target. A large fraction – 15.1% – create the executable in the user's folder (under `Document`

| Autostart Location | Percentage of Samples |
|---|---|
| HKLM\System\Currentcontrolset\Services\%\Imagepath | 17.53% |
| HKLM\SW\MS\WIN\CV\Run% | 16.00% |
| HKLM\SW\MS\Active Setup\Installed Components% | 2.50% |
| HKLM\SW\MS\WIN\CV\Explorer\Browser Helper Objects% | 1.72% |
| HKLM\SW\MS\WIN\CV\Runonce% | 1.60% |
| HKLM\SW\MS\WIN\CV\Explorer\Shellexecutehooks% | 1.30% |
| HKLM\SW\MS\WIN NT\CV\WIN\Appinit_Dlls | 1.09% |
| HKLM\SW\MS\WIN NT\CV\Winlogon\Notify% | 1.04% |
| HKLM\SW\MS\WIN\CV\Policies\Explorer\Run% | 0.67% |
| C:\Documents and Settings\%\Start Menu\Programs\Startup\% | 0.20% |
| Abbreviations | |
| HKLM=HKEY_LOCAL_MACHINE<br>SW=Software<br>MS=Microsoft<br>WIN=Windows<br>CV=Currentversion | |

Table 2.4: Top 10 Autostart locations in percentage of samples.

| Autostart Location | Percentage of Clusters |
|---|---|
| HKLM\SW\MS\WIN\CV\Run% | 17.80% |
| HKLM\System\Currentcontrolset\Services\%\Imagepath | 11.67% |
| HKLM\SW\MS\WIN\CV\Runonce% | 3.07% |
| HKLM\SW\MS\Active Setup\Installed Components% | 2.79% |
| HKLM\SW\MS\WIN\CV\Explorer\Shellexecutehooks% | 2.29 % |
| HKLM\SW\MS\WIN\CV\Explorer\Browser Helper Objects% | 1.75% |
| HKLM\SW\MS\WIN NT\CV\Winlogon\Notify% | 1.89% |
| HKLM\SW\MS\WIN\CV\Policies\Explorer\Run% | 1.04% |
| C:\Documents and Settings\%\Start Menu\Programs\Startup\% | 0.95% |
| HKLM\SW\MS\WIN NT\CV\WIN\Appinit_Dlls | 0.89% |
| Abbreviations | |
| HKLM=HKEY_LOCAL_MACHINE<br>SW=Software<br>MS=Microsoft<br>WIN=Windows<br>CV=Currentversion | |

Table 2.5: Top 10 Autostart locations in percentage of clusters.

`and Settings`). This is interesting, and might indicate that, increasingly, malware is developed to run successfully with the permissions of a normal user (and hence, cannot modify the system folder).

The second group of files contains non-executables, and 63.8% of all samples are responsible for creating at least one. This group contains a diverse mix of temporary data files, necessary libraries (DLLs), and batch scripts. Most of the files are either in the Windows directory (53% of all samples) or in the user folder (61.3%[1]). One aspect that stands out is the significant amount of temporary Internet files created by Internet Explorer (in fact, the execution of 21.3% of the samples resulted in at least one of such files). These files are created when Internet Explorer (or, more precisely, functions exported by `iertutil.dll`) are used to download content from the Internet. This is frequently used by malware to load additional components. Most of the DLLs are dropped into the Windows system folder.

The *modifications* to existing files are less interesting. An overwhelming majority of this activity is due to Windows recording events in the system audit file `system32\config\SysEvent.Evt`. In a small number of cases, the malware programs infect utilities in the system folder or well-known programs (such as Internet Explorer or the Windows media player).

In the next step, we examined the deleted files in more detail. We found that most delete operations target (temporary) files that the malware code has created previously. Hence, we explicitly checked for delete operations that target log files and Windows event audit files. Interestingly, Windows malware does not typically attempt to clear any records of its activity from log data (maybe assuming that users will not check these logs). More precisely, we find that 0.26% of the samples delete a `*log` file, and only 0.0018% target `*evt` files.

We also checked for specific files or file types that malware programs might look for on an infected machine. To this end, we analyzed the file parameter to the `NtQueryDirectoryFile` system call, which allows a user (or program) to specify file masks. We found a number of interesting patterns. For example, a few hundred samples queried for files with the ending `.pbk`. These files store the dial-up phone books and are typically accessed by dialers. Another group of samples checked for files ending with `.pbx`, which are Outlook Express message folder.

[1]Note that the numbers exceed 100% as a sample can create multiple files in different locations.

### 2.3.2 Registry activity

A significant number of samples (62.7%) create registry entries. In most cases (37.7 % of those samples), the registry entries are related to control settings for the network adapter. Another large fraction – 22.7% of the samples – creates a registry key that is related to the unique identifiers (CLSIDs) of COM objects that are registered with Windows. These entries are also benign. But since some malware programs do not change the CLSIDs of their components, these IDs are frequently used to detect the presence of certain malware families. We did also find two malicious behaviors that are related to the creation of registry entries. More precisely, a fraction (1.59%) of malware programs creates an entry under the key `SystemCertificates\TrustedPublisher\Certificates`. Here, the malware installs its own certificate as trusted. Another group of samples (1.01 %) created the `Windows\CurrentVersion\Policies\System` key, which prevents users from invoking the task manager.

We also examined the registry entries that malware programs typically modify. Here, one of the most-commonly-observed malicious behavior is the disabling of the Windows firewall – in total, 33.7% of all samples, or almost half of those that modify Windows keys, perform this action. Also, 8.97% of the binaries tamper with the Windows security settings (more precisely, the `MSWindows\Security` key). Another important set of registry keys is related to the programs that are automatically launched at startup. This allows the malware to survive a reboot. We found that 35.8% of all samples modify registry keys to get launched at startup. We list that Top 10 Autostart locations in Table 2.4 and Table 2.5. As can be seen, the most common keys used for that purpose are `Currentversion\Run` with 16.0% of all samples and `Services\Imagepath` with 17.53%. The `Services` registry key contains all configuration information related to Windows services. Programs that explicitly create a Windows service via the Windows API implicitly also modify the registry entries under this key.

### 2.3.3 Network activity

Table 2.6 provides an overview of the network activities that we observed during analysis. Figure 2.4 depicts the use of the HTTP, IRC, and SMTP protocols by individual samples over a one and a half year period. In contrast, Figure 2.5 shows the usage of the HTTP, IRC, and SMTP protocols once families of malware samples are clustered together (using our clustering approach presented in Chapter 3). These two graphs clearly demonstrate the usefulness of clustering in certain cases. That is, when the first graph is observed, one would tend to think that there is an increase in the number of samples that

| Observed Behavior | Percentage of Samples | Percentage of Clusters |
|---|---|---|
| Listen on a port: | 1.88% | 4.39% |
| TCP traffic: | 45.74% | 41.84% |
| UDP traffic: | 27.34 % | 25.40% |
| DNS requests: | 24.53% | 28.42% |
| ICMP-traffic: | 7.58% | 8.19% |
| HTTP-traffic: | 20.75% | 16.28% |
| IRC-traffic: | 1.72% | 4.37% |
| SMTP-traffic: | 0.89% | 1.57% |
| SSL: | 0.23% | 0.18% |
| Address scan: | 19.08% | 16.32% |
| Port scan: | 0.01% | 0.15% |

Table 2.6: Overview of network activities.

use the HTTP protocol. However, after the samples are clustered, one realizes that the use of the HTTP protocol remains more or less constant. Hence, the belief that there is an increase in HTTP usage is not justified, and is probably caused by an increase in the number of polymorphic samples. However, the graph in Figure 2.5 supports the assumption that IRC is becoming less popular.

Moreover, we observed that 796 (i.e., 0.23%) of the samples used SSL to protect the communication. Almost all use of SSL was associated to HTTPS connections. However, 8 samples adopted SSL to encrypt traffic targeting the non-standard SSL port (443). Interestingly, most of the time the client attempted to initiate an SSL connection, it could not finish the handshake.

In the samples that we analyzed, only half of the samples (47.3%) that show some network activity also query the DNS server to resolve a domain name. These queries were successful most of the time. However, in 9.2% of the cases, no result was returned. Also, 19% of the samples that we observed engaged in scanning activity. These scans were mostly initiated by worms that scan specific Windows ports (e.g., 139, 445) or ports related to backdoors (e.g., 9988 – Trojan Dropper Agent). Finally, 8.9% of the samples connected to a remote site to download another executable. Figure 2.6 shows the file sizes of these second stage malware programs, compared with the size of the executable samples submitted to Anubis. As one may expect, the second stage executables are in average smaller than the first stage malware.

Note that over 70% of the samples that downloaded an executable actually

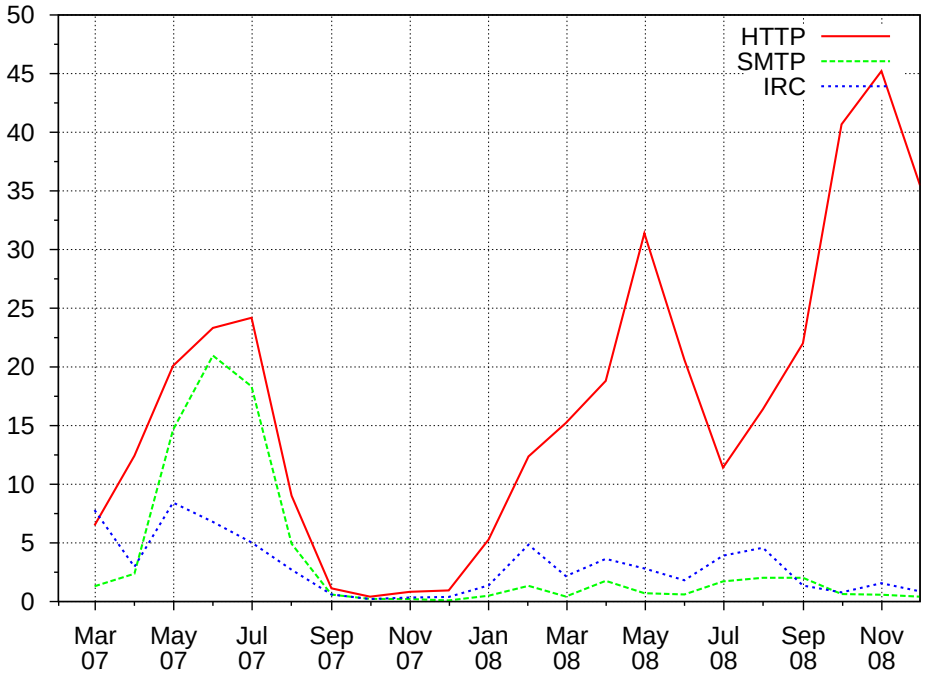


Figure 2.4: Network protocols (by samples).

downloaded more than one. In fact, we observed one sample that downloaded the same file 178 times during the analysis time of four minutes (i.e., the download was corrupted with each download, so the sample immediately attempted another download).

### 2.3.4 GUI windows

Table 2.2 shows that a surprising fraction of samples (33.26%) display a GUI window. Closer analysis reveals that only a small set (2.2%) is due to program crashes. The largest fraction (4.47%) is due to GUI windows that come without the usual window title and contain no window text. Although we were able to extract window titles or window text in the remaining cases, it is difficult to discover similarities. Window names and texts are quite diverse, as a manual analysis of several dozens of reports confirmed. The majority of GUI windows are in fact simple message boxes, often pretending to convey an error of some kind. We believe that their main purpose lies in minimizing user suspicion. An error message draws less attention than a file that does not react at all when being double clicked. For example, 1.7% of the samples show a fabricated message box that claims that a required DLL was not found.

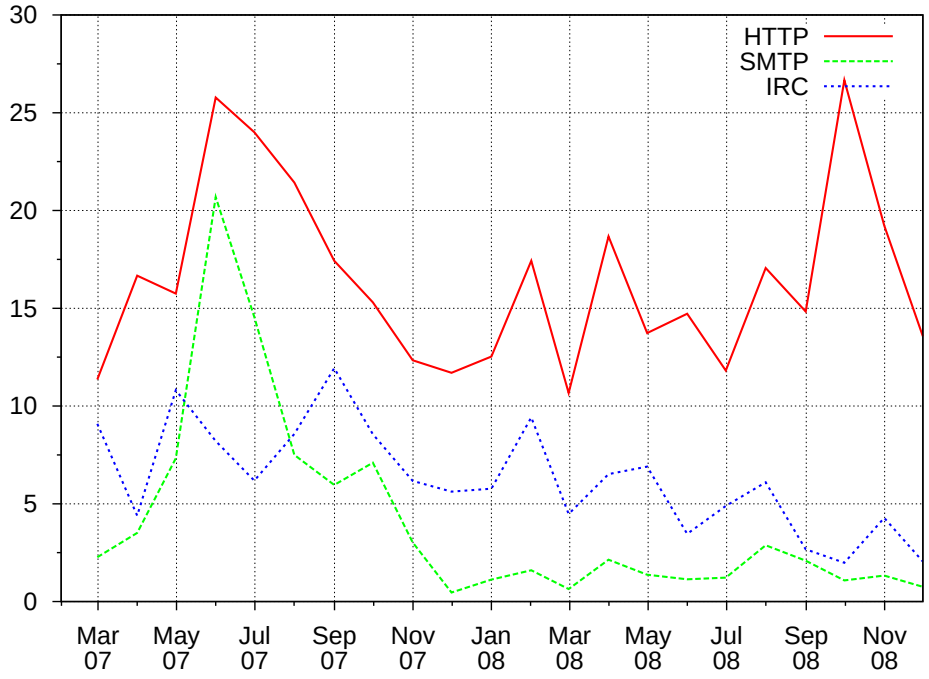


Figure 2.5: Network protocols (by families/clusters).

However, if this error message was authentic, it would be created on behalf of the `csrss.exe` process.

### 2.3.5 Botnet activity

Although a relative recent phenomenon, botnets have quickly become one of the most significant threats to the security of the Internet. Recent research efforts have led to mechanisms to detect and disrupt botnets [44]. To determine how prevalent bots are among our submissions, we analyzed the network traffic dumps that Anubis has recorded. For this, we were interested in detecting three bot families: IRC, HTTP, and P2P.

The first step in identifying a bot based on an analysis report is to determine the network protocol that is being used. Of course, the protocol detection needs to be done in a port-independent fashion, as a bot often communicates over a non-standard port. To this end, we implemented detectors for IRC, HTTP, and the following P2P protocols: BitTorrent, DirectConnect, EDonkey, EmuleExtension, FastTrack, Gnutella, and Overnet.

In the next step, we need to define traffic profiles that capture expected, bot-like behaviors. Such profiles are based on the observation that bots are

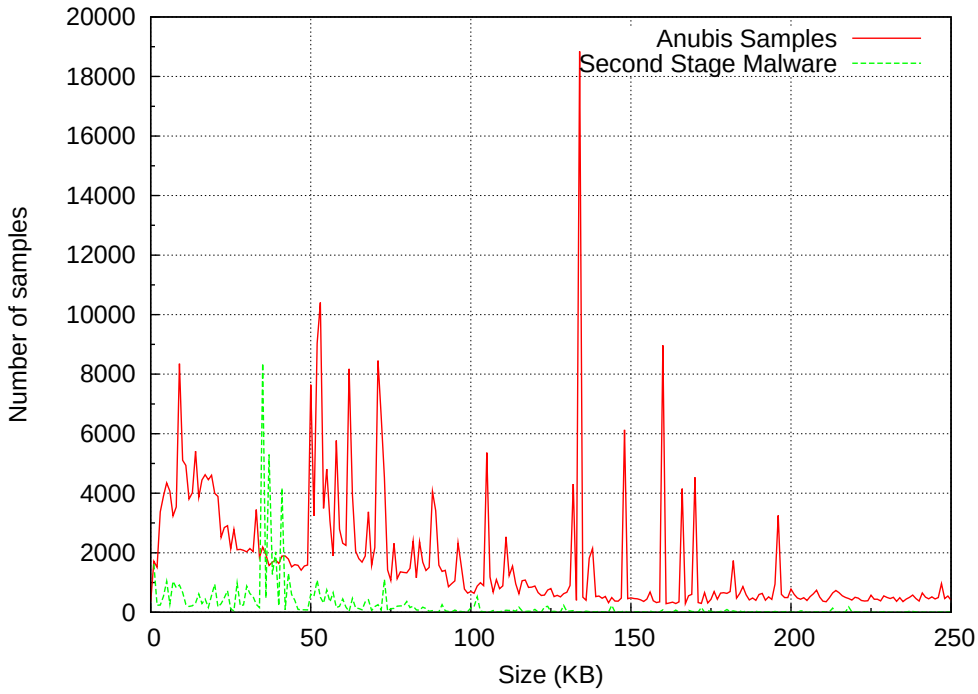


Figure 2.6: Sample sizes.

usually used to perform distributed denial-of-service (DDoS) attacks, send out many spam e-mails, or download malicious executables. Hence, if we see signs for any such known activity in a report (e.g., address scans, port scans, DNS MX queries, a high number of SMTP connections, etc.), we consider this sample a bot candidate. In addition, we use some heuristics to detect known malicious bot conversations such as typical NICKNAME, PRIVMSG, and TOPIC patterns used in IRC communication, or common HTTP bot patterns used in URL requests. The bot analysis is also used to create a blacklist of identified command and control servers. This blacklist is constantly updated and is also used to identify and verify new bot samples.

Our analysis identified 36,500 samples (i.e., 5.8%) as being bots (i.e., 30,059 IRC bots, 4,722 HTTP bots, and 1,719 P2P bots). Out of the identified samples, 97.5% were later correctly recognized by at least two anti-virus as malware. However, it was often the case that anti-virus programs misclassified the sample, e.g. by flagging a storm worm variation as an HTTP Trojan. Also, all P2P bots we detected were variations of the Storm worm.

Figure 2.7 and 2.8 show the bot submission (grouped by type) based on unique samples and unique clusters, respectively. By comparing the IRC botnet submissions in the two graphs, we can observe that, in 2007, most of IRC

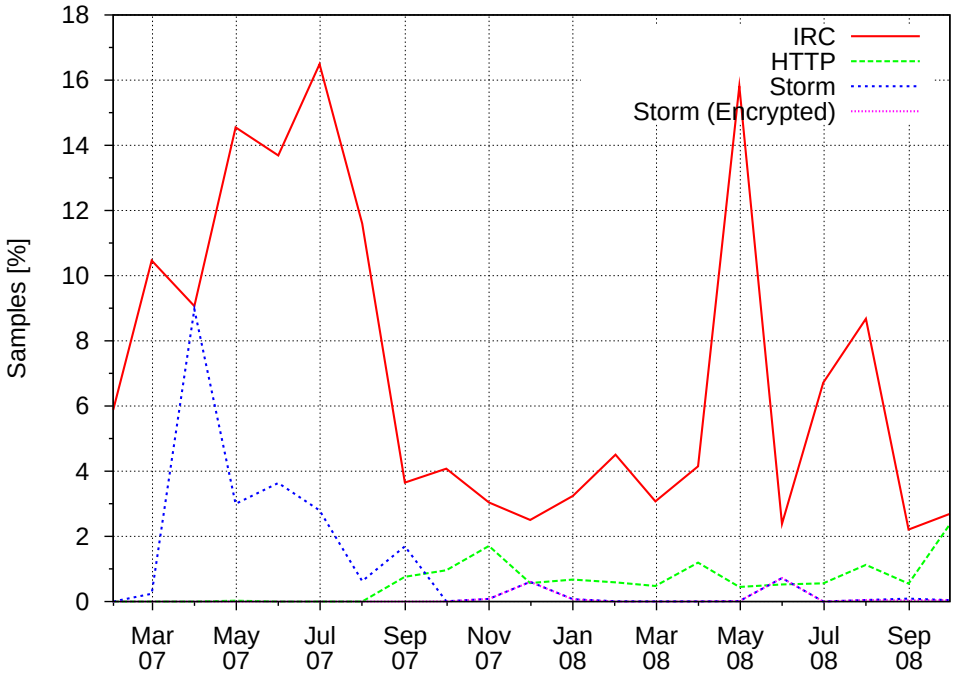


Figure 2.7: Botnet submissions (by samples).

botnets were belonging to different clusters. In 2008 instead, we still received an high number of IRC bots, but they were mostly polymorphic variations of the same family. As an example, the peak that we observed in May 2008 is due to a large number of polymorphic variations of W32.Virut.

Interestingly, we are able to identify samples that, months after their first appearance, are still not recognized by any anti-virus software. This is probably due to the polymorphism and metamorphism techniques used in the malware code. We also verified how many samples were identified by one anti-virus vendor as being a bot and cross-checked these samples with our detection technique. We missed 105 samples that the anti-virus software was able to detect. One reason for this could be the four-minute maximum runtime limit for the samples emulated in the Anubis system.

The Storm worm began infecting thousands of computers in Europe and the United States on Friday, January 19, 2007. However, Anubis received the first storm collection (96 samples) in April 2007. Note that most of the submitted samples of Storm after October 1st are dominated by variants with the encryption capability (i.e., 93%). We obtained the first sample using encrypted communication in October 2007.

When IRC bots are analyzed in more detail, one observes that the channel

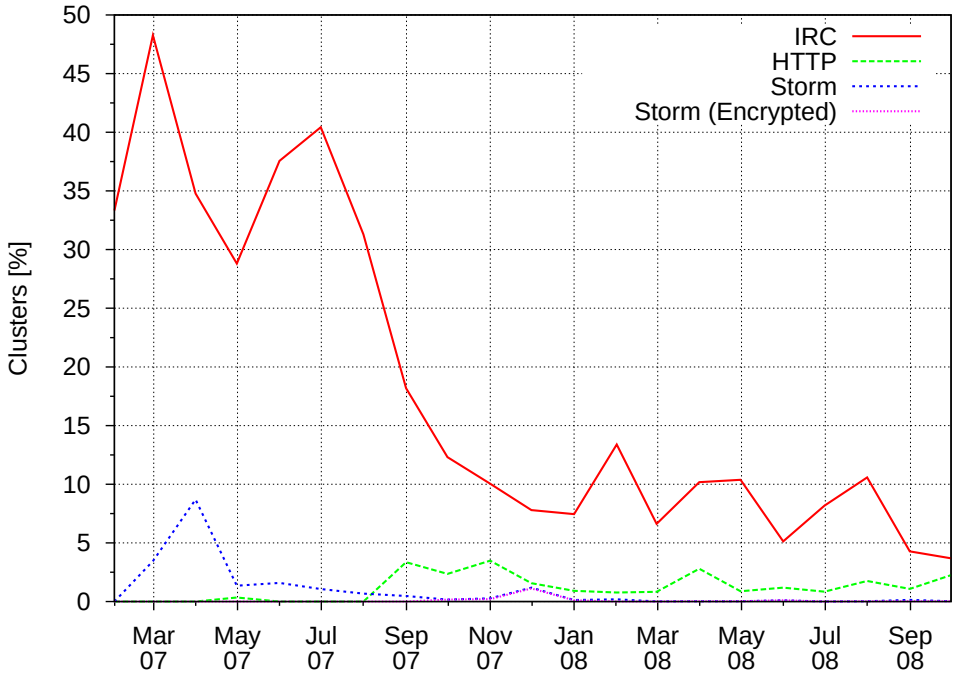


Figure 2.8: Botnet submissions (by families/clusters).

topic is base64-encoded 13% of the time. During the time in which the samples were executed in Anubis, we also collected over 13,000 real commands that the bot master sent to malware under analysis. In 88% of the cases, the commands were instructing the client to download some file (e.g., `get` and `download` commands). Some other interesting commands that we observed were `ipscan`, `login`, `keylog`, `scan`, `msn`, and `visit`.

We also analyzed how many samples tried to disguise their activities by using standard protocols on non-standard ports. For the HTTP bots, 99.5% of the samples connected to the ports 80 and 8080. Only 62 samples were using non-standard ports. However, for the IRC bots, the picture is quite different. 92% of the samples were connecting to an IRC server running on a non-standard port. For example, the ports 80 and 1863 (i.e., Microsoft Messenger) are very common alternatives, often used to bypass firewalls.

Finally, we can classify the 1,719 Storm samples that have been submitted to Anubis into two classes: variants that use encrypted communication channels, and those that do not support encryption. As far as the decryption key is concerned, we only observe one symmetric key consistently being used to encrypt Storm traffic.

| Observed Comparison with | Number of Samples | Number of Clusters |
|---|---|---|
| Windows Product Id of Anubis: | 55 | 28 |
| Windows Product Id of CWSandbox: | 32 | 14 |
| Windows Product Id of Joebox: | 32 | 14 |
| Executable name of `sample.exe`: | 35 | 17 |
| Computer name of Anubis: | 4 | 4 |
| Qemu's HD name: | 2 | 2 |
| VMWare's HD name: | 1 | 1 |
| Windows user name of 'user': | 2 | 2 |
| Any Anti-Anubis comparison: | 99 | 54 |
| Any Anti-Sandbox comparison: | 100 | 55 |

Table 2.7: Overview of observed comparisons.

### 2.3.6 Sandbox detection

Another interesting aspect of malware behavior is its capability to detect the presence of an analysis environment such as Anubis. Dynamic analysis systems are a popular means to gather data about malicious code, and it is not surprising that malware is using techniques to thwart such analysis. When a malware program detects a sandbox, it typically alters its behavior - most of the time, it just quits. In this section, we attempt to estimate the number of samples that use (general and Anubis specific) anti-sandbox techniques.

Sandbox detection techniques fall into two main classes: One class is comprised of *instruction-level* detection techniques, which are techniques that determine the difference between a real CPU and an emulated one by only making use of CPU instructions. The second class are *API-level detection* methods, which query the environment by calling one or several (Windows) API functions.

Unfortunately, we have currently no good way to detect *instruction-level* detection attempts. While it is theoretically possible to monitor single instructions in a full system emulator, for performance reasons, Anubis essentially works at the API (and system call) level. Since Anubis leverages Qemu for its full system emulation, it is susceptible to the same detection methods as Qemu is. In particular, several of the detection strategies (all of them are instruction-level detections) outlined in [42] and [64] also work for Anubis.

Since we do monitor Windows API calls, we are in a better position to check for possible *API-level detection* methods. At the time of writing this thesis, several Anubis-specific detections have been published [11] on the Internet.

All of them work by comparing the return value of a Windows API function such as `GetComputerName` to a a hard-coded value that is known to identify Anubis. Our system includes a data tainting component that allows us to detect when the output of an interesting function is subsequently used as an operand in a comparison operation [25]. Thus, we were able to analyze our sample set for specific comparisons that represent known Anubis checks published in forum postings or blogs. Table 2.7 summarizes our findings. A couple of checks deserve more discussion. Our system names each analyzed sample `sample.exe`. Thus, a well-known Anubis detection routine compares the return value of the Windows API function `GetCommandLine` with that string. Similarly, in the past, every sample was executed as a Windows user with the name "user." Consequently, it was possible to compare the result of `NtQuerySystemInformation` with the string "user" to detect Anubis (as a side note, we have since changed this username to "Administrator").

In total, we have found that only 0.03% of the samples (99 distinct malware programs) contain a known Anubis check. Most of these samples perform the check in the beginning of their execution, and immediately exit if they detect the presence of our sandbox. Of course, it is possible that malware samples perform a wide range of other checks. Moreover, as mentioned previously, we have no good way to identify instruction-level detections or timing-based detection attempts. Thus, the reported numbers are a clear lower bound.

Under the assumption that a sample that detects Anubis (or a sandbox) does not perform much activity, we can also provide an upper bound for the samples that do sandbox detection. Based on our experience with Anubis, we consider a behavioral report (a profile [25]) to contain "not much activity" when it contains less than 150 features. For comparison, the average profile has 1,465 features. Using this definition, we found that 12.45 % of the executable samples (13.57 % of the clusters) show not much activity.

Of course, not of all these samples really contain anti-sandbox routines, as there are multiple reasons why Anubis might not be able to produce a good report. For example, GUI programs that require user input (such as installers) cannot be analyzed sufficiently. Anubis only has a very limited user input simulation, which simply closes opened windows. Moreover, some programs require non-existing components at runtime (note, though, that programs that fail because of unsatisfied, static DLL dependencies are not included in the 12.45 %). In addition, at least 0.51% of the reports with not much activity can be attributed to samples that are protected with a packer that is known to be not correctly emulated in Qemu (such as Telock and specific packer versions of Armadillo and PE Compact). Last but not least, bugs in Anubis and Qemu are also a possible cause.

## 2.4 Conclusion

Malware is one of the most important problems on the Internet today. Although much research has been conducted on many aspects of malicious code, little has been reported in literature on the (host-based) activity of malicious programs once they have infected a machine.

In this chapter, we aim to shed light on common malware behaviors. We perform a comprehensive analysis of almost one million malware samples and determine the influence of code polymorphism on malware statistics. Understanding common malware behaviors is important to enable the development of effective malware countermeasures and mitigation techniques.

We have seen in this chapter that it is possible and very useful to use the large number of analysis results at our disposal to create a high level view of common malicious behavior. In the next chapter, we will discuss a complementary approach for dealing with such a huge number of malware samples and analysis reports. We present a system for grouping analysis reports of malware according to the similarity of their behavior.

# Chapter 3

# Scalable, Behavior-Based Malware Clustering

## 3.1 Introduction

In this chapter, we present a novel clustering technique that scales well and produces more precise results than previous approaches. Our technique is based on a dynamic analysis system that monitors the execution of a malware sample in a controlled environment. Unlike many previous systems that operate directly on low-level data such as system call traces, we enrich and generalize the collected data and summarize the behavior of a malware sample in a behavioral profile. These profiles express malware behavior in terms of operating system (OS) objects and OS operations. Moreover, profiles capture a more detailed view of network activity and the ways in which a malware program uses input from the environment. This allows our system to recognize similar behaviors among samples whose low-level traces appear very different. Finally, we cluster the analyzed samples according to their behavioral profile. We employ a scalable clustering algorithm that avoids calculating $n^2$ distances between all pairs of $n$ samples, and thus, is suitable for clustering large, real-world malware collections. To summarize, the contributions of this solution are as follows:

- We present a novel, precise approach to capture a malware program's behavior. To this end, we monitor the execution of a program and create its behavioral profile by abstracting system calls, their dependences, and the network activities to a generalized representation consisting of OS objects and OS operations.

- We present an efficient and fast algorithm for clustering large sets of malware samples that avoids calculating $n^2$ distances between all pairs of $n$ samples, and thus, is suitable for clustering large, real-world malware collections.

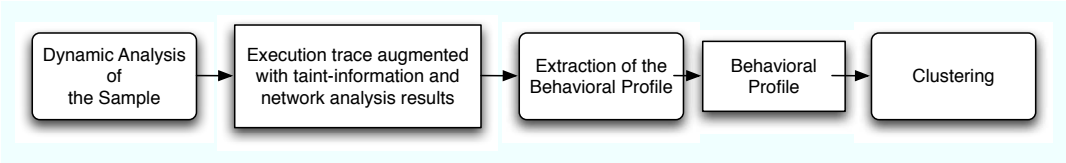


Figure 3.1: System overview.

- We have evaluated our system on large, real-world data sets. Our experiments demonstrate that our technique achieves more precise clustering results than previous approaches and scales to tens of thousands of malware samples.

## 3.2 System Overview

The goal of our system is to cluster large collections of malware-samples based on their behavior. That is, we want to find a partitioning of a given set of malware programs so that subsets share some common traits. As illustrated in Figure 3.1, clustering malware samples is a multi-step process. It consists of an initial, dynamic malware analysis phase, a subsequent extraction of behavioral profiles, and a final clustering phase.

**Dynamic Analysis.** The first step in the clustering process is the automated analysis of malware samples. For this purpose, we have extended ANUBIS, our system for automated, dynamic malware analysis [28]. This system is based on Qemu [29], a whole-system emulator for PCs and the Intel x86 architecture. The analysis system works by executing binaries in the emulated environment, producing a trace of the system calls that this binary invokes.

We first extended ANUBIS with taint tracking. Similar to tainting systems in previous work [39, 60, 63], we attach (taint) labels to certain interesting bytes in memory and propagate these labels whenever they are copied or otherwise manipulated. Our taint tracking system builds upon the taint tracking implementation used in a previous prototype [60]. While the propagation of taint labels works the same as in [60], the tainting systems differ in their use of taint sources. In our system, system calls serve as taint sources. More precisely, we taint the out-arguments and return values of all system calls. At the same time, we check whether any in-argument of a system call is tainted. The goal is to identify how the program uses information that it obtains from the operating system.

While the idea of taint tracking itself is not new, we leverage this infor-

mation to obtain a number of novel, important features that better capture the behavior of a malware program. For example, we can observe when a program uses the return value of the `GetDate` system call in a subsequent `CreateFile` call. This allows us to determine that a file name depends on the current date and changes with every malware execution. As a result, the filename is generalized appropriately. Furthermore, we taint the entire code of the executable. This allows us to uncover cases in which a program reads its own code segment. This is helpful to detect important propagation patterns, such as a worm sending itself over the network or a Trojan horse copying itself into the Windows system directory. Finally, we record program control flow decisions that are based on tainted data. This allows us to identify similarities between programs that perform the same date checks or that attempt to shut down the same anti-virus software.

To address the problem that a program's network activity is not sufficiently captured by system call traces, we have built a network analysis component that operates on the network traffic itself. The problem is that on the system-call level, all network activities are performed through calls to a single, native API function called `NtDeviceIoControlFile` - only differing in their arguments. Ideally, we would like to know what emails are sent, what HTTP downloads are performed, what IRC conversations take place, etc. To this end, our network analysis component leverages Bro [65] and makes use of its capabilities to recognize and parse application-level protocols (such as HTTP, SMTP, and IRC).

The output of the analysis step is an execution trace that is augmented with taint information. This trace lists all system calls together with their argument values. Moreover, it provides taint information for each argument. This taint information allows us to connect the return values (and out-arguments) of one system call with the in-arguments of subsequent calls.

**Behavioral Profile.** In this step, we process the execution traces provided by the previous step. More precisely, for each sample, we extract a behavioral profile that accurately describes the runtime activity of the binary and serves as input to our clustering algorithm.

Unlike existing systems [43, 52], our clustering algorithm does not operate directly on system calls. The reason is that system call traces can vary significantly, even between programs that exhibit the same behavior. For example, consider the different ways to read from a file: Program A might read 256 bytes at once, while program B calls `read` 256 times, reading 1 byte with each call. Moreover, it is easily possible to interleave the read calls with other, independent system calls so that the system call trace changes. For this reason, we abstract system call traces into a set of operating system objects, together

with a set of operations (such as read, write, create) that were performed on these objects.

An OS object represents a resource, such as a file or registry key, that can be manipulated and queried via system calls. For example, our behavioral profile might include the file object `C:\Windows` and its accompanying operation `query_directory`. An OS operation is a generalization of a system call that unifies different system calls with similar semantics but different function signatures (e.g., the system calls `NtCreateProcessEx` and `NtCreateProcess` both map to the same operation).

Based on the information that the tainting system provides, we infer dependences between OS objects. Copying a file, for example, is represented as a dependence between the source file OS object and the destination file object. Dependency information implicitly captures the order of certain operations. This is important, because we do not explicitly consider the order of OS operations that are performed on a specific OS object. The reason is that a behavior profile should not rely on the order in which unrelated operations are executed. Moreover, dependences help to determine resource names that are derived from data sources whose values change between execution traces (such as random values or the current time). This information allows us to generalize the corresponding OS object names.

The output of this step is an abstraction of the program's execution trace that contains information about the OS objects that the program operates on, as well as of the type of operations and dependences. These abstractions are called a *behavioral profile*.

**Scalable Clustering.** In this step, we cluster a set of behavioral profiles such that samples that exhibit similar behavior are combined in the same cluster. Given the rapidly increasing number of malware programs, it is clear that one of the most important requirements for a clustering algorithm is scalability. It must be possible to cluster a large amount of malware, such as a hundred thousand samples, in a reasonable time. Most clustering methods require the computation of the distances between all pairs of points, which invariably results in a computational complexity of $O(n^2)$. This might lead to systems that take three hours to process 400 samples [52].

In this chapter, we propose to efficiently solve the clustering problem using an approximate, probabilistic approach. Our clustering algorithm is based on locality sensitive hashing (LSH), which was introduced by Indyk and Motwani [48]. LSH provides an efficient (sublinear) solution to the approximate nearest neighbor problem ($\epsilon$-NNS). Clustering is one of the main applications of this technique: LSH can be used to perform an approximate clustering while computing only a small fraction of the $n^2/2$ distances between pairs of points.

Leveraging LSH clustering, we are able to compute an approximate, single-linkage hierarchical clustering for a data set of more than 75,000 samples in less than three hours.

## 3.3 Dynamic Analysis

Dynamic malware analysis systems have become increasingly popular because they deliver good analysis results even in case of obfuscated or self-modifying code and analysis resistance techniques [55, 61, 72]. Since meaningful analysis results are a prerequisite for good clustering results, we have chosen to further extend ANUBIS, our existing, dynamic analysis system [28]. More precisely, we have added support to track dependences between operations on system code objects, as well as support to analyze control flow decisions that involve tainted data operands, and we have improved the network analysis.

### 3.3.1 System Call Dependences

Data tainting is a well-known technique for tracking information flows in a whole-system emulator. In this work, we are leveraging the tainting approach to capture the dependences between system calls. As noted by Christodorescu et al. [36], system call dependences provide valuable insights into the behavior of an application. For example, knowledge about dependences allows one to see when a program searches for files with a specific filename pattern and then opens all files that were found. In our analysis, system call dependences are used in the following contexts:

- *Generalization of execution traces*: An execution trace inherently includes many execution-specific events and names (filenames, host names). These execution-specific tokens change every time the binary is executed. Based on dependence analysis, such execution-specific artifacts can be recognized. For example, knowing that a filename depends on the current time helps to remove the filename as a characteristic for a program's behavior. Also, tainting the return value of the Windows function `GetTempFileName` puts us into a position where we can identify temporary file names.

- *Copy Operations*: Tainting allows us to recognize data movements, such as the case when data is copied. This allows us to determine the malware's propagation vector. For example, we see when the malware copies itself to the Windows system directory or sends a copy of its code over the network.

**Taint Sources.** Although our behavioral profile is primarily based on system calls and their dependences, we are not focusing on the native API interface[1] alone. Instead, we also include several Windows API functions. This is different from previous systems, which either operate on the system call interface [41, 63] or perform whole-system taint analysis [74]. In the latter case, taint sources are typically devices such as a network card or the keyboard.

The Windows API is a large collection of user mode library routines, which in turn invoke native API functions when necessary. Considering the Windows API is important for several reasons: First, some functionality is managed and provided exclusively by user-mode portions of the operating system. That is, no calls to native API functions are performed. Among other things, this is true for the random number generator, the path-related Windows API functions (e.g. `GetTempFileName`, `GetTempPath`), and DLL-management functions (e.g., `GetProcAddress`). Second, there are Windows API functions that have semantically-equivalent native API functions, but, because of performance reasons, have been implemented in a way that does not require invoking the appropriate system service. An important instance are the time-related Windows API function, such as `GetTickCount` or `GetSystemTime`. These functions do not invoke a system call but work by reading from a special page in the virtual address space. This page is always mapped read-only to a fixed address in the user-mode, virtual address space of a processes. The kernel maps the same page with write access and updates the time-related information in the timer interrupt handler.

**Memory-Mapped Files.** Memory-mapped files, officially termed section objects in Windows NT, pose a special challenge to the analysis system. When a process maps a file into its virtual address space, reading and writing to the file is possible by simply reading and writing to the mapped memory region. These read and write operations do not result in any system calls. Thus, in current analysis systems such as [1, 5, 16, 19], all read and write activity to a memory-mapped file will go unnoticed. However, it is crucial to add support for Windows section objects to obtain a complete view of the operations of a program.

To keep track of indirect write operations to a file, we modified the function that is responsible for writing to the physical memory in our emulation. Whenever the process writes to memory, we check whether the address is in a memory-mapped area. If this is the case, we report this operation as a write to the corresponding, mapped file. Tracking read operations from a memory-mapped file requires tainting the appropriate region in memory whenever a program maps a file into its address space. However, Windows does not load

[1] In Windows NT, the operating system call interface is termed native API.

the contents of the file into physical memory at the time of the section creation. Instead, Windows defers loading (portions of) the file into the physical memory until the time when a virtual address in this region is accessed. Because our tainting system is only able to taint values in the physical memory and the CPU registers, we have to wait until the file is eventually mapped into physical memory before we can taint it. We solve this problem by monitoring all invocations of the page fault handler. When the page fault handler brings in a page that is part of a memory-mapped region, we taint the page after the handler returns.

### 3.3.2 Control Flow Dependences

Taint information is useful to track dependences between system calls. However, it is also interesting to analyze how tainted data is used by the program itself. More specifically, we would like to identify the control flow decisions that involve data that a process has obtained via system calls. Information about such control flow decisions reveals many interesting aspects about a program. For example, it allows us to discover which processes a malware sample potentially wishes to terminate by observing all comparisons that take place as the program iterates over the list of running processes. Since the list of running processes has to be retrieved by means of system calls, the process names that this system call returns are tainted. Hence, we are aware of all comparisons that involve the retrieved process list as argument.

On the x86 architecture, many different assembler instructions for comparing two values exist. Fortunately, inside the Qemu intermediate language, all of these different compare instructions (such as `CMP`, `CMPS`, `SCAS`) map to the same intermediate language construct. Thus, we can easily handle all compare instructions by building our analysis on top of Qemu's intermediate language. The REP instruction prefix is correctly handled as a consecutive execution of the same compare instruction. However, as explained in the following paragraphs, consecutive compare instructions are merged into a single one during our analysis.

To detect comparisons with tainted values, the following extensions were necessary:

**Representation.** In the Qemu intermediate language a compare instruction has two operands with a size of either one, two, or four bytes. For each comparison, we can examine the taint labels that are attached to the bytes of both arguments (if there are any). When a taint label is present, we can determine the system call and the exact argument where the corresponding data byte was retrieved from. Based on this information, we can also determine the

original data type of a tainted byte. This is possible because the Windows native API header files declare all system calls and the types of their arguments. Based on the data type, we consider the operand of a comparison as signed/unsigned integer or as a character string. Knowing the data type also allows us to pinpoint the exact member of a structure. For example, we do not only see that a value '6' is compared with `struct_SYSTEMTIME`, but we can also determine that the value is compared to the struct's `wDay` member.

One problem arises when more complex data structures (e.g., structs, strings, etc.) are involved in a comparison. In this case, we observe several, consecutive `cmp` instructions that operate on a few bytes of the data structure. To handle such cases, consecutive compares on successive labels are merged. Also, when comparing for equality, the comparison terminates as soon as the first differing byte is encountered. In these cases, we cannot see the complete values that the program actually compares. However, the complete data structure exists in the computer's main memory. Thus, when string comparisons are involved, we try to recover the entire string by reading it from the main memory. To this end, we assume that the operand being compared marks the beginning of the string and check until a null byte is found.

There are two types of comparisons that we record as part of an execution trace: A comparison of a labeled value (i.e., at least a single byte of an operand is tainted) with an unlabeled value (called a *label-value comparison*) and a comparison of a labeled value with another labeled value (called a *label-label comparison*). In both cases, we do not output the concrete values of the labeled (tainted) data but the source where this data originates from. More precisely, for tainted data, we record the function name, the function argument, and, if applicable, the name of the structure that holds the data together with the member name. This allows us to identify which inputs to the program are used for comparisons. In case of a label-value comparison, we also learn the concrete value that the program checks for.

**Filtering.** An important part of analyzing control flow dependences is to filter out the irrelevant ones. Compare instructions occur very frequently, and a raw execution trace typically contains millions of compares with tainted operands. To focus on compare instructions that are done by the actual malware program, we discard those that were executed on behalf of (user-mode) Windows API functions. In this way, we ignore comparisons that do not represent the direct intent of the program's author, but that are present as a result of standard Windows behavior. The only exception to this rule are a number of Windows API functions that are used for comparing more complex data types, such as strings or dates. Obviously, the comparisons that occur inside these API functions are the direct consequence of the programmer's intent.

For this reason, we catch all comparisons that take place inside `strcmp`, for example.

### 3.3.3 Network Analysis

The network activities of a malware sample provide one of the most important and characterizing insights into a sample's behavior. Thus, the analysis of a sample's network activity plays an important role in our approach.

**Environment.** A successful network analysis requires that a sample is able to perform the network activities that it has been programmed to do. Dynamic analysis cannot observe email activity of a program when it fails to establish a TCP connection to a mail server. Thus, as a first step, we run the sample in an environment that permits a sample to perform its built-in network activities.

To create the environment for malware execution, we allow an analyzed sample to download files via HTTP and to contact IRC servers directly on the Internet. All other traffic is rerouted to a specially-prepared server, called the victim machine, which has been configured to accept incoming connections on a number of ports that are frequently used by malware programs. For example, the victim machine runs its own SMTP server that answers all SMTP requests (but does not deliver any emails). Moreover, we have set up nepenthes [22] - a honeypot system that emulates known vulnerabilities of popular services. Of course, we are not using the nepenthes server as a honeypot system in the usual sense, i.e., as a way to gain new malware samples. Instead, we have deployed nepenthes only for having a basic service listening on ports that are frequently used for spreading (such as the Windows Samba ports).

**Analysis.** The goal of our network analysis is to extract high-level semantic operations from the low-level socket system calls. For example, instead of reporting that a TCP connection was established, together with the amount of bytes that were exchanged, we aim to report that an HTTP GET request was sent to download the file "foo.bar." We have chosen to build our analysis on top of the packets that are sent and received at the network level. This is easier and more comprehensive than attempting to infer all information from the arguments of the `DeviceIoControlFile` system call, which serves as a funnel for all network-related activity on Windows. To capture network traffic, we have modified our system emulator's network card to simply dump all packets to a log file in PCAP-format. This way, we have a wide range of standard network analysis tools at our disposal to aid us in our analysis efforts. Also, by parsing network packets and parsing application protocols, such as HTTP, we are able to identify network activity on a higher level of abstraction. We use Bro [65] for our analysis, a system that has built-in

support for identifying and parsing HTTP, IRC, SMTP, and FTP protocols. For these protocols, we extract information such as names of downloaded files, names of IRC channels, or mail subjects.

## 3.4 Behavioral Profile

When the dynamic analysis step finishes processing a sample, the next task is to transform the augmented execution trace into a behavioral profile. As mentioned previously, a behavioral profile captures the operations of a program at a higher level of abstraction. To this end, we model a sample's behavior in the form of OS objects, operations that are carried out on these objects, dependences between OS objects and comparisons between OS objects. More formally, a behavioral profile $P$ is defined as an 8-tuple

$$P = (O, OP, \Gamma, \Delta, CV, CL, \Theta_{CmpValue}, \Theta_{CmpLabel})$$

where $O$ is the set of all OS objects, $OP$ is the set of all OS operations, $\Gamma \subseteq (O \times OP)$ is a relation assigning one or several operations to each object, and $\Delta \subseteq ((O \times OP) \times (O \times OP))$ represents the set of dependences. $CV$ is the set of all compare operations of type label-value, while $CL$ is the set of all compare operation of type label-label. $\Theta_{CmpValue} \subseteq (CV \times O)$ is a relation assigning label-value compare operations to an OS object. $\Theta_{CmpLabel} \subseteq (CL \times O \times O)$ is a relation assigning label-label compare operations to the two appropriate OS objects.

**OS Objects.** An OS object represents a resource, such as a file or registry key, that can be manipulated and queried via system calls. Formally, an OS object is a tuple of the following form:

```
OS Object ::= (type, object-name)
type ::= file|registry|process|job|
         network|thread|section|
         driver|sync|service|random|
         time|info
```

That is, an OS object has a name and a type that together uniquely identify the object in the operating system. The 'file' type covers file, named pipe, and mailslot resources, 'registry' consists of registry keys, 'process' includes processes, and 'job' denotes Windows NT jobs, which allow for combining individual processes into a group. The 'network' category describes network objects, 'thread' represents thread activity, 'section' refers to memory-mapped files, and 'driver' captures the loading and unloading of Windows device drivers.

The type ‘sync‘ abstracts all synchronization activities, such as operations on semaphores and mutexes, and ‘service‘ contains objects that represent Windows services. The type ‘random‘ includes several sources of randomness, each of which can be used by a program to generate a random number. The type ‘time‘ consists of time sources, and ‘info‘ contains only two objects. One is the object *info-executable*, which represents the loaded executable. The other one is *info-general*, which represents information such as pathnames of the windows system directory and the temporary directory.

| *OS Object* | | *OS Operation* | |
|---|---|---|---|
| *Type* | *Name* | *Name* | *Attributes* |
| net | http_server | contact | ‘www.gson.com‘,‘80‘ |
| net | http_request | get | ‘/down/s.htm‘ |
| net | dns_resolver | query | ‘Type A‘,‘mx.gmx.net‘ |
| net | port_listener | listen | ‘TCP‘,‘6777‘ |
| net | smtp_attmts | send | ‘fpw.exe‘ |
| net | smtp_content | send | ‘Test yep.‘ |
| net | smtp_subjs | send | ‘Hi‘ |

Table 3.1: Example network OS objects.

To create OS objects, we search the execution trace for all system calls that produce new OS resources. For example, the function `NtCreateFile` creates new files. For each such system call, we extract the object name from the argument list, deduce the object type from the type of the system call, and then create a new OS object. Typically, native API calls have a parameter, named `ObjectAttributes`, that can be directly translated to an object name. In a few cases, it is more difficult to determine the object name. For example, `NtCreateProcess` expects a handle argument that points to a section object (a memory-mapped file), instead of an argument that specifies the filename of the executable. To address this problem, we have extended our system call logger to resolve handles to NT kernel objects and provide this information.

Since network activities are not directly represented in the execution trace, we rely on the network analysis component for extracting the virtual network OS objects. Depending on the type of network traffic observed, we create different kinds of network objects. Table 3.1 lists some example network objects, together with their corresponding operations.

**OS Operations.** An OS operation is a generalization of a system call. Formally, an operation is defined as:

```
OS operation ::= (operation-name,
```

```
operation-attributes?,
successful?)
```

An operation must have a name, it may have one or more attributes that provide additional information about the operation, and it may have a value describing whether the operation was successful.

We map system calls to OS operations with the intent of abstracting from API-specific details. For example, we ignore whether a process is created by means of `NtCreateProcess` or `NtCreateProcessEx` and unify these two system calls into the single OS operation `create`. Our mapping function only considers the most essential system calls, such as functions for reading, writing, and creating operating system objects. This allows us to abstract from many unimportant details. For example, we ignore all functions relating to NT's Local Procedure Call functionality, because this is an undocumented feature that is not available via the Windows API. Currently, we map 130 native API and Windows API functions to 55 OS operations.

System calls that operate on a resource typically have a (handle) parameter that references the target resource. This is necessary for the OS to know the resource to which an operation should be applied. We make use of these handles to map operations to the appropriate OS objects. There are few cases where a function that logically constitutes an operation on an object does not have a handle parameter that specifies this object. The `NtQueryAttributesFile` function, for example, uses a filename instead of a handle to indicate the file object that it works on. After assigning operations to OS objects, our implementation stores all of an object's operations in a set. As a consequence, the order of OS operations is irrelevant. This is important, because it is very easy to reorder system calls on a resource without changing the semantics of a program. Thus, we are able to generalize our behavioral profile by neglecting the order of operations. System call dependences are used to capture the order between those OS operations where the actual order is implied by a data dependence. Moreover, the number of operations on a certain resource does not matter in our system. This sacrifices some precision, but makes the behavioral profile more general, and thus, harder to evade by introducing superfluous operations.

**Example of a Behavioral Profile.** Figure 3.2 shows an example of a behavioral profile. Note that although this example is shown in C code, our profile extraction algorithm works on execution traces. This example shows code that copies the file `C:\sample.exe` to `C:\Windows\sample.exe` by memory-mapping the source file. As one can see, independent of the number of times the write operation in Line 14 is executed, the write operation appears only once in the corresponding behavioral profile. It is also notewor-

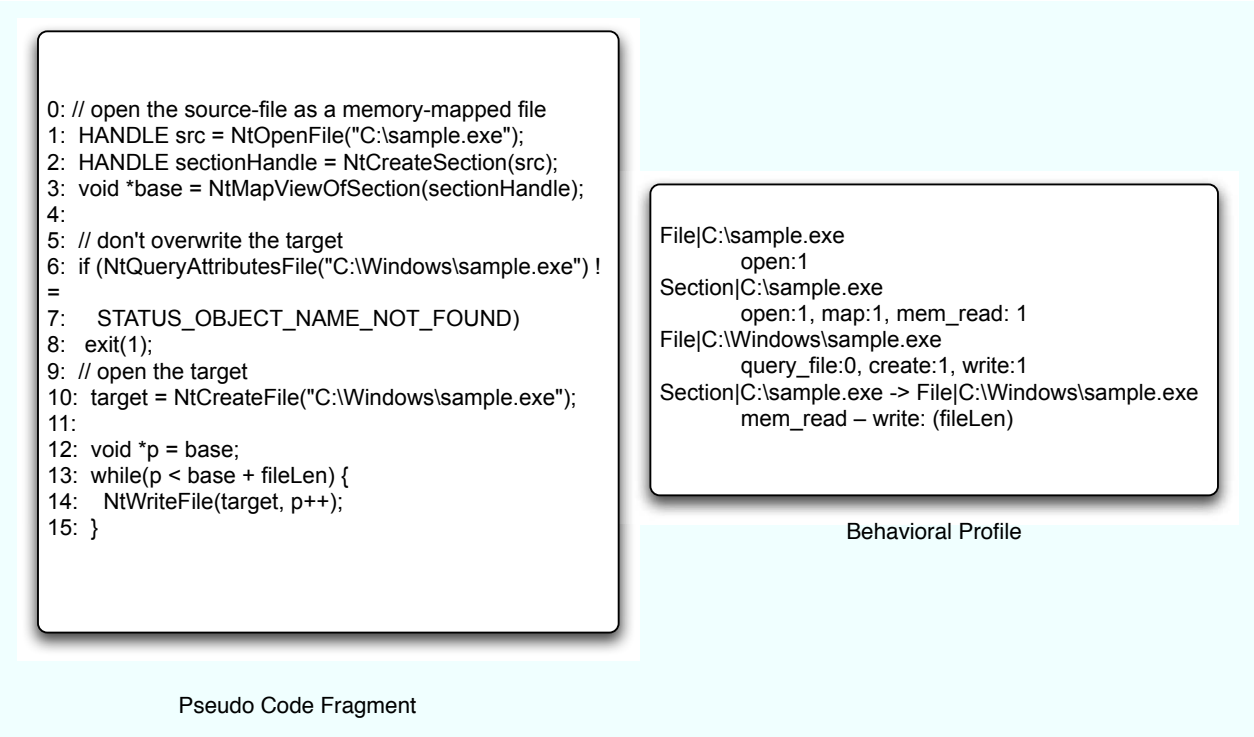


Figure 3.2: Example Behavioral Profile

thy that the `NtQueryAttributesFile` operation in Line 6 is assigned to the object `C:\Windows\sample.exe`, although it does not use a handle argument to reference its OS object.

**Object Dependences.** We abstract dependences between system calls to dependences between OS objects. While a system call dependence is a dependence relation between two system call instances, an OS object dependence is a dependence between two OS objects and their operations. For each existing system call dependence, we first check whether the two involved system calls map to OS operations. If this is the case, we introduce an object dependence between the corresponding OS objects. The behavioral profile shown in Figure 3.2 contains a dependence between the section OS object of the source file and the file object of the destination file. This dependency reflects the fact that data from the source was copied to the destination file.

Due to the fact that all our object dependences originate from system call dependences, we would lack network-related dependences. As explained previously, this is because the extraction of network OS objects is a separate process that is mostly based on the captured network traffic. To address this problem, we have partly reverse-engineered the semantics of the `NtDeviceIoControlFile` function. `NtDeviceIoControlFile` is a universal interface that allows user-mode programs to communicate with device drivers, including the network stack. It is possible to recognize network-related invocations of `NtDeviceIoControlFile` by checking two of its arguments, the handle argument as well as its IO control code. In addition, `NtDeviceIoControlFile` has an input buffer and an output buffer argument for transferring data. For each call to `NtDeviceIoControlFile` that represents network activity, we insert an artificial system call into the execution trace that represents a decoded form of the original call. In particular, we have to decode the buffer arguments. In the case of network activities, `NtDeviceIoControlFile`'s buffer arguments contain pointers to network-specific structs. There are four different artificial system calls:

```
AfdSend(SocketHandle h, char *buffer)
AfdReceive(SocketHandle h, char *buffer)
AfdBind(SocketHandle h, short localPort)
AfdConnect(char *foreignAddress,
           short foreignPort)
```

We insert `AfdSend` when we determine that a process calls `NtDeviceIoControlFile` to send data. Analogously, we insert `AfdReceive` when data is received, `AfdBind` when a socket is bound to a specific port number, and `AfdConnect`, when a TCP connection is established. The arguments of the

four artificial calls reflect the taint information of their corresponding system calls. The SocketHandle parameter allows us to attribute the individual invocations to the appropriate network connection.

Based on our representation of objects and their dependences, it is straightforward to find execution-specific artifacts. For example, we recognize random or temporary filenames by checking whether there is a dependence between a file object and a random source. If this is the case, we do not want to keep the actual object in the profile, since it is different for each execution. Thus, we replace the concrete object name with a placeholder token that indicates the source of the object name (such as TEMPORARY for a temporary filename). Moreover, we append the value of a counter that is increased by one until the object name becomes unique in this profile. When comparing two behavioral profiles that both contain objects with temporary filenames, it is possible to match these two objects. However, we have to avoid that an object $a_1$ of profile $A$ matches with object $b_1$ of profile $B$, when the operations associated with the object make it actually more similar to object $b_2$ of profile $B$. We address this problem by calculating a checksum over all OS operations, using the resulting value as part of the new object name. That is, execution-specific names are replaced with a new name of the form `<token><checksum><counter>`. The checksum guarantees that only objects with the same OS operations will receive the same name in two different profiles, and consequently match.

**Control Flow Dependences.** Control flow dependences are translated into comparisons between OS objects. Depending on the type of the comparison, a control flow dependency is associated with either one or two OS objects. A label-label comparison involves two OS objects (one for each operand), while a label-value comparison involves only a single one. To find the appropriate OS resource, the labels are used. That is, we search for the OS operation that created a particular label. Then, we search for the object that the operation is associated with.

## 3.5 Scalable Clustering

Clustering a set of $n$ points in a high-dimensional space is a computationally expensive task. Most clustering algorithms require to compute the distances between all pairs of points in the set. In this case, computational complexity is at least $O(n^2)$ evaluations of the distance function, which is unacceptable for large data sets.

There exist algorithms, such as the k-means algorithm (Lloyd's algorithm) [57], that only compute the distance from the $n$ points to $k$ cluster centers, and repeat this computation for each of $i$ iterations required to converge to a local

optimum. The computational complexity is, therefore, $O(nki)$ evaluations of the distance functions. Unfortunately, there are no guarantees that the value of $i$ is small (in fact, the number of iterations is super-polynomial in $n$ in the worst-case [21]). Furthermore, the accuracy of k-means is limited (the solution is only locally optimal), and the number of clusters $k$ has to be specified *a priori*.

In this work, we employ locality sensitive hashing (LSH), introduced by Indyk and Motwani [48], to compute an approximate clustering of our data set that requires significantly less than $n^2$ distance computations. Our clustering algorithm takes as input the set of malware samples $A = a_1, .., a_n$, where $a_i \subseteq F$, and $F$ is the set of all features. LSH algorithms have been proposed for metric spaces where the similarity between two points is defined by one of a few simple functions, such as Jaccard index [30], or cosine similarity [34]. In this work, we employ the Jaccard index as a measure of similarity between two samples $a$ and $b$, defined as $J(a, b) = |a \cap b|/|a \cup b|$. A similarity value of $J(a, b) = 1$ indicates that two samples have identical behavior. While other, more complex similarity functions, such as normalized compression distance [24], may be more accurate measures of the similarity between behavioral profiles, choosing this simple set similarity measure allows our clustering approach to leverage LSH and to scale up to the size of real-world malware collections.

In the following Section 3.5.1, we explain how we map a behavioral profile into a set of features that are suitable for LSH. Section 3.5.2 briefly explains the LSH algorithm. In Section 3.5.3, we discuss how we can use the output of the LSH algorithm to compute an approximate, hierarchical clustering of our malware sample set. Finally, in Section 3.5.4, we discuss the asymptotic performance of our approach.

### 3.5.1 Transforming Profiles into Features Sets

Before we can run the clustering algorithm, we have to transform each behavioral profile into a feature set. Informally, a feature is a behavioral characteristic of a sample, such as "file xy was created." We use the following algorithm to transform a behavioral profile $P = (O, OP, \Gamma, \Delta, CV, CL, \Theta_{CmpValue}, \Theta_{CmpLabel})$ into a set of features: For each object $o_i \in O$, and for each assigned $op_j \in OP|(o_i, a) \in \Gamma$, create a feature:

$$f_{ij} = "op|" + name(o_i) + "|" + name(op_j)$$

where $name()$ is a function that returns the name of an OS object, operation, or comparison as string, quotes (") denote a literal string, and + concatenates

two strings. Moreover, for each dependence $\delta_i \in \Delta = ((o_{i1}, op_{i1}), (o_{i2}, op_{i2}))$, we create a feature:

$$f_i = "dep|" + name(o_{i1}) + "|" + name(op_{i1}) + \\ + " \rightarrow " + name(o_{i2}) + "|" + name(o_{i2})$$

For each label-value comparison $\theta_i \in \Theta_{CmpValue} = (cmp, o)$, we create a feature:

$$f_i = "cmp\_value|" + name(o) + "|" + name(cmp)$$

For each label-label comparison $\theta_i \in \Theta_{CmpLabel} = (cmp, o_1, o_2)$, we create a feature:

$$f_i = "cmp\_label|" + name(o_1) + \\ + " \rightarrow " + name(o_2) + "|" + name(cmp)$$

The output of this transformation step is a set of features that captures the behavioral characteristics of a sample in a form that is suitable for the clustering algorithm. We then discard all features of a sample that are unique with regards to all other samples in the data set. That is, we do not consider a feature for clustering when it does not occur in at least one other sample's feature set. This is because a unique feature of a sample does not help us to find other samples that behave similarly (i.e., the information gain of this feature is very low). Moreover, our experiments show that the robustness of our clustering to the selection of the threshold $t$ improves when we discard such unique outliers.

### 3.5.2 Locality Sensitive Hashing (LSH)

The idea behind locality sensitive hashing is to hash a set $A$ of points in such a way that near (or similar) points have a much higher collision probability than points that are distant. We achieve this by employing a family $H$ of hash functions such that $Pr[h(a) = h(b)] = similarity(a, b)$, for $a, b$ points in our feature space, and $h$ chosen uniformly at random from $H$. By defining the locality sensitive hash of $a$ as $lsh(a) = h_1(a), .., h_k(a)$, with $k$ hash functions chosen independently and uniformly at random from $H$, we then have $Pr[lsh(a) = lsh(b)] = similarity(a, b)^k$.

In the case of sets for which the Jaccard index is used as similarity measure, a family of hash functions $H$ with the desired property has been introduced in [30]. A hash in $H$ imposes a random order on the set of all features. The hash value for a feature set $a$ is then determined by the index of the smallest element of $a$ according to this order. Since it is inefficient to generate truly random permutations, random linear functions in the form $h(x) = c_1x + c_2$

mod $P$ are used instead [46], with $P$ a prime number larger than the total number of features in $F$.

Given a similarity threshold $t$, we employ the LSH algorithm to compute a set $S$ which approximates the set $T$ of all near pairs in $A \times A$, defined as $T = \{(a, b)|a, b \in A, J(a, b) > t\}$. Given the threshold $t$, we first choose the number $k$ of hash functions in each LSH hash, and the number of iterations $l$. Furthermore, we initialize the set $S$ of candidate near pairs to the empty set. Then, for each iteration, the following steps are performed:

- choose $k$ hash functions $h_1, .., h_k$ at random from $H$
- compute $lsh(a) = h_1(a), .., h_k(a)$ for each $a \in A$
- sort the samples based on their LSH hashes
- add all pairs of samples with identical LSH hashes to $S$

**LSH Parameters.** For a given similarity threshold $t$, we must choose appropriate values of $k$ and $l$. For a pair $p = (a, b)$ such that $similarity(a, b) = v$, we have $Pr[p \in S] = 1 - (1 - v^k)^l = g(v)$. Thus, given $t$, we can choose $k$ and $l$ such that $g(t)$ is close to 1 and $g(t/(1 + \epsilon))$ is small, for any $\epsilon > 0$. That is, $t$ is the only parameter that needs to be chosen. For a threshold value of $t = 0.7$ we selected $k = 10$ and $l = 90$.

### 3.5.3 Hierarchical Clustering

The result of the locality sensitive hashing step is a set $S$, which is an approximation of the true set of all near pairs $T = \{(a, b)|a, b \in A, J(a, b) > t\}$. Because LSH only computes an approximation, $S$ might contain pairs of samples that are not similar. To remove those, for each pair $a, b$ in $S$, we compute the similarity $J(a, b)$ and discard the pair if $J(a, b) < t$. Then, we sort the remaining pairs by similarity. This allows to produce an approximate, single-linkage hierarchical clustering [56] of $A$, up to the threshold value $t$. Single-linkage clustering allows us to simply iterate over the sorted list of pairs to produce an agglomerative clustering. We stop the clustering when there are no more near pairs left.

In some cases, one would like to continue the hierarchical clustering process until all elements are merged into a single cluster. However, all subsequent clustering steps would require to merge two clusters that have a similarity value below $t$. Of course, this information is not readily available. The reason is that the LSH algorithm avoids the calculation of distances between elements that have a similarity value below $t$. To solve this problem and to obtain an

exhaustive, hierarchical clustering, we use the following technique: We choose a representative element for each cluster, calculate the distances between all representatives, and then perform exact, hierarchical clustering between these elements. We create the representative element $r$ of a cluster $C$ by adding all features to $r_C$ that exist in at least half of all the feature sets in $C$. Of course, exact hierarchical clustering has a complexity of $O(n^2)$. This is acceptable because the number of representatives is very low.

### 3.5.4 Asymptotic Performance

The LSH scheme described previously requires the computation of $nkl$ hashes. The computational complexity of each hash of a sample $a$ is $O(|a|)$. Therefore, the overall complexity of the hashing step is $O(nkld)$, where $d = avg(|a|), \quad a \in A$, is the average number of features in a sample. After hashing, $|S|$ similarity functions must be computed.

The set $S$ is an approximation of the true set of all near pairs $T$. We may, therefore, have false negatives $(T - S)$, and false positives $(S - T)$. We have $|S| \leq |T| + |S - T|$. Clearly, $|T| < nc$, where $c$ is the maximum cluster size for the given threshold. Unfortunately, we cannot provide a theoretical bound for the fraction of false positives $|S - T|/|S|$ without making some assumptions on the distribution of the distances between pairs in $A$. However, in practice, the value is small (below 0.19 in our experiments). Therefore, the number of similarity computations is limited by the size of $|T|$ and the complexity of $O(nc)$. Since a single similarity computation is $O(d)$, computational complexity of this step is $O(ncd)$. Finally, the pairs in $S$ need to be sorted to perform hierarchical clustering. This step is $O(nc \log(nc))$.

For large data sets, the cost of the similarity computations, which is $O(ncd)$, dominates. Note that while in practice $nc$ is significantly smaller than $n^2$, the asymptotic performance has not improved. The reason is that $c$ can still be $O(n)$ in the worst case. Consider, for instance, a trivial dataset where all $n$ samples are identical. Clearly, for such a dataset we would have a single cluster of size $n$ (and, therefore, $c = n$) for any $t$. More generally, if the threshold value $t$ is too low, it may lead to most samples being concentrated in a few large clusters. However, for meaningful datasets and reasonable values of $t$, $nc$ is significantly smaller than $n^2$. The performance gained by using LSH is therefore sufficient to allow us to cluster large, real-world malware data sets, as we will show in Section 3.6.3.

For extremely large datasets, on the other hand, more aggressive approximate clustering techniques may need to be employed (at the cost of some accuracy), such as the ones described in [46]. In [46], LSH is used to generate the set of approximate near pairs $|S|$, but there are no similarity computations.

A pair $(a, b) \in S$ is not verified to be near by computing $similarity(a, b)$, but by using a faster approximate method that is based on the already computed hashes.

## 3.6 Evaluation

To verify the effectiveness of our approach, we used our system to cluster real-world malware data sets. In the next section, we discuss the quality of the generated clusters. Then, in Section 3.6.2, we compare our solution with previously-proposed clustering techniques [24, 52]. In Section 3.6.3, we present performance measurements of running our prototype on a very large data set. Finally, in Section 3.6.4, we discuss some examples of the clusters produced by our tool and of the insight they provide to the malware analyst.

### 3.6.1 Quality

Assessing the quality of the results that are produced by a clustering algorithm is an inherently difficult task. Obviously, it is possible to quantify the number of clusters, the average number of samples per cluster, or the relative sum of all pairwise distances for a cluster. Alternatively, one could randomly pick a few clusters and manually verify that the samples in these clusters are similar. The best option for demonstrating the correctness of a produced clustering, however, is to compare it with an existing reference clustering. Unfortunately, no such reference clustering exists for malware samples[2]. As a result, to verify that our clustering approach is meaningful, we first needed to create a reference clustering.

**Reference Clustering.** To create a reference clustering, we took the following approach: First, we obtained a set of 14,212 malware samples that were submitted to ANUBIS [1] in the period from October 27, 2007 to January 31, 2008. These samples were contributed by a number of security organizations and individuals, spanning a wide range of sources (such as web infections, honeypots, botnet monitoring, peer-to-peer systems, and URLs extracted from other malware analysis services). Then, we scanned each sample with six different anti-virus programs. For the initial reference clustering, we selected only those samples for which the majority of the anti-virus programs reported the same malware family (this required us to define a mapping between the different labels that are used by different anti-virus products). This resulted

[2] In fact, providing a reference clustering for a set of malware samples is a difficult problem by itself, mostly because it requires human expertise to compile such a clustering or confirm the correctness of existing results.

in a total of 2,658 samples. For each sample, we examined the corresponding ANUBIS [1] report and manually corrected classification problems.

**Precision and Recall.** To evaluate the quality of the clustering produced by our algorithm, we compared it to the reference clustering described above. To quantify the differences between the two clusterings, we introduce two metrics, precision and recall.

The goal of *precision* is to measure how well a clustering algorithm can distinguish between samples that are different. That is, precision captures how well a clustering algorithm assigns samples of different types to different clusters. Intuitively, we strive for results where each cluster contains only elements of one particular type. More formally, precision is defined as follows: Assume we have a reference clustering $T = T_1, T_2, .., T_t$ with $t$ clusters and a clustering $C = C_1, C_2, .., C_c$ with $c$ clusters (for a sample set $A = a_1, a_2, .., a_n$). For each $C_j \in C$, we calculate a cluster precision value as:

$$P_j = max(|C_j \cap T_1|, |C_j \cap T_2|, .., |C_j \cap T_t|)$$

The overall precision value is:

$$P = \frac{(P_1 + P_2 + .. + P_c)}{n}$$

In addition to precision, we use *recall* to measure how well a clustering algorithm recognizes similar samples. That is, recall captures how well an algorithm assigns samples of the same type to the same cluster. Clearly, we prefer a clustering where all elements of one type are assigned to the same cluster. We formally define recall as follows: Assume we have a reference clustering $T = T_1, T_2, .., T_t$ with $t$ clusters and a clustering $C = C_1, C_2, .., C_c$ with $c$ clusters. For each $T_j \in T$, we calculate a cluster recall value as:

$$R_j = max(|C_1 \cap T_j|, |C_2 \cap T_j|, .., |C_c \cap T_j|)$$

The overall recall value is:

$$R = \frac{(R_1 + R_2 + .. + R_r)}{n}$$

The primitive algorithm that creates a cluster for each sample achieves optimal precision, but the worst recall. The algorithm that combines all samples in a single cluster, instead, achieves optimal recall but the worst precision. In practice, an algorithm should provide both high precision and recall. That is, each cluster should contain all samples of one type, but no more.

**Clustering Results.** We have run our clustering algorithm on the reference set of 2,658 samples. For this run, we selected a similarity threshold of $t = 0.7$. The value of this threshold was determined based on our experience with initial

experiments on a small malware sample set with less than a hundred programs. Later in this section, we discuss in more detail the considerations for selecting an appropriate threshold. Moreover, we will show that the algorithm is quite robust with regard to the choice of the concrete threshold value.

Our system produced 87 clusters, while the reference clustering consists of 84 clusters. For our results, we derived a precision of 0.984 and a recall of 0.930. This demonstrates that our approach has produced a clustering that is very close to the reference set. The excellent precision shows that the system was able to differentiate well between different malware classes. The recall shows that, in almost all cases, samples of the same class were grouped in the same cluster. A quantitative comparison to other clustering techniques is presented in the following Section 3.6.2. In Section 3.6.4, we discuss a number of interesting, qualitative observations about the clustering that our system produced.

**Threshold Selection.** The value of the similarity threshold $t$ determines how aggressively the clustering algorithm considers two different profiles as similar. Therefore, selecting a correct threshold often depends on the desired level of granularity of the clustering. For example, an analyst might be interested only in a rough partitioning of a set of malware samples into a few high-level categories (such as dialer, worm, or bot). Another analyst, instead, could be more interested in splitting a single malware family into different variants. In these cases, the first analyst would select a small $t$, while the second one would use a larger value for $t$.

For our experiments, we decided to use a threshold value such that our results would differentiate between malware families (that is, only similar variants of the same family should be clustered). As mentioned previously, a concrete value of $t = 0.7$ was selected, based on our experience with initial, small-scale experiments. However, the selection of the correct value of $t$ is quite robust. Figure 3.3 shows how precision and recall vary with respect to different choices of $t$. One can see that a broad range of choices for $t \in [0.6, 0.9]$ yield good results for both precision and recall.

## 3.6.2 Comparative Evaluation

In the previous section, we have shown that our system has performed accurate clustering. However, we need to put these numbers into context with other approaches to be able to better assess the quality of our results. In this section, we present a comparative evaluation with the current state-of-the-art clustering approach, introduced by Bailey et al. [24]. Moreover, we analyze the impact of our behavioral abstraction and compare our clustering to one

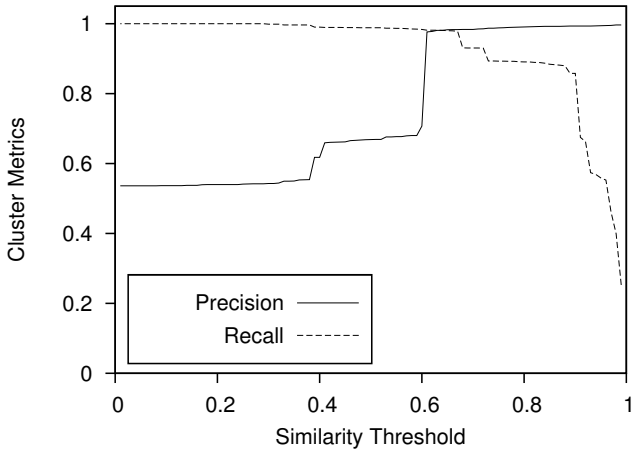


Figure 3.3: Precision and recall.

that is directly based on system call traces [52].

Bailey et al. [24] proposed a system for clustering malware based on the Normalized Compression Distance (NCD), using zlib-compression. NCD is based on the Kolmogorov complexity theory [54] and exploits the fact that similar data, when concatenated, compresses better than more differing data. Moreover, Bailey performs a coarse-grain abstraction from system calls and also uses profiles to represent malware behavior (we refer to these profiles as *Bailey-profiles* from now on). The difference to our approach is that Bailey-profiles contain only behavior in terms of non-transient state changes that a malware sample causes on the system (i.e., changes to the file-system, registry), as well as names of spawned processes and some basic information about network connections and scans. A detailed impression of the contents of Bailey-profiles can be gained from [23]. To evaluate Bailey's system on our reference data set, we adapted our dynamic analysis system to generate Bailey-profiles. Concerning NCD, we made use of the library provided by the Complearn-Toolkit [38].

A number of previous systems (e.g., [52]) based their behavioral profiles essentially on the raw system call traces. Thus, to evaluate the performance of such systems, and to obtain a baseline that shows the improvements due to generalized behavior profiles, we also performed clustering on the raw system call traces.

We used our reference clustering and the precision and recall metrics to directly compare the quality of the produced clusters for the different techniques. As an overall measure of clustering *quality*, we use the product of $precision * recall$. For each of the combinations of profile-types, similarity

measures, and clustering methods presented in Table 3.2, we selected the threshold value which produces the highest quality score. In the clustering column, "exact" means that all $n * n/2$ distances between pairs of samples were computed, while "LSH" means that locality sensitive hashing was used. The last two rows show that the difference between exact and LSH-based clustering is minimal, demonstrating the effectiveness of LSH-based clustering as an approximation.

| *Behavioral Profile* | *Similarity Measure* | *Clustering* | *Optimal Threshold* | *Quality* | *Precision* | *Recall* |
|---|---|---|---|---|---|---|
| Bailey [24] | NCD | exact | 0.75 | **0.916** | 0.979 | 0.935 |
| Bailey [24] | Jaccard | exact | 0.63 | **0.801** | 0.971 | 0.825 |
| Syscalls [52] | Jaccard | exact | 0.19 | **0.656** | 0.874 | 0.750 |
| Our profile | Jaccard | exact | 0.61 | **0.959** | 0.977 | 0.981 |
| Our profile | Jaccard | LSH | 0.60 | **0.959** | 0.979 | 0.980 |

Table 3.2: Comparative evaluation of different clustering methods.

As can be seen in Table 3.2, the quality of our clustering approach (last two rows) outperforms the clustering proposed by Bailey et al. (first row). This is because our profiles represent the actual behavior of a malware sample in a more comprehensive and accurate way. For example, certain samples exhibit behavior that cannot be captured using Bailey-profiles. As a result, such profiles remain empty, or almost empty. Even more troublesome is the fact that Bailey's approach produces significantly worse results when using the Jaccard index as a similarity metric instead of NCD (second row). Unfortunately, a clustering algorithm based on NCD cannot take advantage of LSH to avoid computing all $n^2$ distances. Thus, a clustering approach that uses Bailey-profiles [24] either produces results that are significantly less precise than ours (by using the Jaccard index and LSH), or it does not scale to real-world datasets (when using NCD). When analyzing the results for raw system call traces (third row), the results are significantly worse than for the other two techniques. This is not surprising, since the traces contain far too much noise to effectively find similarities between even closely-related malware instances.

### 3.6.3 Performance

To demonstrate the scalability of our clustering algorithm, we ran our system on a set of 75,692 malware samples (obtained from the complete database of ANUBIS). We performed our experiments on a XEN virtual machine that

was hosted on a PowerEdge 2950 server equipped with two Quad-Core Xeon 1.86 GHz CPUs and 8 GB of RAM. We allocated about 7GB RAM and one physical CPU to the XEN VM.

As shown in Table 3.3, our prototype implementation succeeded to cluster the set of 75,692 samples in 2 hours and 18 minutes. This time could be further reduced by exploiting the inherent parallelism: Both the LSH hashing and the distance calculation step can be easily performed in parallel. The memory requirements of our prototype never exceeded 3.7 GB of virtual memory. For each sample, we store a behavioral profile on disk, which consumes about 96 KB of disk space on average. To load the samples, the clustering algorithm had to read and process $6.9GB$ of behavioral profiles.

We ran the clustering algorithm with the same threshold value $t = 0.7$. The LSH algorithm computed a set $S$, our approximation of the set of near pairs, that contained 66,528,049 pairs. Only 57,024,374 pairs were indeed above the similarity threshold $t$, i.e., LSH hashing resulted in about 14% false positives. Nevertheless, employing LSH hashing allowed us to calculate only 66,528,049 instead of $(75,692^2)/2 = 2,864,639,432$ distances.

| *Algorithm Step* | *Time* | *(Virt.) Mem. Used* |
|---|---|---|
| Loading the samples | 58m | 1.6 GB |
| $l$ iterations of LSH hash. | 1h 0m | 3.6 GB |
| Distance calculation | 16m | 3.7 GB |
| Sorting all pairs | 1m | 3.7 GB |
| Hierarchical clustering | 3m | 3.7 GB |
| Total | 2h 18m | 3.7 GB |

Table 3.3: Runtime for 75K samples.

Compared to previous work, our prototype shows significantly improved performance. To classify malware based on NCD as in Bailey et al. [24], all of the $n^2/2$ distances between the $n$ samples need to be computed. Moreover, it is possible to derive from the run-time graphs presented in their paper that a single distance calculation between two pairs takes about 1.25 milliseconds. As a result, the distance calculation step of their algorithm would require 995 hours (almost 6 weeks) to perform the necessary $75,692^2/2$ distance calculations. This is despite the fact that Bailey profiles are rather small (about 1KB on average). Applying our NCD implementation to the (much larger) behavioral profiles produced by our tool yields even more prohibitive computation times: a single NCD computation takes on average 43 milliseconds. Therefore, clustering $75,692$ samples would take at least 6 months, even if the

implementation were parallelized to run on 8 CPUs.

### 3.6.4 Qualitative Discussion of Clustering Results

In this section, we present a number of observations on the quality of our clustering techniques. First, we discuss the four largest clusters (with regard to the number of samples that they contain). These are Allaple.1 (1,289 samples), Allaple.2 (717 samples), DOS (179 samples), and GBDialer.j (106 samples). Together, they account for 86% of all samples.

Allaple.1 and Allaple.2 are two different variants of the Allaple worm [10]. Allaple is a polymorphic malware, which explains why there are so many different samples in each cluster. It also demonstrates the ability of our system to quickly dispose of polymorphic malware instances that appear different but exhibit the same behavior. Interestingly, we found that virus scanners were inconsistently assigning different *variant names* to samples in both clusters (recall that we only used the malware *family names* that the virus scanners reported to perform the initial reference clustering). However, closer manual analysis showed that our clustering correctly identified two different Allaple variants. While all of the samples in both clusters perform ICMP scans, the Allaple.2 variant is much more aggressive at immediately attempting to exploit the target systems using a wider variety of propagation vectors. For instance, almost all Allaple.2 samples perform DNS lookups for the addresses of hosts they have successfully scanned, and attempt to connect to TCP port 9988, which corresponds to the Windows remote administration service. On the other hand, in none of the samples in the Allaple.1 cluster is there any DNS or port 9988 activity. Furthermore, all samples in Allaple.1 make a copy of themselves to the file "C:\WINDOWS\system32\urdvxc.exe," while none of the samples in Allaple.2 do. Moreover, in the Allaple.1 cluster, we observe the following, interesting object dependences:

```
Section|C:\sample.exe->Network|TCP
File|C:\WINDOWS\system32\urdvxc.exe ->
  File|C:\(..)\Temporary Internet Files\
  \(..)\ccxebztz.exe
Random|Random Value Generator ->
  File|C:\(..)\Temporary Internet Files\
  \(..)\ccxebztz.exe
```

The first dependency indicates that the sample has succeeded in propagating itself over the network (to our nepenthes honeypot). Since our taint-system correctly handles memory-mapped files, we see that the malware propagates by reading a memory-mapped file and writing it to the network. The second

and third dependences provide a strong indication that this is polymorphic malware, since data from the malware sample and from a random number generation API is written to the new file "ccxebztz.exe." This shows how system call dependences can provide valuable insight on malware behavior.

GBDialer.J is the biggest of several dialer clusters in our sample set. It is interesting that we were able to correctly group the samples in this cluster, because our analysis environment does not directly support the analysis of dialers. That is, there is no modem (emulation) present that would allow dialers to perform their main task. Nevertheless, the remaining behavior (such as startup actions and system modifications) was sufficiently characterizing to differentiate between the various dialer variants. This is not the case for the forth cluster, called "DOS." This cluster contains various DOS malware samples. The reason for not being able to distinguish between different DOS variants is that our analysis environment can only execute Windows PE executables. The Windows loader treats all non-Windows PE files as DOS executables, and attempts to execute them by emulating them in the `ntvdm.exe` process. This activity was recognized as similar behavior.

In addition to the four large clusters, there are several interesting, smaller clusters. For example, there is a cluster of only two samples that are labeled as "Keylogger.Ghostbot" by the Kaspersky virus scanner. Our dynamic analysis discovered that this malware constantly checks for key presses using the Windows API function `GetKeyState`. The profile contains the following interesting comparisons:

```
cmp_val|registry|HKLM\SOFTWARE\MICROSOFT\
 \WINDOWS\CURRENTVERSION\RUN
 NtEnumerateValueKey-KeyValueInformation
 - PCCNTMON
```

This tells us that the malware looks for known anti-virus and firewall programs in the list of autostart registry values. Please note that the above is only an excerpt. In total, the profile lists 98 different program names that are compared against the result of `NtEnumerateValueKey`. We also have a cluster that consists of four samples that are recognized as "Mabezat" by the majority of virus scanners. Our behavioral profile shows that it is a file infector that searches for executable files on the local hard disk and infects them. This characteristic behavior was correctly identified and resulted in one cluster that precisely captured all four samples in the data set. We also discovered, with the help of control flow dependences, that the program is searching for different kinds of document files in the directory that Windows uses for temporarily storing data that is scheduled to be written to a CD. Again, we show only parts of the list of comparisons.

```
cmp_val|file|
C:\Documents and Settings\user\Local
 Settings\Application Data\Microsoft\
 \CD Burning\
 NtQueryDirectoryFile-FileInformation
  - .TXT
```

According to the virus description database of AVG [3], the malware program checks whether the current date is greater then 2012/10/16, and if so, starts encrypting user documents. Our system was only partly able to find this date check. Our profile is shown below:

```
cmp_val|time|System Time
 GetSystemTime-
 lpSystemTime.struct _SYSTEMTIME.wYear
 -2012
```

As one can see, the system correctly recognizes the fact that a comparison between the current year and the value 2012 takes place. As this comparison already fails, the rest of the date is not further checked. That is why we cannot determine the complete date. However, we are considering to improve our system with the ability to read the entire data structure from the main memory (in a fashion that is similar to our current approach for strings).

Of course, there are also malware programs for which our system did not produce the correct results. One common case is when a sample did not show any suspicious activity in our analysis environment. This could be because the malware program is damaged, or because it detects the presence of the analysis environment and exits prematurely. In any case, it underlines the dependence of our system on the quality of the behavioral profiles. One cluster in particular is composed of 25 samples which belong to 10 different clusters according to the reference clustering. Manual analysis reveals that these samples all crash, which causes the Dr. Watson debugger application to be executed, generate a crash report, and display a pop-up window asking the user permission to send the report to Microsoft. Clearly, this behavior is not specific to the malware family and it leads to misclassification.

## 3.7 Limitations and Future Work

**Trace Dependence.** As mentioned previously, a limitation of any dynamic malware analysis approach is that it is trace-dependent. Analysis results will be based only on the sample's behavior during one (or more) specific execution runs. Unfortunately, some of a malware's behavior may be triggered only

under specific conditions. A simple example of trigger-based behavior is a time-bomb. That is, a malware that only exhibits its malicious behavior on a specific date. Another examples is a bot that only performs malicious actions when it receives specific commands through a command and control channel. Also, malware aimed at identity theft may only exhibit certain behavior when the user performs certain actions, such as logging into specific banking websites. Since we run malware samples automatically with no human interaction, such behavior will not occur in our traces.

Interestingly, our clustering may still succeed in grouping similar samples even when their most significant malicious behavior is not triggered, as is the case for the GBDialer.J cluster discussed in Section 3.6.4. The reason is that the behavioral features used for clustering encompass all malware behavior, not just malicious actions. Also, one could use techniques that explore multiple execution paths [60] to obtain a more comprehensive picture of the functionality of a program.

**Evasion.** Clearly, a malware author could manually modify a malware sample until its behavior is different enough from the original that the two are assigned to different clusters by our tool. We are not interested in this kind of labor-intensive, manual evasion. Instead, we consider an adversary who attempts to automatically produce an arbitrary number of mutations of a malware sample in such a way that all (or most) such mutations are assigned to different clusters by our tool. To this end, a malware author could randomly mutate parts of the malware's behavior that are not essential to its functionality. An example would be the often arbitrary file names under which the malware copies itself on the file system. These could be replaced with random strings, hard-coded into each malware instance. Nonetheless, adding enough randomness to make each mutation different is not a simple task. A sample in our dataset has more than one thousand features on average, many of which represent behavior from inside system libraries that is only indirectly a consequence of the malware writer's intent. Also, since our tool discards features that are unique to a single malware instance, simple random variations would just lead to these features being discarded. In addition, we could add more aggressive generalization to our algorithm for extracting behavioral profiles. As an example, we could consider the name of any file created by the malware as irrelevant, and replace it with a special token (as we currently do for the names of temporary files).

Another issue is that dynamic data tainting of untrusted software is vulnerable to evasion. A malicious binary could inject fake data dependencies, using NOP-equivalent operations to taint clean data without modifying its value. Furthermore, it could hide data dependencies from our tool, using

implicit flows to "clean" tainted data [33]. Unfortunately, there is no easy defense against such techniques. To address this issue, we would have to disable dynamic data tainting, sacrificing some of the system's accuracy.

## 3.8 Conclusion

In this chapter, we propose a novel approach for clustering large collections of malware samples. The goal is to find a partitioning of a given set of malicious programs so that subsets exhibit similar behavior. Our system begins by analyzing each sample in a dynamic analysis environment that we have enhanced with taint tracking and additional network analysis. Then, we extract behavioral profiles by abstracting system calls, their dependences, and the network activities to a generalized representation consisting of OS objects and OS operations. These profiles serve as the input to our clustering algorithm, which requires less than a quadratic amount of distance computations. This is important to handle large data sets that are commonly encountered in the real world. Our experiments demonstrate that our techniques can accurately recognize malicious code that behaves in a similar fashion. Moreover, our results show that we can cluster more than 75 thousand samples in less than three hours.

The approach presented in this chapter allows us to better manage the large number of analysis reports that automatic, dynamic analysis systems are creating in response to the flood of new malware files appearing each day. In the following chapter, we will present an approach that allows us to keep up with the ever-increasing, high number of malware samples each day. By streamlining the automated analysis process, we are able to substantially improve the number of analyzed files per day without requiring additional hardware infrastructure.

# Chapter 4

# Improving the Efficiency of Dynamic Malware Analysis

## 4.1 Introduction

Automated, dynamic malware analysis systems work by running a binary in a safe environment, monitoring the program's execution and generating an analysis report summarizing the behavior of the program. These analysis reports typically cover file activities (e.g., what files were created), Windows registry activities (e.g., what registry values were set), network activities (e.g., what files were downloaded, what exploit were sent over the cable), Windows service activities (e.g., what services were installed) and process activities (e.g., what processes were terminated). Several of them are publicly available on the Internet (Anubis [1, 28], CWSandbox [5], Joebox [13], Norman Sandbox [16], ThreatExpert [19]) but many similar internal systems exist behind the closed doors of A/V companies.

In this chapter, we present a novel and practical approach for improving the efficiency of dynamic malware analysis systems. Our approach is based on the insight that the huge number of new malicious files appearing each day is due to mutations of only a few malware programs [45]. More precisely, malware authors write programs that reproduce polymorphically [72] or employ run-time packing algorithms to create new malware instances that differ on the file level, but exhibit the same behavior. We propose a system that avoids analyzing malware binaries that merely constitute slightly mutated instances of already analyzed polymorphic malware. To detect polymorphic binaries, we have extended our dynamic analysis system to check—after executing the malware program for only a short time—whether our database of existing analysis reports contains a behaviorally almost identical (for the time frame in question) analysis report. If this is the case, we stop the analysis process and instead, return the existing analysis result.

The contributions of our technique outlined in this chapter are as follows:

- We propose an approach that drastically reduces the amount of time re-

quired for analyzing a set of malware programs. To achieve this, we avoid analyzing the same polymorphic program multiple times. For detecting that a program is a polymorphic variation of an already analyzed binary, we dynamically analyze it for a short period of time. In a next step, we search the behaviorally nearest program. If such a program is similar enough (with respect to a specified threshold), we stop the currently ongoing analysis and instead return the existing analysis result.

- We present experimental evidence that demonstrates that our approach is feasible and usable in practice.
- We have designed an algorithm that is efficient and scalable. We find a program's behaviorally nearest neighbor without having to perform $n - 1$ comparisons.

## 4.2 Background: Analysis Time

A dynamic malware analysis system faces the problem that it has to analyze as many suspicious binaries as possible within a limited time frame and a limited amount of computing resources available. At the same time, it still has to provide meaningful analysis reports. Clearly, it is necessary that a dynamic analysis system executes and monitors a given binary for a reasonable amount of time to determine the binary's purpose. Traditional systems either analyze a given binary until its execution as well as the execution of all of its children processes ends, or a certain timeout limit has been reached. This timeout is four minutes long in the case of the dynamic analysis system that we are modifying. This means that the execution of a binary under analysis lasts for a maximum of four minutes. The total analysis time for a file, however, might be longer because in most cases, a post-processing step follows the actual execution phase. Our system, for instance, permits the post-processing step to run for a maximum of another four minutes. In case the program exits (or dies because of an error) before the timeout is reached, the analysis will naturally take less time. The assumption behind this modus operandi is that the typical malicious program tries to perform its malicious actions as soon as possible. However, we want to point out that this assumption is not always true and that a longer analysis might be desirable in some situations . For example, a binary could try to sleep for several minutes before it begins its (malicious) work. To allow for a longer analysis (in certain cases or in general), even more computing resources are required. In the following paragraphs, we will describe a technique that reduces the amount of required analysis time. Thus, this technique helps a dynamic analysis system both to analyze more

programs in a given period and to analyze programs for a longer amount of time. More formally, this relationship can be described as:

$$OverallAnalysisTime = (|B| * \sum_{b \in B} t_a(b))/I$$

with $B$ being our set of binaries, $t_a$ the analysis time for a single binary and $I$ the number of instances of the analysis system that are running in parallel.

More precisely, the analysis time $t_a(b)$ of a binary $b$ is composed of a *setup time* $t_s(b)$ and a *post-processing time* $t_p(b)$ in addition to the actual time $t_e(b)$ used for executing the binary $b$ in a secure environment. That is:

$$t_a(b) = t_s(b) + t_e(b) + t_p(b)$$

During the setup time, we prepare the analysis environment—possibly by loading a virtual machine and transferring the program into it. In the final post-processing step, we apply all kinds of offline analysis methods to the information gained during the execution of the binary. Tasks performed during the post-processing step range from archiving the analysis result and updating databases to running scripts for analyzing a network traffic dump file.

Note that analysis systems usually treat binaries scheduled for analysis as a mathematical set consisting of unique files. That is, to save analysis time, one avoids analyzing the same file multiple times. Practically, this technique is implemented by computing a hash value for the file before the analysis starts. If a matching analysis result already exists in the report repository, the analysis system can simply return the already existing result to the user.

## 4.3 Reducing the Overall Analysis Time

Our solution is based on the insight that the large quantity of new malicious files appearing each day is due to mutations of only a few malware programs (e.g., polymorphic reproduction or use of runtime packing algorithms with a random crypt seed resulting in a slightly changed binary). Indeed, we made the experience that, in our system, analysis reports are in many cases almost identical suggesting that we've analyzed a polymorphic malware instance several times. We propose a system that makes use of the fact that we can avoid analyzing the large percentage of incoming malware binaries that merely constitute slightly mutated instances of already analyzed polymorphic malware binaries. This system is the logical extension of the hash-based technique that saves analysis time by not analyzing the same file (identified via its hash value) twice.

To this end, our system checks after running a binary $b$ for only a short amount of time that we call the *checkpoint time* $T_c$ whether the behavior seen in this short time is almost identical to behavior seen in a previous analysis. If this is the case, we stop analyzing the binary $b$. We return the analysis result of the program that we found to behave almost identically instead. This means that we are able to deliver a full analysis report which covers all of a program's behavior as observed in time $t_e(b)$ for a binary $b$ that we effectively analyzed only for a much shorter period $T_c$ . Of course, this scheme only makes sense if $T_c$ is a lot smaller than $t_e(b)$: $T_c << t_e(b)$. We will discuss the selection of $T_c$ in the evaluation section.

Thus, the analysis time of a pre-empted binary $b$ is given by $t_{pre-empted}(b) = t_s(b) + T_c$. Since in the case of pre-empted binaries we return an already existing analysis result there is no need for a post-processing phase. The time saved by pre-empting a file $b$ is consequently $t_a(b) - t_{pre-empted}(b)$.

### 4.3.1 Behavioral Profiles

To determine whether a program's behavior after time $T_c$ corresponds to one that we have already analyzed, we leverage a presentation of a program's behavior that we call *behavioral profile*. We represent a binary's $b$ behavioral profile as $bp(b)$ in this chapter. Behavioral profiles were introduced in Chapter 3. A behavioral profile aims to capture a program's behavior at an higher level of abstraction than a raw system call trace while correctly retaining a program's behavioral semantics. Among other things, a behavioral profile relies on information gained from the data tainting system of a dynamic analysis program. This is used, for example, in order to determine whether execution artifacts, such as filenames, registry-key names, etc. depend on randomness and thus change with every execution of the program. Clearly, it is of utter importance to detect randomness when comparing two behavioral profiles. It is known, for example, that the polymorphic Allaple worm scans a randomly chosen IP sub-net for potential victims. Even in the simplistic case of comparing two different executions of the same binary we have to detect that the target IP is randomly chosen for achieving a high similarity score. We refer the reader to chapter 3 for the details of behavioral profiles.

For this project, it proved useful to extend behavioral profiles with timing information. More concretely, we assigned a timestamp value to each feature representing the feature's first occurrence in an execution trace. This permits us to order features based on their first occurrence. Moreover, it allows for more advanced comparison techniques between behavioral profiles. Please note that a behavioral profile still remains a set of string features. If, for

instance, a program creates and deletes a certain file several times, the behavioral profile contains only a single feature representing the file's creation. Its timestamp would equal the time when the program created the file the very first time. This timestamp value specifies the offset to a well defined starting time. We decided to use the time when a program's very first user-mode instruction is being executed as the starting time. This approach is robust against varying durations of the setup phase where among other things we have to load the snapshot and copy the program into the virtual environment.

### 4.3.2 Comparison

We consider a program $b$ to be a polymorphic variant of another program $a$ if the distance between their behavioral profiles at time $T_c$ is below a certain distance threshold $d$. Formally, we demand that $dist(bp(a), bp(b)) < d$. As a distance function we employ the Jaccard distance [49], defined as

$$J(a, b) = 1 - |a \cap b|/|a \cup b|$$

We define two programs $a$ and $b$ as being *behaviorally identical* if $J(bp(a), bp(b)) < d$ is true. In the evaluation section, we are going to discuss the selection process for the distance threshold parameter $d$. In the ideal case, we would expect to have a distance of 0 between the profiles of two behaviorally identical binaries. Practically, our experiments show that this distance is rarely exactly zero. The reason for that is that our behavioral profile cannot capture all randomized artifacts. Another reason is that frequently the analyzed program is not able to execute the exact same number of system calls during different analysis runs. This is due to OS scheduling decisions, differing server workloads, network connection latencies and similarities.

**Extended Jaccard Distance.** Although we perform our analysis runs each time in exactly the same configuration, we cannot prevent the existence of a small number of differences in behavioral profiles due to timing issues. Consider, for example, that we have chosen a checkpoint time $T_c$ of 45 seconds and that we have two analysis runs of the same file. Let us furthermore assume that this file is programmed to sleep for 45 seconds and to proceed by creating twelve files. Clearly, it is easily possible that in one execution all twelve files have been created at time $T_c$ while in the other one no files at all have been created. To alleviate this problem, we introduce an extended Jaccard distance $J_e$ that is more robust against the described timing issues.

Without loss of generality we assume that for two behaviorally identical programs $a$, $b$ the program $a$ has already advanced further in its execution at a certain point in time. Thus, its behavioral profile contains more features.

At the same time, $b$'s behavioral profile $bp(b)$ is an approximate subset of $bp(a)$ (since we are assuming that b exhibits the same behavior). We define an approximate subset as a relationship where a large percentage $p$ of the features are the same: $|bp(b) \cap bp(a)|/|bp(b)| >= p$. For our experiments, we have—based on our experience with the reference set—chosen a value of 0.9 for p. We will demonstrate that this selection of $p$ is reasonable and yields good results. This model motivates the following algorithm for computing a more timing resilient distance value:

1. if $bp(b)$ is not an approximate subset of $bp(a)$ we stop and return the normal Jaccard distance $J(bp(a), bp(b))$

2. otherwise we select the feature $f_{highest} \in bp(a) \cap bp(b)$ with the highest timestamp

3. we compute a $bp_{normalized}(a)$ by removing all features from $bp(a)$ with a timestamp higher than $timestamp(f)$

4. return $J(bp_{normalized}(a), bp(b))$ as a result

In our experiments, we compare the results achieved by employing either one of them. The computation of the extended Jaccard distance is more costly than the simple Jaccard distance. Additionally, as we will see in the next section, it lacks some of the properties that make the Jaccard distance so attractive. In particular, techniques exist that allow the efficient search for a behavioral profile's nearest neighbor when the Jaccard distance is used as a distance metric. This is why we use the Jaccard distance as our primary distance metric and resort to the extended Jaccard distance only in a second step when it is computationally more feasible.

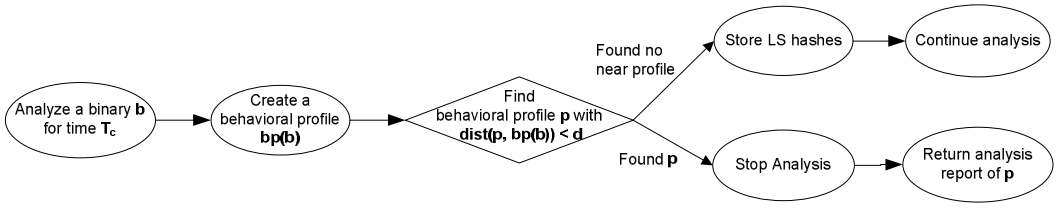


Figure 4.1: Overview of our approach for saving analysis time

### 4.3.3 Efficient Nearest Neighbor Search

As explained in the preceding paragraphs, we represent a program's behavior in the form of a *behavioral profile* and use the Jaccard distance for determining the dissimilarity between two behavioral profiles. In this subsection, we will describe how to efficiently find for a program $b$ at time $T_c$, an almost identical program if such a program exists (i.e., was already analyzed). The naive solution, a *linear search*, would be to compare $bp(b)$ with all existing behavioral profiles. This solution has a runtime complexity of $O(n * d)$ where $n$ is the number of behavioral profiles in our database and $d$ is the number of features present in the union of all behavioral profiles. In addition to the runtime costs, we would need to keep all behavioral profiles in main memory to allow for a efficient comparison. This is clearly not very scalable.

A more efficient technique for finding the nearest behavioral profile is *Locality Sensitive Hashing* (LSH) [48]. LSH provides an efficient (sublinear) solution to the approximate nearest neighbor problem ($\epsilon$-NNS). We already successfully leveraged LSH for developing the scalable, malware clustering system that was described in Chapter 3. In the following, we assume that the reader has already read the description of LSH in Chapter 3.

### 4.3.4 The Analysis Process

In this subsection, we explain the steps necessary to integrate our approach into the traditional analysis work flow. First, we we assume that the results of completed analysis runs are being stored. Second, the LSH configuration consisting of $l * k$ hash functions $h_{1,1}, ..., h_{k,l}$ has to be selected once and stored persistently for later use. Third, LS hashes for completed analysis runs need to be stored persistently in a hash database.

Figure 4.1 gives an overview of the entire analysis process. After analyzing a binary $b$ for $T_c$ seconds, we create a behavioral profile $bp(b)$. This profile captures the program's behavior until time $T_c$. In a next step, we employ LS hashing to find the set of candidate near behavioral profiles $N$. To this end, we first initialize the set $N$ to the empty set. Then, this search is performed in $l$ iterations with each iteration consisting of the following steps:

1. We load the $k$ hash functions $h_1, ..., h_k$ for iteration $l$ from our persistent storage
2. We compute the LS hash for binary $b$: $lsh(b) = h_1(b), ..., h_k(b)$
3. We check whether $lsh(b)$ exists in our database of hashes and add all behavioral profiles with identical LS hashes to the set of candidate neighbors $N$

Since the set of candidate neighbors $N$ might contain false positives due to the probabilistic nature of LS hashing, we compute all the distances $J(b, n)$ for all $n \in N$. In the evaluation section, we will demonstrate that results of replacing $J$ with $J_e$ for this step. We keep the nearest behavioral profile $n$ if one exists with a distance smaller than our chosen threshold. In case we found a behaviorally identical profile, we stop the currently ongoing analysis and return the analysis report of profile $n$. Otherwise, we store the $l$ LS hashes that we calculated before in our hash database and let the dynamic analysis continue.

## 4.4 Evaluation

To verify the correctness and efficiency of our approach, we have implemented a prototype system. We will first shortly describe our prototype implementation and then discuss the experiments that we conducted, and the results obtained.

### 4.4.1 Prototype Implementation

For testing our approach, we modified an existing dynamic analysis system. In the following, we summarize the most important changes:

- *On-the-fly generation of the behavioral profile*: First, we had to modify the analysis system so that behavioral profiles are built incrementally while the analysis of a program progresses. Each invocation or return of a system call triggers the update of our behavioral profile. Consequently, it is possible to create a behavioral profile at each point during the analysis of a program. Special handling was necessary to account for network actions. Since the existing network analysis examines the raw network traces, we have to execute this network analysis script (which in turn parses the network traffic dump file) each time a behavioral profile is generated.

- *Timestamps*: The generation of behavioral profiles was modified to include a timestamp for each feature. The timestamp indicates the time of a feature's first occurrence.

- *LSH*: For performance reasons, the LSH computation code is written in C++. To interface this code with the rest of the analysis scripts, which are written in Python, we wrapped the C++ code in a Python module (with the help of Boost.Python [4]). The LS hashes for a profile are stored in a relational DBMS (MySQL). Searching for LS hashes and adding new hashes is performed via regular SQL queries.

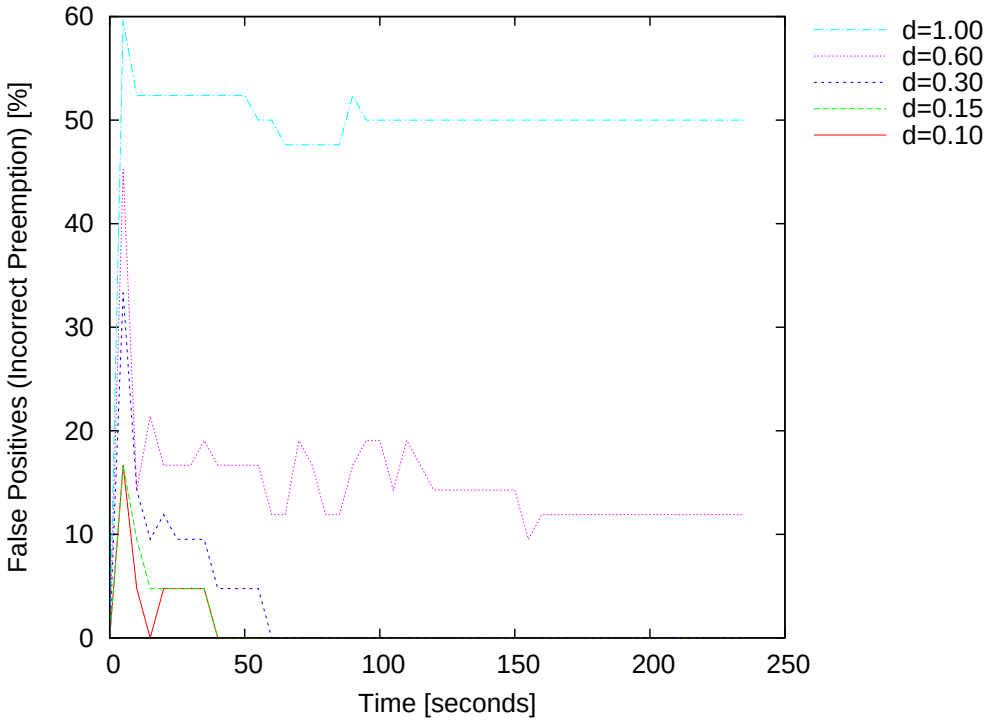


Figure 4.2: False Positives

- *Mapping feature strings to integer values*: It would be inefficient to perform all of our distance and LSH computations directly with behavioral profiles in the form of sets of (feature) strings. Instead, we map each feature string to a unique integer value with the help of a table in our relational DBMS. Currently, we store feature ids as 32-bit numbers.

- *LSH configuration*: We decided to store the LSH configuration in the relational DB as well. This permits each analysis run to rebuild the identical $k*l$ hash functions $h_{1,1}, ..., h_{k,l}$. The LSH configuration consists of $l * k$ (pseudo-) random numbers $c_1$, $c_2$ and the prime number $P$. It is necessary to select a prime number $P$ that is higher than the number of all features. Since we cannot predict how many features we will have in the future (each analysis adds new features) we chose the largest 32 bit prime number for $P$.

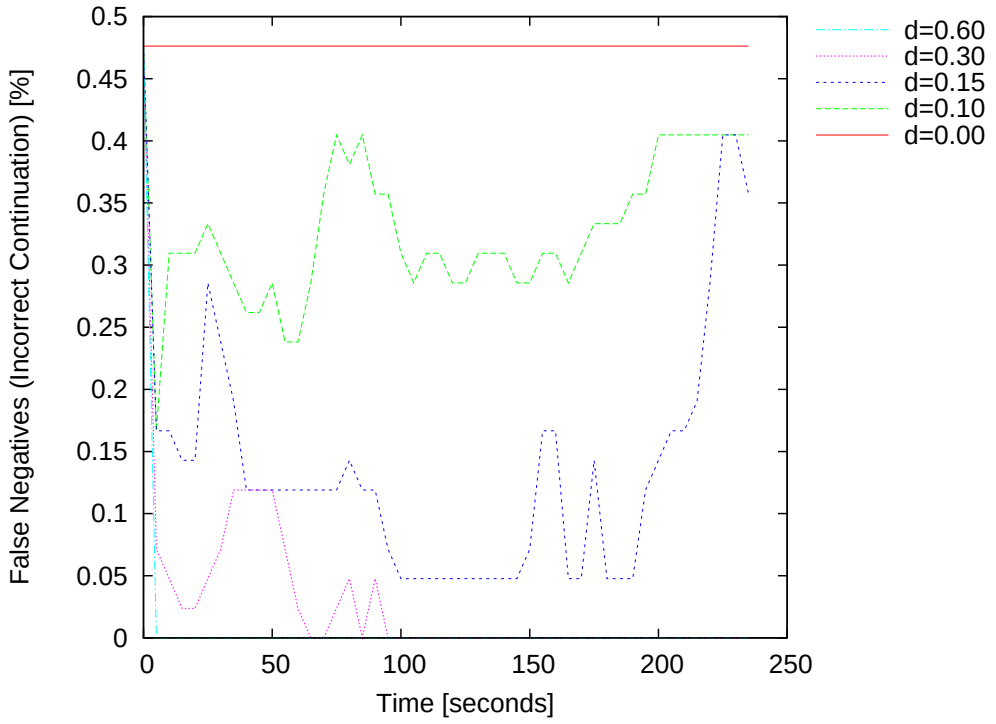


Figure 4.3: False Negatives

### 4.4.2 Experiment with a Reference Set

To assess the effects of the checkpoint time parameter $T_c$ and the distance threshold $d$ on our algorithm, we chose to compare the outcome of our algorithm under different configurations with a reference set.

**Reference Set.** In a first step, we manually compiled a set of 20 polymorphic programs and 22 non-polymorphic programs, 42 files in total, that should serve as our reference set. More precisely, we included four different types of malware which are known to be polymorphic and which appeared in the wild during August 2009:

- *Virut*: Virut is a polymorphic file infector [8, 9]. It infects files with an .EXE or .SCR extension by appending a slightly modified copy of itself to the file. We were able to manually verify that the five Virut files in our set were indeed all infections of the same original executable file. That is, the five Virut files contained the same host file and only differed in their last section where the actual (polymorphic) virus code resides.

- *Allaple.1*: Allaple [6] is a polymorphic worm that spreads by exploiting a number of vulnerabilities. Whenever the worm propagates it newly encrypts its code. The result is a copy of the virus that differs at the byte-level from its source.

- *Allaple.2*: This is another variant of Allaple.

- *Trojan-PWS.Win32.LdPinch*: LdPinch [7] is a Trojan that is designed for stealing passwords and mailing them back to the virus author. Unlike viruses and worms Trojans do not replicate. Consequently, someone must have created the different LdPinch files in our set. We discovered that a toolkit for creating Pinch Trojans exists [2] that allows for the easy creation of new Pinch Trojans. We suspect that this or a similar tool was used by the virus author(s) for the semi-automatic creation of new LdPinch files. The toolkit was first detected in the wild in 2008 but newly created variants continue to appear. Mcafee, for example, included a new variant in its signature file update on 08/28/2009 [18].

Each of the four polymorphic malware programs in the preceding list is represented by five unique binaries in our reference set.

**Selecting the checkpoint time and the distance threshold.** The time parameter $T_c$ determines after how many seconds we search for the nearest behavioral profile. The distance threshold $d$ specifies the maximum distance that two behavioral profiles are allowed to have in order to be considered as behaviorally identical. To understand the effects of these parameters on our results, we conducted the following experiment.

For all parameters combinations $T_c \in \{1, 2, ..., 240\}$,
$d \in \{0.05, 0.1, 0.15, ..., 0.4, 0.5, ..., 1.0\}$ we calculated a full distance matrix based on the Jaccard distance. After choosing the nearest profile for each file, we decided in accordance with the current threshold $d$ whether the analysis of this file is pre-empted or not. Each time, we compared the outcome with the reference set. We measure the success of our algorithm by calculating the number of the *false positives* (i.e., the number of programs that were wrongly determined to be behaviorally identical) and the number of *false negatives* (i.e., the number of programs that we did not correctly identify as being behaviorally identical).

In this experiment, we are mainly interested in the distances between behavioral profiles at specific execution times. This is why, we we do not assume any specific submission order and calculate all possible distances.

Figure 4.2 shows the percentage of false positives in relation to the checkpoint time. The figure contains five different lines showing the false positive

rate for five different distance thresholds. Naturally, a higher distance threshold leads to higher false positive rate. In the extreme case of a distance threshold of $d = 1.00$, all 22 non-polymorphic binaries are wrongly found to be behaviorally identical with one of the four polymorphic malware types. On the other hand, a value of $d = 1.00$ leads to 0 percent of false negatives. Furthermore, we can see that the percentage of false positives diminishes over time. This makes perfectly sense because the longer we execute a program the more characteristic actions we are going to include in its behavioral profile.

We show the false negative rate in Figure 4.3. In contrast to the false positives, the false negative rate improves when the distance threshold increases. A threshold of $d = 0.30$ paired with a checkpoint time greater than approximately 100 seconds is sufficient to not miss any polymorphic binary. That is, we recognize all polymorphic programs correctly as being polymorphic. In the extreme case of $d = 0.00$, none of our 20 polymorphic programs is correctly recognized as being polymorphic. Moreover, it is interesting to see that a distance threshold of 0.10 still results in quite a high number of false negatives. This indicates that our behavioral profiles still contain more execution-specific artifacts than we desire. It is noteworthy that the false negative rate fluctuates a lot at different checkpoint times in case of tight thresholds, such as 0.10 or 0.15. This is because a longer analysis time increases the number of actions and the number of execution artifacts in a behavioral profile. As a consequence, a longer analysis time can make behavioral profiles drift more apart than would be appropriate.

### 4.4.3 Real-World Experiments

After completing our experiments with the reference set, we started to test our algorithm in a real-world setting. To this end, we installed and operated our prototype system inside our dynamic analysis platform for several days. In this period, the system has analyzed a set $B$ of 10,922 unique executable files. For each analysis, we created and stored a behavioral profile including timestamps for all features. In this experiment, we did not stop the analysis of any programs prematurely. Instead, we decided to allow all programs to continue running until the normal timeout is reached. This puts us into a position to review the full behavior of otherwise pre-empted programs and to reason whether our algorithm's decision to prematurely end an analysis run is justified.

While it is straight-forward to calculate the total amount of time saved by running our algorithm in this real-world setting, it is inherently more difficult to estimate the number of false positives and false negatives in these results.

We developed the following strategy to evaluate how reliable our algorithm is in its decisions to prematurely stop an analysis : For all the programs $b_i \in B$, that, according to our algorithm, have a behaviorally identical program $s_i \in B$ at checkpoint time $T_C$, we compute the pair-wise Jaccard distances at the time of the traditional analysis end. In other words, we are evaluating how much the behavior of two programs $a$ and $s$ that were found to be behaviorally identical at an earlier time $T_c$ differs after the normal timeout of four minutes. This distance calculation permits us to quantify to what extent our analysis result would have differed in case our algorithm was actively deployed. We are not claiming that the current analysis results, which are delivered after 4 to 8 minutes, are always correct. We are only examining the question to what degree our algorithm, while saving time, returns possibly worse analysis results. This strategy does not allow us to directly measure the false positive rate, but it is suited to give us an estimate of the false positive rate.

Although the size of the real-world set makes an exhaustive evaluation of all possible parameters infeasible, we were able to try out several interesting ones thanks to the fact that we had the full behavioral profile including timestamps available. We show the results of performing five runs of our algorithm on set $B$ with varying parameters for the checkpoint time $T_c$ and the distance threshold $d$ in Table 4.1. Our initial parameter selection was guided by our experience gained while performing tests on the reference set. Additionally, we were examining the effects of using the extended Jaccard distance.

| *Configuration* | *Pre-empted files* | *Time saved/ pre-emption* | *Total time saved* |
|---|---|---|---|
| 45s, 0.12 | 3,087 (28.26%) | 265s | 227.2 hours |
| 60s, 0.12 | 2,747 (25.15%) | 250s | 190.8 hours |
| 60s, 0.12, $J_e$ | 3,659 (33.5%) | 250s | 284.1 hours |
| 60s, 0.08 | 1,653 (15.13%) | 250s | 114.8 hours |
| 60s, 0.08, $J_e$ | 2,539 (23.24%) | 250s | 176.2 hours |

Table 4.1: Results of testing our approach in different configurations

When testing a new configuration, of course, no LS hashes exist in the beginning. Clearly, for the very first program, we cannot possibly find a nearest neighbor. During each run of our algorithm, we performed the steps detailed in section 4.3.4. This means that we iterated (in the same order that the files were originally analyzed) over the set of files in $B$: For each $b \in B$, we searched the nearest behavioral profile $s_i \in B$. This search process consists of the LS hashing step that efficiently calculates a set of candidate near

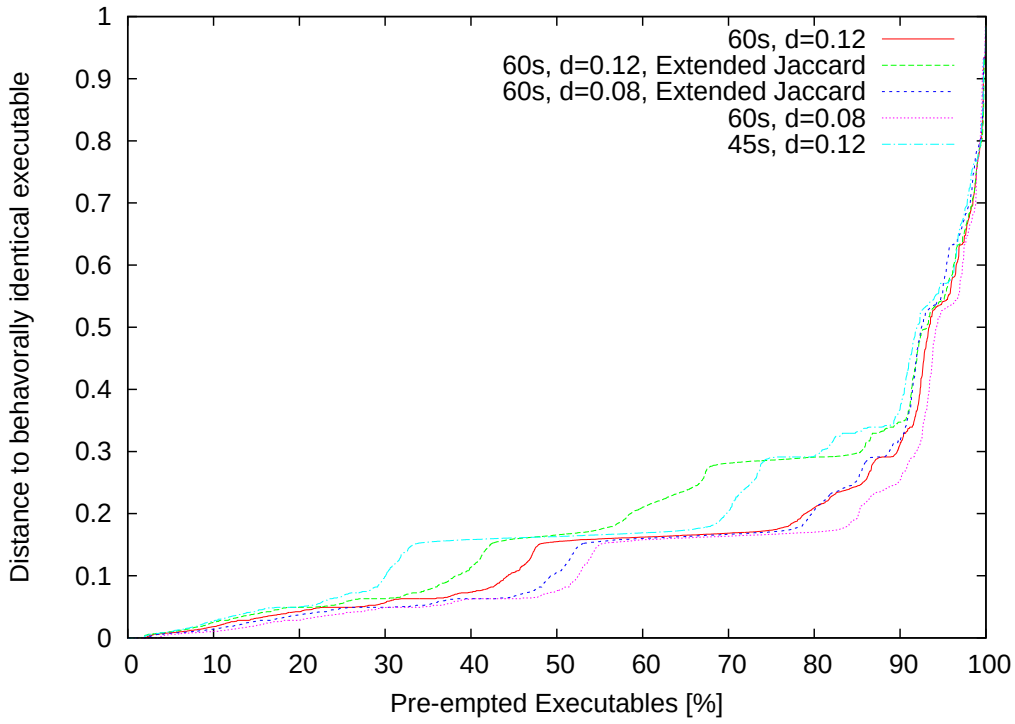


Figure 4.4: CDF in [%] of distances $J(b_i, s_i)$ at time $t_e$

profiles and a subsequent traditional comparison with all the candidate near profiles to eliminate false positives. The number of traditional comparisons averaged to 1.2820 during all our runs. For practical reasons, we conducted all our algorithm runs with a value of $l = 140$ and $k = 25$. We selected these values to get good results for distances of 0.15 and smaller. More precisely, these parameters cause behavioral profiles with a distance of 0.15 to collide with a probability of 0.912. The chosen value for $k$ and $l$ allowed us to test our algorithm reliably with all threshold distances smaller than 0.15. For thresholds $< 0.15$ we simply adapted our traditional comparison function, which checks all candidate near profiles emitted by the LSH step, accordingly. Furthermore, for the runs in Table 4.1 having $J_e$ listed in their configuration, we performed the false positive removal by computing the extended Jaccard distance.

As stated before, for all binaries that have a behaviorally identical program at time $T_c$, we recomputed the distance at time $t_e$. Figure 4.4 shows the distribution of these distances as a CDF. The x-axis of the diagram details the percentage of behaviorally identical programs while the y-axis specifies

the distance between two programs at time $t_e$ (i.e., after four minutes). One can see that in most parameter configurations around 90% of all pre-empted executables have a distance ¡ 0.3 after executing for four minutes. We saw in Figure 4.3 that executions of the same malware program can easily have a distance of 0.3. In fact, a threshold below 0.3 leads to a number of false positives for checkpoint times below 100 seconds. At the same time, a distance threshold of 0.3 causes no false positives, as can be seen in Figure 4.2. This is why these results are absolutely encouraging. They demonstrate that our algorithm works well with only a small number of serious distance deviations.

Looking at Figure 4.4 makes it clear that we get better results in the runs where we chose a checkpoint time of 60 seconds as opposed to 45 seconds. Moreover, we see that the extended Jaccard distance does not perform necessarily better. It is also quite intuitive that demanding a smaller distance threshold at time $T_c$ results in smaller distances at time $t_e$. At the same time, the effectiveness (i.e., number of pre-empted files) of our algorithm decreases with tighter distance thresholds as can be seen in Table 4.1. Thus, there is no single correct value for the checkpoint time and the distance threshold. When selecting these parameters, one has to take the requirements of the application into consideration. For our purposes, we believe that a checkpoint time of 60 seconds and and a distance threshold of 0.12 strike a good balance between reliability of the analysis results and efficiency of the algorithm.

To calculate the time saved by pre-empting an analysis run, we make use of average values for $t_a(b)$ and for $t_{pre-empted}(b)$. In case of our dynamic analysis system, the average analysis time for a program including the setup-time and post-processing amounts to 334 seconds. For pre-empted analysis runs, we have to add on average 24 seconds to the checkpoint time to account for the setup-time. As a consequence, we save on average 265 seconds with a checkpoint time of 45 seconds and 250 seconds when the checkpoint time is 60 seconds. A configuration of $T_c = 60s$ and $d = 0.12$ saves in total 190.8 hours of analysis time. In this time, we can perform 2,056 additional, full analysis runs on average.

## 4.5 Limitations

It is obviously possible that a malicious adversary crafts two files that appear to be behaviorally identical at checkpoint time $T_C$ but change their behavior afterward. In such a case, our algorithm would wrongly pre-empt the analysis of one file. There is no easy defense against this kind of attack since this is an intrinsic problem of dynamic analysis systems. A dynamic analysis system is evadable by programs that do not reveal their true behavior during the

short period where their behavior is monitored. While it is possible to defend against specific attacks (such as sleep operations), it is more difficult to find solutions in the general case.

## 4.6 Conclusion

In this chapter, we propose a novel approach for making the dynamic analysis of malicious programs more efficient. It drastically reduces the amount of time necessary for analyzing a set of malicious program. Our approach makes use of the fact that the huge number of new malware programs appearing each day is due to mutations of only a few malware programs. Therefore, we suggest a technique that avoids fully analyzing a program again if we have already analyzed this program (albeit different on the file level) once. We detect that a program is a polymorphic variation of an already analyzed binary by executing it for a short period of time. In a next step, we check whether the behavior seen in this period is almost identical to an already analyzed binary. If this is the case, we stop the currently ongoing analysis and instead return the existing analysis result.

We have empirically demonstrated that this technique works well in practice and that it is efficient. By leveraging locality sensitive hashing we avoid performing $n - 1$ comparisons for determining whether an almost identical program has already been analyzed. Moreover, our experiments show that for a set of 10,922 randomly chosen executable files, we were able to avoid the full analysis of 2,747 files (25.25%). This equals 190.8 hours of saved analysis time. In the future, we plan to actively use this technique in our dynamic analysis system because it helps us to analyze more of today's malicious programs.

In the following chapter, we discuss the contributions of our thesis in the context of other existing work.

# Chapter 5

# Related Work

## 5.1 Static vs Dynamic Analysis

Dynamic analysis system are not the only means to analyze malicious binaries. Systems based on static analysis (e.g., [35, 37, 40, 51]) also exist. Static analysis in this context means that no code is executed. Such techniques are less popular though because malware is usually well-protected against static techniques. In particular, today's malware programs leverage code obfuscation [55], code encryption and runtime packing [45] to make dis-assembly difficult. Since sophisticated static techniques rely on disassembling the binary in a first step, these techniques suffer from the fact that they cannot analyze the majority of malicious binaries. The biggest advantage of static techniques is their potential to reason about all possible execution paths of a (malware) program while dynamic analysis is limited to a single execution path. Moreover, implementations of static methods are fast compared to dynamic techniques because they do not depend on a program's execution. Thus, static techniques are often preferred when speed or scalability is an issue. For example, applications (e.g., virus scanners) running on a user's computer are more inclined to the use of static than dynamic techniques. One problem of static techniques is that they are undecidable in the general case. Consequently, one has to resort to approximations in many cases.

Dynamic analysis sees the instructions that are actually executed and so they are unaffected by code obfuscation, runtime packing or anti-debug techniques. Their main drawback is that they traditionally examine only a single execution path. Recently, Moser at al suggested in their paper [60] to enhance dynamic analysis by exploring multiple executions paths. Similarly, symbolic execution of programs as described in [31] was proposed with the goal of observing more than a single execution path and in particular to detect trigger-based behavior in malware.

In theory, also static techniques could profit from the fact that a large portion of today's malicious program landscape is composed of file-level variations of a small number of malware programs. Analog to our technique for improving the efficiency of dynamic analysis systems, an expensive static analysis run

could be avoided if it's possible to prematurely detect that an analysis of this malware program already exists.

## 5.2 Malware Analysis

Researchers have extensively studied the malware problem domain. One line of research has focused on the extent to which certain classes of malware have penetrated the Internet. For example, there have been studies that quantify the size of botnets [69], the number of executables infected with spyware [62], and the number of malicious web sites that launch drive-by downloads [68]. Another line of research deals with tools to collect and study malware. Here, researchers have introduced various forms of honeypots [22, 70], static analysis techniques [35], and dynamic monitoring tools [41, 74]. Finally, there are proposals to detect and remove malware once it has infected a machine, using either signature-based or behavior-based approaches.

While previous research has shed light on many aspects of malicious code, relatively little is known about the behavior of malicious programs once they infect a host. With behavior, we refer to the interaction of a program with the host operating system, other applications, or the network. Of course, a few popular malware families are very well-understood [59]. Also, folk wisdom associates with malware behavior programs that read and copy their own executables into the Windows system folder. Finally, network activity has been analyzed more thoroughly [58], possibly because it is more straightforward to collect and evaluate. However, there is little knowledge about general, host-based interactions that are characteristic for or common among a large and diverse set of different malware families. For example, we would like to know common mechanisms that malware binaries use to propagate, to make their programs persistent, and to evade detection. On one hand, such information is valuable to better understand the motives and goals of malware authors and to see the ways in which malicious code evolves over time. On the other hand, the information is crucial for the development of better behavior-based detection techniques.

The recent advances in the field of automated malware analysis (e.g., [31, 41, 60, 73, 74]) have created a rising interest in the automatic grouping of the analysis results (and reports) that are created. For this purpose, researchers have proposed supervised as well as unsupervised machine learning techniques. Because it is crucial that these techniques can process a large number of samples, their scalability is one of the decisive properties.

At the core of every system that aims to find malware families is the notion of similarity. Therefore, these systems need to solve two problems. First, they

need to find a suitable representation of a malware sample. Second, based on these representations, they need to compute a distance between two samples. In the literature, content-based and behavior-based comparison approaches have been proposed.

**Content-Based Analysis.** The first attempts to cluster malware samples were based on static analysis of the malware samples. In [43], the author proposes an automated virus classification system that works by first disassembling the binaries, and subsequently, comparing their basic code blocks. Other researchers have proposed to represent a malware program as a hexdump of its code segment, building a classification system on top of this [50]. In [40], Dullien proposes a system for comparing executables based on their control flow graph.

All content-based analysis approaches share the problem that they need to disassemble the binary. This is often difficult or even impossible, given that malware is frequently obfuscated and packed. Also, it is possible to write semantically-equivalent programs that have large difference in their code. Thus, it is possible for malware authors to thwart content-based similarity calculations.

**Behavior-Based Analysis.** Recently, Holz et al. [47] presented a system that classifies unknown malware samples based on their behavior. A significant limitation is that the system requires supervised learning, using a virus scanner for labeling the training set. Lee et al. developed a system for classifying malware samples that relies on system calls for comparing executables [52]. The scalability of the technique is limited; the system required several hours to cluster a set of several hundred samples. Also, the tight focus on system calls implies that the collected profiles do not abstract the observed behavior.

The approach that is closest to ours was presented by Bailey et al. [24]. Their proposed system abstracts from system call traces and clusters samples that exhibit similar behavior. Unfortunately, Bailey's system does not scale well (it requires to compute $O(n^2)$ distances), and, compared to our system, their generated behavioral profiles lack important information that we can obtain via a fine-grained analysis and behavioral abstraction. This results in a clustering that is less accurate.

Leita et al. [53] suggest classifying malware based on the epsilon-gamma-pi-mu model. In this model, additional information on how the malware is originally installed on the target system is considered for classification. This can include information on the exploit and exploit payload used to install the malware dropper, and on the way the dropper in turn downloads and installs the malware. Since in [53] the malware itself is characterized by simply using anti-virus names, this approach is complementary to the one described in this

thesis.

**Dynamic Data Tainting.** Taint analysis is a technique that has been extensively used in the field of computer security. For example, it has been successfully applied to the detection of exploits that hijack the control flow of a program and, in some cases, automatic signature generation against detected threats [39, 63, 67]. Similar to our approach, there are systems that employ tainting for extracting characteristic information flows from malware binaries. Yin et al. [74] extended Qemu with data tainting to capture system-wide information flows. Recently, dynamic taint analysis has been also used for the automatic analysis of network protocols [32, 71].

**Improving the efficiency.** Both clustering systems, as well, as our algorithm for detecting already analyzed programs (albeit different on the file level) have to compare different execution traces and define a notion of similarity. While clustering aims to find groups of behaviorally similar programs, we are only interested in finding a program's nearest neighbor. Finding a program's nearest neighbor is a step necessary to reach our goal of not analyzing binaries that have already been analyzed. We are performing this search based on monitoring the behavior for a more limited amount of time. Clustering systems, on the other hand, work of course on the whole execution trace of a binary.

# Chapter 6

# Conclusion

The root cause of many criminal activities on the Internet are malicious programs. Trojans, viruses, bots, etc. give miscreants a wide range of possibilities for conning unsuspecting Internet users. For this reason, we see a huge number of new malware programs appearing each day. This number has grown dramatically over the last few years, and it will continue to grow in all likelihood. At the time of writing this thesis, one has to assume that around 35,000 new malicious binaries appear each day. Obviously, A/V companies cannot analyze such a high number of files manually. They need automated tools for verifying whether all of the suspicious files they receive are, in fact, malicious or not. Because of the limits of static analysis [61], this prompted researchers and practitioners to develop automated, dynamic malware analysis systems.

In this dissertation, we presented several novel techniques to improve the large-scale dynamic analysis of malware. First, we shed light on common malware behaviors. To this end, we evaluated the analysis results of almost one million malware samples. Second, we presented a clustering algorithm that allows the grouping of malware samples according to their behavior. Third, we proposed and evaluated a technique for improving the efficiency of dynamic malware analysis.

To demonstrate the usefulness of our techniques, we have implemented prototype systems and tested them in real-world scenarios. The results show that our approaches work well in practice. All of our prototype systems are available online [1].

# Bibliography


[1] ANUBIS. http://anubis.seclab.tuwien.ac.at, 2009.

[2] Article about Pinch in the Washington Post. http://blog.washingtonpost.com/securityfix/2007/11/new_malware_defeats_sitekey_te.html?nav=rss_blog, 2009.

[3] AVG Virus Database - Mabezat. http://www.avg.com/virbase?nam=win32/mabezat, 2009.

[4] Boost.Python C++ library. http://www.boost.org, 2009.

[5] CWSandbox. http://www.cwsandbox.org/, 2009.

[6] Description of Allaple by F-Secure. http://www.f-secure.com/v-descs/allaple_a.shtml, 2009.

[7] Description of Trojan-PWS.Win32.LdPinch by Mcafee. http://vil.nai.com/vil/content/v_100539.htm, 2009.

[8] Description of Virus.Win32.Virut by F-Secure. http://www.f-secure.com/v-descs/virus_w32_virut.shtml, 2009.

[9] Description of Virus.Win32.Virut by Mcafee. http://vil.nai.com/vil/content/v_154029.htm, 2009.

[10] F-Secure Malware Information Pages - Allaple.A. http://www.f-secure.com/v-descs/allaple_a.shtml, 2009.

[11] Forum Posting - Detection of Sandboxes. http://www.opensc.ws/snippets/3558-detect-5-different-sandboxes.html, 2009.

[12] IDA Pro. http://www.hex-rays.com/idapro/, 2009.

[13] JoeBox. http://www.joebox.org, 2009.

[14] Kasperksy News - Kaspersky Lab forecasts ten-fold increase in new malware for 2008. http://www.kaspersky.com/news?id=207575629, 2009.

[15] MWCollect. http://www.mwcollect.org/, 2009.

[16] Norman Sandbox. `http://www.norman.com/microsites/nsic/`, 2009.

[17] Shadowserver. `http://shadowserver.org/wiki/`, 2009.

[18] Signature Update by Mcafee that includes a new LdPinch variant. `http://www.f-secure.com/v-descs/allaple_a.shtml`, 2009.

[19] ThreatExpert. `http://www.threatexpert.com/`, 2009.

[20] Virus Total. `http://www.virustotal.com/`, 2009.

[21] David Arthur and Sergei Vassilvitskii. How slow is the k-means method? In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*, pages 144–153, New York, NY, USA, 2006. ACM.

[22] Paul Baecher, Markus Koetter, Thorsten Holz, Maximillian Dornseif, and Felix C. Freiling. The nepenthes platform: An efficient approach to collect malware. In Diego Zamboni and Christopher Kruegel, editors, *RAID*, volume 4219 of *Lecture Notes in Computer Science*, pages 165–184. Springer, 2006.

[23] M Bailey. Malware clustering results. `http://www.eecs.umich.edu/~mibailey/malware/`, 2009.

[24] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, September 2007.

[25] U. Bayer, P. Milani Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Symposium on Network and Distributed System Security (NDSS)*, 2009.

[26] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. Insights Into Current Malware Behavior. In *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET), Boston, MA*, 2009.

[27] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. Improving the Efficiency of Dynamic Malware Analysis. In *25th Symposium On Applied Computing (SAC), Track on Information Security Research and Applications, Lusanne, Switzerland*, 2010.

[28] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A Tool for Analyzing Malware. In *15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, April 2006.

[29] F. Bellard. Qemu, a Fast and Portable Dynamic Translator. In *Usenix Annual Technical Conference*, 2005.

[30] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Comput. Netw. ISDN Syst.*, 29(8-13):1157–1166, 1997.

[31] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Book chapter in "Botnet Analysis and Defense", Editors Wenke Lee et. al.*, 2007.

[32] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. In *Proceedings of ACM Conference on Computer and Communication Security*, October 2007.

[33] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *GI SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2008.

[34] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. ACM, 2002.

[35] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Usenix Security Symposium*, 2003.

[36] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 5–14, New York, NY, USA, 2007. ACM.

[37] Mihai Christodorescu, Somesh Jha, Sanjit Seshia, Dawn Song, and Randal Bryant. Semantics-Aware Malware Detection. In *IEEE Symposium on Security and Privacy*, 2005.

[38] R. Cilibrasi and P. Vitányi. Complearn version 1.15. `http://www.complearn.org/`, 2009.

[39] J. Crandall and F. Chong. Minos: Architectural support for software security through control data integrity. In *International Symposium on Microarchitecture*, 2004.

[40] Thomas Dullien and Rolf Rolles. Graph-based comparison of Executable Objects. In *In Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, June 2005.

[41] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Proceedings of USENIX Annual Technical Conference*, June 2007.

[42] Peter Ferrie. Attacks on more virtual machine emulators.

[43] Marius Gheorghescu. An Automated Virus Classification System. In *Virus Bulletin conference*, 2005.

[44] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *16th Usenix Security Symposium*, 2007.

[45] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A study of the packer problem and its solutions. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 98–115, Berlin, Heidelberg, 2008. Springer-Verlag.

[46] Taher H. Haveliwala, Aristides Gionis, and Piotr Indyk. Scalable techniques for clustering the web. In *WebDB (Informal Proceedings)*, pages 129–134, 2000.

[47] Thorsten Holz, Carsten Willems, Konrad Rieck, Patrick Duessel, and Pavel Laskov. Learning and Classification of Malware Behavior. In *Fifth Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 08)*, June 2008.

[48] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. of 30th STOC*, pages 604–613, 1998.

[49] P. Jaccard. The Distribution of Flora in the Alpine Zone. *The New Phytologist*, 11(2):37–50, 1912.

[50] J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, 2006.

[51] Christopher Kruegel, William Robertson, and Giovannni Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Annual Computer Security Application Conference (ACSAC)*, 2004.

[52] Tony Lee and Jigar J. Mody. Behavioral Classification. In *EICAR Conference*, 2006.

[53] Corrado Leita and Marc Dacier. SGNET: a worldwide deployable framework to support the analysis of malware threat models. In *EDCC 2008, 7th European Dependable Computing Conference, May 7-9, 2008, Kaunas, Lituania*, 2008.

[54] M Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, New York, second edition, 1997.

[55] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, New York, NY, USA, 2003. ACM.

[56] L.Kaufman and P.J. Rousseeuw. *Finding groups in data: An introduction to cluster analysis*. New York: John Wiley & Sons, 1990.

[57] J. B. Macqueen. Some methods of classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[58] Niels Provos Michalis Polychronakis, Panayiotis Mavrommatis. Ghost Turns Zombie: Exploring the Life Cycle of Web-based Malware. In *Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[59] D. Moore, C. Shannon, and k claffy. Code-Red: A case study on the spread and victims of an Internet worm. In *ACM Internet Measurement Workshop*, 2002.

[60] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 231–245, 2007.

[61] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of Static Analysis for Malware Detection. In *ACSAC*, pages 421–430. IEEE Computer Society, 2007.

[62] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A Crawler-based Study of Spyware on the Web. In *Network and Distributed Systems Security Symposium (NDSS)*, 2006.

[63] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[64] Travis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments.

[65] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Comput. Networks*, 31(23-24):2435–2463, 1999.

[66] Microsoft PECOFF. Microsoft Portable Executable and Common Object File Format Specification. `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`, 2009.

[67] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[68] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All Your iFrames Point to Us. In *Usenix Security Symposium*, 2008.

[69] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *Internet Measurement Conference (IMC)*, 2006.

[70] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[71] G. Wondracek, P. Milani Comparetti, C. Kruegel, and E. Kirda. Automatic Network Protocol Analysis. In *15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[72] Tarkan Yetiser. Polymorphic Viruses - Implementation, Detection, and Protection. `http://vx.netlux.org/lib/ayt01.html`, 2009.

[73] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[74] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, New York, NY, USA, 2007. ACM.

# Lebenslauf

| | |
|---|---|
| 21.08.1981 | geboren in Wien, Österreich |
| 09/1987 - 06/1991 | Besuch der Volksschule Pressbaum |
| 09/1991 - 06/1999 | Gymnasium Sacre Coeur Pressbaum |
| 10/1999 - 02/2004 | Bakkalaureatsstudium “Software & Information Engineering” an der TU Wien |
| 03/2004 - 12/2005 | Magisterstudium “Software Engineering & Internet Computing” an der TU Wien (mit Auszeichnung bestanden), Diplomarbeit “TTAnalyze: A Tool for Analyzing Malware” |
| 12/2006 - 12/2009 | PhD Studium Informatik an der TU Wien |