FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Detecting Environment-Sensitive Malware

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Martina Lindorfer
Matrikelnummer 0626770

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Priv.-Doz. Dipl.-Ing. Dr. Engin Kirda
Mitwirkung: Dr. Paolo Milani Comparetti, Dipl.-Ing. Clemens Kolbitsch

Wien, 11.04.2011

(Unterschrift Verfasserin) (Unterschrift Betreuer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Detecting Environment-Sensitive Malware

MASTER'S THESIS

for the degree of

Master of Science

in the field of

Software Engineering & Internet Computing

submitted by

Martina Lindorfer
Matriculation Number 0626770

at the
Faculty of Informatics, Vienna University of Technology

Supervision
Supervisor: Priv.-Doz. Dipl.-Ing. Dr. Engin Kirda
Assistance: Dr. Paolo Milani Comparetti, Dipl.-Ing. Clemens Kolbitsch

Vienna, April 11th, 2011

Declaration
(Erklärung)

Martina Lindorfer
Margaretenstrasse 145
1050 Vienna, Austria

I hereby declare that I am the sole author of this thesis, that I have exhaustively specified all sources and resources used, and that parts of this thesis - including tables, maps and figures - if taken from other works or from the Internet, whether copied literally or by sense, all have been marked as replications including a citation of the source.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Vienna, April 11th, 2011 _____

Abstract

Malware poses one of the Internet's major security threats today. Due to the vast amount of new malware samples emerging every day, researchers and Anti-Virus vendors rely on dynamic malware analysis sandboxes such as Anubis to automatically analyze malicious code in a controlled environment. In order to evade detection by these sandboxes, environment-sensitive malware aims to differentiate the analysis sandbox from a real user's environment. In the absence of an "undetectable", fully transparent analysis sandbox, defense against sandbox evasion is mostly reactive: Sandbox developers and operators tweak their systems to thwart individual evasion techniques as they become aware of them, leading to a never-ending arms race.

In this thesis we present DISARM, a system that automates the screening of malware for evasive behavior using different analysis sandboxes. We present novel techniques that normalize malware behavior across these analysis sandboxes and detect malware samples that exhibit semantically different behavior. We further present a light-weight monitoring technology that is portable to any Windows XP environment. Nevertheless, our techniques are compatible with any monitoring technology that can be used for dynamic malware analysis and are completely agnostic to the way that malware achieves evasion. In a large-scale evaluation on real-world malware samples we demonstrate that DISARM can accurately detect evasive malware, leading to the discovery of previously unknown evasion techniques.

Kurzfassung

Malware stellt eines der größten Sicherheitsrisiken im Internet dar. Durch die enorme Anzahl an neuer Malware, die täglich erscheint, benötigen Forscher und Hersteller von Anti-Viren Software Unterstützung durch dynamische Analyse-Sandboxen wie zum Beispiel Anubis. Diese Sandboxen ermöglichen die automatisierte Analyse von Malware in einer kontrollierten Umgebung. Sogenannte "umgebungs-sensitive" Malware versucht eine solche Sandbox vom System eines echten Benutzers zu unterscheiden und somit die Analyse und Erkennung zu umgehen. In Abwesenheit einer "unerkennbaren", vollkommen transparenten Analyse-Sandbox ist die Abwehr solcher Umgehungsmethoden hauptsächlich reaktiv: Hersteller und Betreiber von dynamischen Sandboxen modifizieren ihre Systeme um Umgehungsmethoden zu verhindern sobald diese bekannt werden. Dies führt wiederum zu einem endlosen Wettrüsten zwischen den Entwicklern von Malware-Entwicklern von Analyse-Sandboxen.

In dieser Arbeit präsentieren wir DISARM, ein System, das Malware automatisch in mehreren Analyse-Sandboxen auf Umgehungsmethoden überprüft. Wir präsentieren neue Methoden zur Normalisierung von Malware-Verhalten in verschiedenen Sandboxen und zur Erkennung von semantisch unterschiedlichem Verhalten. Des Weiteren entwickeln wir eine Monitoring-Technologie zur Analyse von Malware mit geringem Overhead in jeder beliebigen Windows XP Umgebung. Nichtsdestotrotz sind unsere Methoden mit jeder Monitoring-Technologie zur dynamischen Analyse von Malware kompatibel. Zusätzlich funktionieren unsere Methoden unabhängig von der Art und Weise mit der Malware versucht die Analyse zu umgehen. Wir unterziehen DISARM einer umfangreichen Evaluierung, anhand welcher wir die Genauigkeit in der Erkennung von Umgehungsmethoden in realer Malware demonstrieren. Wir erkennen damit neuartige Methoden, mit denen Malware die Analyse in dynamischen Analyse-Sandboxen umgeht.

Acknowledgements

I am deeply grateful to my advisor Engin Kirda for the opportunity to work on this thesis. I would further like to thank Paolo Milani Comparetti and Clemens Kolbitsch for their countless suggestions and guidance during the thesis. It was a pleasure to work at the Secure Systems Lab and I would like to thank the whole team and especially Matthias Neugschwandtner for his inputs on the loading of Windows kernel drivers.

This thesis further would not have been possible without my parents Rita and Hans Lindorfer and their constant support and encouragement throughout my life. Lastly, my thanks go to my friends, first and foremost Lukas Reiter and Clemens Peither, who helped me keep my focus and my sanity. Kamsahamnida to all of you!

Contents

CHAPTER 1

Introduction

1.1 Motivation

As the Internet is becoming an important part of people's everyday life, Internet users face increasing security threats posed by malware. Malware, short for malicious software, refers to software that infects a system without the user's consent and with the intent to cause damage or steal private information. Depending on the method of infection and the exhibited behavior, instances of malware are classified as viruses, worms, Trojan Horses, rootkits, spyware, etc.

Thousands of new malware samples surface every day. In 2010 McAfee Labs[1] identified more than 20 million new malware samples resulting in an average of 55,000 new instances of malware identified per day [35]. As Fig. 1.1 illustrates, this number has increased dramatically over the past few years. Similarly, PandaLabs[2] reported more than 60,000 new malware samples per day in 2010 and an average of more than 73,000 in the first quarter of 2011 [51].

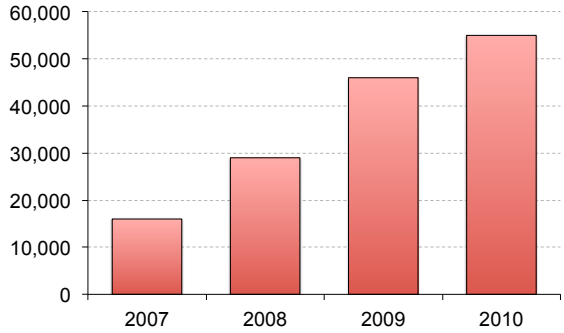

Figure 1.1: Average number of malware samples identified by McAfee Labs per day [34, 35].

[1] http://www.mcafee.com/us/mcafee-labs.aspx
[2] http://www.pandasecurity.com/homeusers/security-info/pandalabs/

Due to this vast amount of new samples emerging every day, researchers and Anti-Virus vendors rely on automated analysis tools in order to distinguish malicious from benign code. Dynamic malware analysis sandboxes (DMAS) such as Anubis [13] and CWSandbox [67] provide means to automatically execute a sample in a controlled environment and monitor its behavior. Based on the report of a DMAS, analysts can quickly determine whether a sample is benign, a variant of an already known malware family or whether a sample exhibits unseen malicious behavior and therefore requires further manual analysis. In addition to public-facing services such as tools operated by security researchers[3] and companies[4,5], which are freely available to the public, private malware analysis sandboxes are operated by a variety of security companies such as Anti-Virus vendors.

With the rise of a flourishing underground economy, malware authors become strongly motivated by financial gains [71]. This leads to more sophisticated malware samples trying to stay under the radar of researchers and Anti-Virus vendors in order to avoid detection and increase profits. To this end, malware samples try to discover when they are running inside an analysis sandbox instead of on a real user's system and evade the analysis by refusing to perform malicious activities. As a DMAS is usually a combination of a monitoring technology (the analysis environment) and a specific configuration of a Windows environment in which the malware is executed, malware samples can detect the presence of a sandbox either by detecting characteristics of the analysis environment or by identifying configuration characteristics of the Windows environment. We therefore call such evasive samples "*environment-sensitive*". As we will discuss in Chapter 2, sandbox evasion is not a theoretical problem. We will present a number of concrete examples for evasion techniques, which malware samples have used to evade Anubis in the past.

One approach to defeating sandbox evasion is to try to build a transparent analysis environment that is indistinguishable from a real, commonly used production environment. However, research has shown that building such an environment is fundamentally unfeasible [23]. Even if such a technology were available, a specific sandbox installation could be detectable based on the configuration characteristics [11]. Another approach relies on running a sample in multiple analysis sandboxes to detect deviations in behavior that may indicate evasion [9, 16, 29, 32]. As sandbox developers and operators become aware of individual evasion techniques, they can tweak their systems to thwart these techniques, leading to a never-ending arms race.

In order to provide more automation in this arms race, we require scalable tools to automatically screen large numbers of malware samples for evasive behavior, regardless of the type of evasion techniques they employ. Thus, we propose a system called DISARM: *DetectIng Sandbox-AwaRe Malware*. DISARM executes malware samples in multiple analysis sandboxes and is able to detect differences in behavior regardless of their cause. We propose a number of techniques for normalizing and comparing behavior in different sandboxes, which discard spurious differences that do not correspond to semantically different behavior. DISARM is therefore completely agnostic to the way that malware may perform sandbox detection. Furthermore, it is also largely agnostic to the monitoring technologies used by the analysis sandboxes. For

[3] Anubis: Analyzing Unknown Binaries (`http://anubis.iseclab.org/`)
[4] SunbeltLabs (`http://www.sunbeltsecurity.com/sandbox/`)
[5] ThreatExpert (`http://www.threatexpert.com/`)

this thesis, we implement a light-weight execution monitoring system that can be applied to any Windows XP environment. Nevertheless, any monitoring technology that can detect persistent changes to a system's state at the operating system level, such as modifications of files or the registry, can take advantage of our techniques. DISARM does not, however, automatically identify the root cause of a divergence in behavior. Samples we detect could therefore be further analyzed using previously proposed approaches to automatically determine how they evade analysis [9, 29].

We evaluated DISARM against more than 1,500 malware samples in four different analysis sandboxes. We thereby identified over 400 samples that exhibit semantically different behavior in at least one of the sandboxes. By further investigation of these samples we discovered a number of previously unknown analysis evasion techniques. We can use our findings to make our analysis sandboxes resistant against these kinds of evasion attacks in the future. We further discovered issues with the software configuration of our analysis sandboxes that, while unrelated to evasion, nonetheless prevent us from observing some malicious behavior.

1.2 Approach

As illustrated in Fig. 1.2, DISARM works in two phases. In the execution monitoring phase (described in Chapter 3), we execute a malware sample in a number of analysis sandboxes. For the purpose of this thesis, we define an analysis sandbox as the combination of a monitoring technology with a system image, i.e. a specific configuration of a Windows operating system on a (possibly virtual) disk. DISARM uses two different monitoring technologies. The first is the out-of-the-box monitoring technology Anubis. The second is a custom-built in-the-box monitoring technology implemented as a Windows kernel driver. Additionally, DISARM also uses different Windows installations with different software configurations.

The output of the execution monitoring phase is the malware's behavior represented as a number of behavioral profiles (one behavioral profile for each execution). In the behavior comparison phase (described in Chapter 4), we first normalize these behavioral profiles. This normalization eliminates spurious differences that are not caused by a malware's behavior but result from environmental differences of the sandboxes. We then compute the distances between each pair of normalized behavioral profiles. Finally, we combine these distances into an evasion score that we compare against a threshold to determine whether the malware displays semantically different behavior in any of the sandboxes.

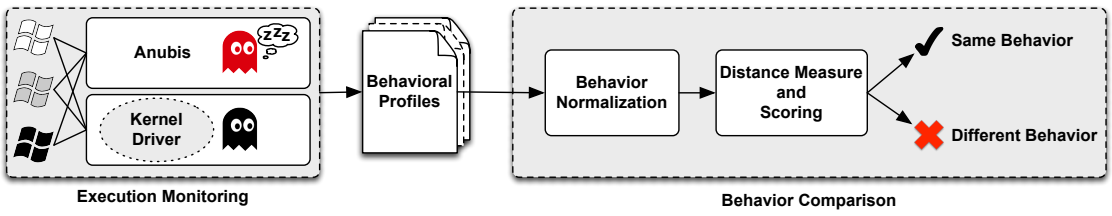

Figure 1.2: System architecture of DISARM.

A sample may actively evade the analysis in one of the sandboxes, e.g. by detecting the presence of Anubis and sleeping until the analysis times out, while displaying malicious activity in sandboxes using our kernel driver. Samples also may simply display different behavior that is influenced by some characteristic of the Windows environment, such as the presence of a specific application or configuration setting. In either case, the results of DISARM provide useful information to the operators of a malware analysis sandbox, who then can develop countermeasures for specific evasion techniques or tweak the software configuration of their sandbox in order to observe a wider variety of malware behavior.

1.3 Overview

The rest of this thesis is organized as follows: First, in Chapter 2 we discuss the state of the art of malware analysis, known analysis evasion techniques and countermeasures that aim to prevent or detect analysis evasion. We then discuss the details of our approach to detecting environment-sensitive malware in Chapter 3 and 4. Chapter 3 provides the implementation details of the execution monitoring component, while Chapter 4 describes our behavior comparison techniques. We evaluate our approach in Chapter 5 on a small training dataset and a large scale test dataset of over 1,500 malware samples. We discuss the quantitative results of our experiments as well as the evasion techniques we discovered. In Chapter 6, we discuss the limitations of our approach and propose improvements for future work. Finally, we conclude this thesis in Chapter 7 with a summary of our work.

CHAPTER 2

State of the Art

In this chapter we present current malware analysis techniques and methods employed by malware to evade analysis. We further discuss approaches to detect and prevent analysis evasion. As our approach includes the comparison of malware behavior, we also discuss related work concerned with finding similarities in malware behavior.

2.1 Malware Analysis

There are two general approaches to malware analysis: The first is static analysis, which is the process of inspecting a malware sample's binary without executing it. Static analysis can cover all possible execution paths of a malware sample. However, malware authors can hamper static analysis by employing code obfuscation techniques [44].

Therefore, researchers and Anti-Virus vendors widely deploy dynamic analysis, which relies on monitoring a malware sample's behavior while executing it in a controlled environment. The Secure Systems Lab at the Vienna University of Technology has been offering dynamic malware analysis with Anubis [1, 13] as a free service since February 2007. This service has over 2,000 registered users, has received submissions from 200,000 distinct IP addresses and has already analyzed over 10,000,000 malware samples. Other dynamic analysis tools are CWSandbox [67], Joe Sandbox [3], ThreatExpert [6] and Norman Sandbox [4]. Additionally to providing detailed reports about the malware's behavior during analysis, various further applications have been proposed: Dynamic analysis can be used to obtain unpacked code [31, 38], detect botnet command and control (C&C) servers [62] and generate signatures for C&C traffic [53] as well as remediation procedures for malware infections [49].

Unlike static analysis, dynamic analysis observes only one execution path and therefore suffers from incomplete code coverage. For example, behavior, which is triggered by user input, by the existence of certain files or by a specific date, is only observable when those trigger conditions are met. Furthermore, malware samples may alter their behavior or exit when they detect the dynamic analysis sandbox. Thus, such samples evade the analysis by displaying no observable malicious activity in the sandbox.

2.2 Evasion Techniques

Dynamic analysis sandboxes commonly operate in a virtualized or emulated environment and are therefore vulnerable to a number of evasion techniques that identify the presence of the virtual machine [21, 22, 59] or the emulator [21, 22, 50, 54]. As Anubis is based on an instrumented Qemu [15] emulator, we focus on evasion techniques targeting emulators, but also give a few examples of virtualization detection techniques. Furthermore, previous work also observed application-level detection of characteristic features of a sandbox, such as the presence of specific processes or executables in the system [11].

Chen et al. [16] proposed a taxonomy of approaches that can be used by malware to detect analysis sandboxes. They grouped evasion techniques by the system abstractions at which they operate: *Hardware*, *Environment*, *Application* and *Behavior*. Table 2.1 lists a number of Anubis evasion techniques that have been observed over the years, classified according to an extended version of this taxonomy. We added one abstraction (*Network*) and two classes of artifacts (*connectivity* and *unique identifier*) to this taxonomy. The connectivity class is required because the network configuration of a DMAS faces a trade-off between transparency and risk. It is typically necessary to allow malware samples some amount of network access to be able to observe interesting behavior. On the other hand, DMAS operators need to prevent the samples from causing harm to the rest of the Internet. A malware sample, however, may detect that it only has limited access to the Internet and may refuse to function. The unique identifier class is required because many of the evasion techniques that have been used against Anubis are not targeted at detecting the monitoring technology used by Anubis, but characteristics of the specific Windows installation used by the publicly accessible Anubis service.

Evasion techniques targeting the *Hardware* level leverage the fact that virtual machines and emulators often create specific hardware devices [16]. These devices can either be identified by well-known manufacturer prefixes such as `VMWARE VIRTUAL IDE HARD DRIVE` and `QEMU HARDDISK` or by specific drivers that are not present in a real user's system. Experiments with Anubis [11] showed that malware samples also detect Qemu by querying the unique serial number of the hard disk or the MAC address of the network card.

The *Environment* abstraction level comprises evasion techniques targeting memory and execution artifacts that differ between native and virtual or emulated systems. Identifiable memory artifacts include for example the communication channel between virtual machine guests and the Virtual Machine Manager of VMWare [16, 37]. Joanna Rutkowska discovered that VMWare relocates the interrupt descriptor table (IDT) to an identifiable memory address and therefore can be detected by executing a single non-privileged CPU instruction, the so-called "red pill" [59]. Attackers can further differentiate between a real and an emulated CPU by execution artifacts such as the presence or absence of CPU implementation bugs [54]. For example, a bug in the handling of the AAM instruction[1] incorrectly causes an integer division-by-zero in Qemu [48]. Paliari et al. [50] used fuzzing to automatically generate "red pills" that are capable of differentiating a CPU emulator from a physical CPU.

Other, less sophisticated evasion techniques operate on the *Application* level by identifying tools that are installed or executed in an analysis sandbox. Attackers can check the filesystem or

[1] ASCII adjust AX after multiply

Table 2.1: Anubis evasion techniques according to taxonomy [16] (extended).

Abstraction	Artifact	Test
Hardware	device	QEMU hard disk string
	driver	-
	unique identifier	disk serial number [11], MAC address
Environment	memory	-
	execution	"red pills" [50]
		AAM instruction emulation bug [48]
Application	installation	`C:\exec\exec.exe` present
		username is "USER" [11]
		executable name is "sample.exe" [11]
	execution	`popupKiller.exe` process running
	unique identifier	Windows product ID [11]
		computer name [11]
		volume serial number of system drive
		hardware GUID
Behavior	timing	query timing information [52]
Network	connectivity	get current time from Yahoo homepage
		check Gmail SMTP server response string
	unique identifier	server-side IP address check [30, 33, 68]

the registry for strings and artifacts that identify an analysis sandbox [16]. For example, known Anubis evasion techniques check for the presence of the analysis-specific `popupKiller.exe` application or the analysis daemon `C:\exec\exec.exe` or compare the Windows username or the name of the analyzed sample to known values [11]. The Windows installation used for analysis further contains various installation-specific identifiers, that can be read from the registry, such as the Windows product ID, the computer name, the volume serial number of the hard drive and the GUID of the current hardware profile.

Evasion techniques that target the *Behavior* level commonly take advantage of the fact that the execution of some instructions takes longer in an emulated environment than on a native system [16]. By continuously executing such instructions and comparing timing information (such as the Time-Stamp Counter) before and afterwards, malware samples can detect that they are being analyzed [52].

Lastly, malware samples can detect the sandbox on the *Network* level by detecting whether they run in a simulated network environment or if they have full access to the Internet [68]. The experience with Anubis showed that samples for example connect to well-known sites such as the Yahoo homepage to get the current time, or check the response string of well-known SMTP servers such as Gmail in order to detect network access restrictions. Yoshioka et al. [68] further observed that public-facing analysis sandboxes such as Anubis are particularly vulnerable to detection because attackers can probe the sandbox by submitting malware samples specif-

ically designed to perform reconnaissance. Such samples can read out characteristics of the analysis sandbox and then use the analysis report produced by the sandbox to leak the results to the attacker. These characteristics can later be tested by malware that wishes to evade analysis. According to their experiments, however, because of sharing of malware samples between sandbox operators, private sandboxes may also be vulnerable to reconnaissance, as long as they allow executed samples to contact the Internet and leak out the detected characteristics. Furthermore, attackers can send probes to publicly available analysis sandboxes to leak their IP addresses [30, 33]. Malware samples then can blacklist these IP addresses and refuse to perform malicious activities on these hosts.

2.3 Evasion Countermeasures

In order to mitigate analysis evasion, sandbox operators can either prevent sandbox detection by using "undetectable" analysis platforms or try to detect evasion techniques. Once a specific evasion technique is known, it is usually relatively straightforward to render it ineffective. For instance the developers of Anubis mitigated some of the techniques in Table 2.1 by randomizing a number of unique identifiers before each analysis run.

Transparent Monitoring

To prevent sandbox detection, researchers have tried to develop transparent, i.e. undetectable, analysis platforms. Examples include Cobra [66], which provides a stealth supervised code execution environment based on dynamic code translation, and Ether [17], which uses hardware assisted virtualization to implement a transparent out-of-the-box malware analysis platform. However, Garfinkel et al. [23] have argued that perfect transparency against timing attacks cannot be achieved, particularly if a remote timing source (such as the Internet) is available. Pek et al. [52] have succeeded in defeating Ether using a local timing attack.

The approach of Paleari et al. [50] can be used to make existing emulation-based analysis sandboxes more transparent. They used fuzzing to automatically generate "red pills" capable of detecting emulated execution environments. Therefore, they can proactively find emulation detection methods and fix emulator bugs before malicious code can exploit them.

Martignoni et al. [39] proposed to observe malware in more realistic execution environments by distributing the execution between a security lab and multiple real user's hosts. They thereby improved analysis coverage and were able to observe user input that triggers malicious behavior.

Evasion Detection

So far, evasion techniques used by malware to thwart Anubis analysis have been detected mostly "by accident". The first instance of Anubis evasion was a packer called `OSC Binder` that was released in September 2007 and advertised "anti-Anubis" features. Since then, samples that evade Anubis have been pointed out by Anubis users, while a few Anubis evasion techniques have been discussed in hacker forums and security blogs. In a few instances the Anubis developers have made more deliberate efforts to identify evasion techniques, such as obtaining "red

pills" as proposed in [50] and fixing the bugs they identified. In 2009 Bayer et al. [11] evaluated almost one million malware samples in order to study trends in malware behavior. They detected Anubis checks in 0.03 % of the evaluated malware samples, but expected the numbers to be higher. As Anubis works at API and system call level, they found it difficult to observe analysis evasion techniques targeting CPU instructions.

Chen et al. [16] analyzed malware samples in different execution environments (native machine, virtual machine and debugger). They compared a single execution trace obtained from each of the environments and considered any difference in persistent behavior to indicate evasion. In their experiments, 40 % of samples showed less malicious behavior with a debugger and 4 % of samples exhibited less malicious behavior in a virtual machine. However, as we will show in our evaluation in Section 5.4, this approach might lead to a high number of samples incorrectly classified as evasive.

Lau et al. [37] focused on virtual machine detection and employed a dynamic-static tracing system to identify evasion techniques in packers. According to their experiences 2.13 % of samples were able to detect the presence of the virtual machine. For our purposes, we do not give special consideration to packers and consider their behavior as part of a malware sample's behavior. However, we might evaluate evasion techniques applied in packers in future work, as unpacking routines often contain anti-analysis checks.

Moser et al. [43] explore multiple execution paths in Anubis to provide information about triggers for malicious actions. Kang et al. [32] use malware behavior observed in a reference platform to dynamically modify the execution environment in an emulator. They can thereby identify and bypass anti-emulation checks targeted at timing, CPU semantics and hardware characteristics. Balzarotti et al. [9] proposed a system that replays system call traces recorded on a real host in an emulator in order to detect evasion based on CPU semantics or on timing. Differential slicing [29] is able to find input and environment differences that lead to a specific deviation in behavior. The deviation that is to be used as a starting point, however, has to be identified manually.

In contrast to these techniques, DISARM is agnostic to the type of evasion methods used in malware as well as to the monitoring technologies employed. Thus, we propose DISARM for the automated screening of malware samples for evasive behavior. Samples detected by DISARM then could be further processed with the tools presented in [43, 32, 9, 29] to automatically identify the employed evasion techniques.

2.4 Behavior Comparison

One field of application for behavior comparison is malware clustering and classification. Malware clustering tries to discover classes of malware exhibiting semantically similar behavior, while classification assigns unknown malware samples to known clusters of behavior [56]. Bailey et al. [8] described malware behavior as persistent state changes in a behavioral fingerprint. They combined these fingerprints into clusters of similar behavior by using the normalized compression distance to compare behavior. Bayer et al. [10] proposed a scalable clustering technique that generalizes execution traces into behavioral profiles and identifies clusters of similar behavior based on locality sensitive hashing of these profiles. Rieck et al. [55] used machine learning

to classify malware samples based on behavioral patterns and later extended that approach with an automatic framework to perform clustering and classification [56].

Hu et al. [25] proposed a system to automatically classify samples as malicious by comparing a sample's behavior to a database of known malicious behavior. They describe malware behavior as function-call graphs and calculate the similarity between malware behavior by performing a nearest neighbor search on these graphs based on a graph edit-distance metric.

Bayer et al. [12] further compared malware behavior from different executions in Anubis to improve the efficiency of the dynamic analysis by reducing the overall analysis time. They suggested to compare the behavior of a sample during analysis to cached reports containing the behavior of already analyzed samples. Their system can thereby determine whether a sample under analysis was already analyzed before and in this case provides the cached report rather than finishing the analysis. They based the comparison on the same behavior representation as our approach, but had to implement a rather high threshold for the similarity of behavior due to execution-specific artifacts and a lack of behavior normalization.

CHAPTER 3

Execution Monitoring

In this chapter we briefly describe the Windows interfaces that can be used to monitor the execution of a program (see Section 3.1). We then focus on the two different monitoring technologies, both implemented in C and C++, which we use to analyze malware behavior.

The first is Anubis, which is an "out-of-the-box" monitoring technology that captures an executable's behavior from outside the Windows environment using an instrumented full system emulator (see Section 3.2). The second system uses "in-the-box" monitoring based on system call interception from inside the Windows environment (see Section 3.3). The idea is that by using two completely different monitoring technologies we are able to reveal sandbox evasion techniques that target a specific monitoring technology. Furthermore, we employ sandboxes that use different Windows environments in order to detect evasion techniques that rely on application and configuration characteristics to identify analysis sandboxes.

3.1 Monitored APIs

Microsoft Windows provides interfaces to system functions in application programming interfaces (APIs) at different levels of abstraction. Calls to these functions represent a program's communication with the environment, such as accessing file resources or creating processes. Both monitoring technologies observe the execution of a malware sample by intercepting function calls to interfaces provided by the Windows API as well as the Windows Native API.

Windows API

The Windows API is a collection of user-mode routines providing interfaces to core functionalities, such as basic functions for accessing resources, user interface functions, networking and functions for managing Windows services. As it is well documented in the Platform Software Development Kit (SDK) [41, 42] and is consistent across different Windows versions, it is the preferred way to implement an application's interaction with the operating system.

Windows Native API

The Windows Native API provides the system call interface callable from user space. The Native API can change between different Windows and even service pack versions. Furthermore, the most complete documentation is only available from unofficial sources [45, 46] because it is not intended to be called directly by applications. Legitimate Windows applications commonly call the Windows API which then forwards these calls to the Windows Native API. Malware, however, is likely to call the Native API directly to circumvent analysis tools which are only monitoring the Windows API [20].

Execution Flow

As an example, we illustrate the complete execution flow for writing to a file in Fig. 3.1. When an user space application calls `WriteFile`, which is provided by the Windows API located in `kernel32.dll`, the call is forwarded to the interface of `NtWriteFile` provided by the Windows Native API in `ntdll.dll`. As `NtWriteFile` is implemented in the kernel image `ntoskrnl.exe`, the system then issues an instruction to transition into kernel space and passes the system call number as well as the arguments to the kernel. The System Service Dispatcher uses this system call number to locate the address of the system call function in the System Service Dispatch Table (SSDT) and then executes this routine [58].

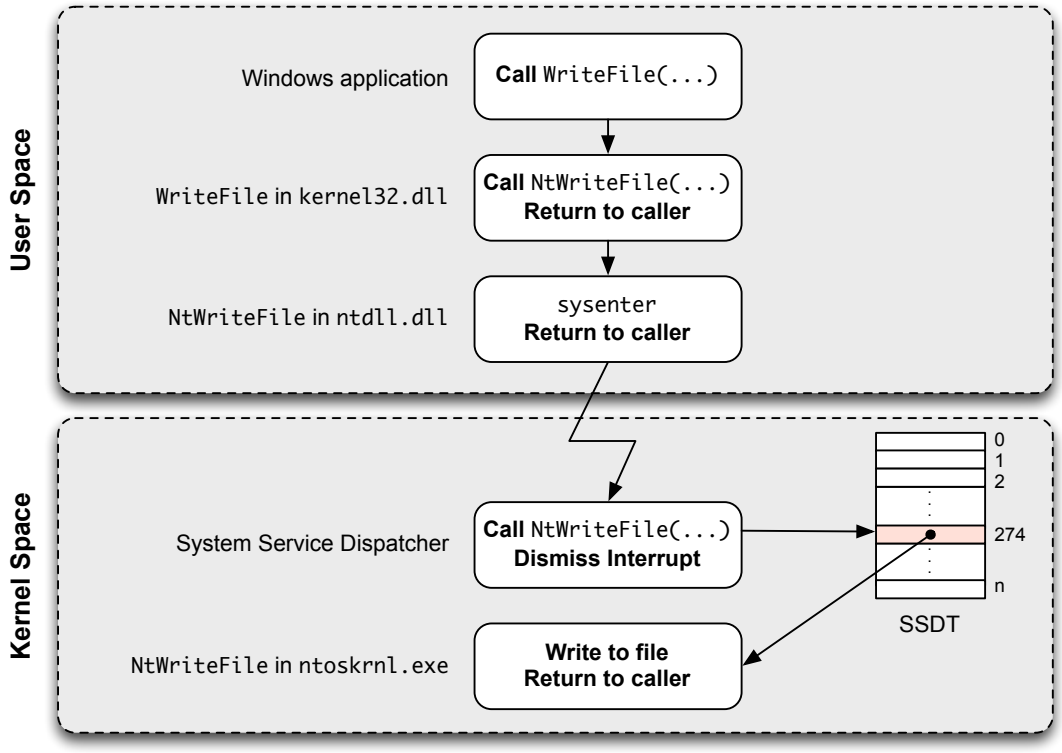

Figure 3.1: Execution flow of `WriteFile` in Windows XP [58].

3.2 Out-of-the-Box Monitoring

For out-of-the-box monitoring we use Anubis, which has been extensively described in previous work [10, 13, 14]. Anubis executes malware in a modified version of the open-source full system emulator Qemu [5, 15] and monitors its behavior by logging invocations of Windows API and Windows Native API functions. Anubis intercepts these function calls by comparing the instruction pointer of the emulated CPU with the entry points of monitored functions and passes the arguments to callback routines performing logging and further analysis [20]. Furthermore, Anubis uses dynamic taint tracking to analyze the data-flow between function calls and reveal data dependencies [10]. These techniques, however, result in heavy-weight monitoring and make Anubis susceptible to the detection of emulation-specific characteristics such as emulation bugs and timing.

Figure 3.2 illustrates Anubis residing outside the Windows analysis environment. The Windows environment is provided by a Windows XP installation represented as a Qemu snapshot. This snapshot saves the state of the running Windows environment before the analysis start. Therefore, Anubis can revert to a clean state after analysis by reloading this snapshot. The analysis produces logs in text as well as XML format, from which a behavioral profile is extracted after analysis. Furthermore, Anubis enriches the network behavior obtained from the monitored function calls with an additional network trace. We will discuss these execution artifacts in Section 4.1.

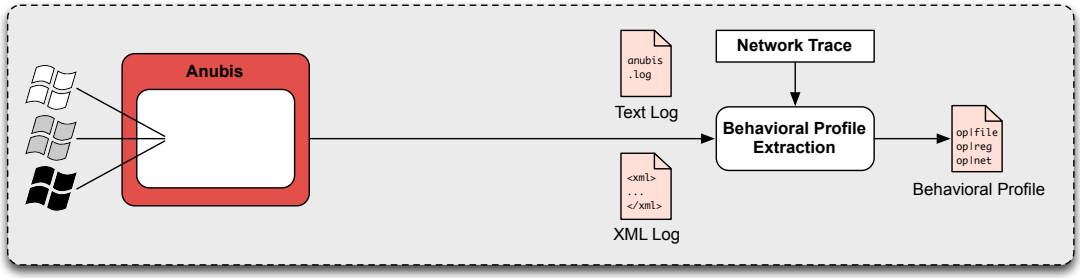

Figure 3.2: Out-of-the-box execution monitoring overview.

3.3 In-the-Box Monitoring

For in-the-box monitoring, on the other hand, we use a custom-built system that provides light-weight monitoring of a malware's behavior at the system call level. To this end, we extended an existing Windows kernel driver [9] that intercepts and logs system calls. Figure 3.3 illustrates the driver residing inside the Windows analysis environment. This design allows us to use any Windows XP installation and does not require a particular analysis environment. Thus, deploying DISARM on any host, both virtual and physical, and expanding it with new Windows installations is trivial. However, resetting the Windows installation after analysis has to be implemented separately by hardware or software solutions, as we will discuss in Section 6.

During analysis the driver produces a binary log and an additional synchronization log. We then convert these log files after analysis to resemble the analysis artifacts produced by Anubis to facilitate the comparison of artifacts produced by both systems. We will discuss the contents of the binary and synchronization log as well as the log conversion later in this section.

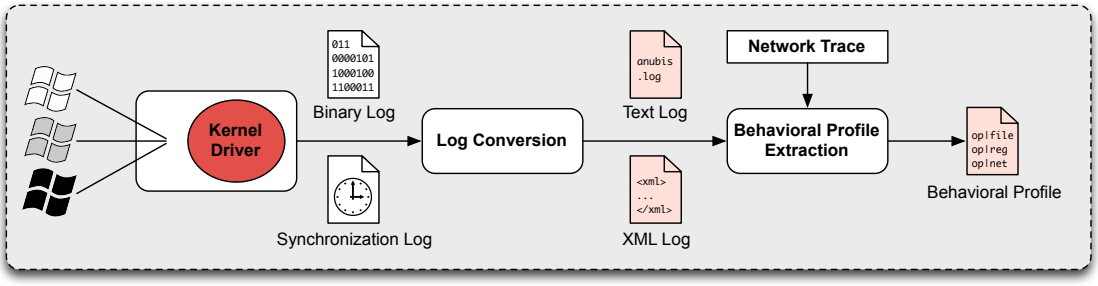

Figure 3.3: In-the-box execution monitoring overview.

Kernel Driver

In our driver we implemented a technique commonly used by rootkits: We intercept calls to the Windows Native API by hooking the entries of the SSDT [24]. As already mentioned, the SSDT contains the addresses of all native system calls. Every time a system call is invoked, the System Service Dispatcher queries the SSDT for the address of the function that implements the system call. On startup our driver overwrites the entries in the SSDT with addresses of replacement functions, which perform the execution monitoring and forward calls to the original system call functions. For this purpose and to restore the original SSDT entries when our driver is unloaded, we maintain a backup table of the original SSDT entries.

We use this approach to monitor 283 system calls (for a full list see Appendix A.1). Because of this high number of system calls we automatically generate the source code for the replacement function. For this we use the *generator* tool, which was originally developed for Anubis [13] and also used for a similar driver by Balzarotti et al. [9].

To log only relevant data, the driver maintains a list of threads related to the processes of the analyzed malware and only logs calls originating from these threads. At analysis start we just monitor threads of the original malware process. The following events trigger the inclusion of further processes and threads into the analysis:

- Process creation
- Service creation
- Injection of a remote thread into another process
- Writing to the virtual memory of another process
- Mapping code/data into another process

For threads not included in the analysis, the replacement functions simply forward the invocation to the original system call. For monitored threads each replacement function records the system call number and input parameters, before forwarding the invocation to the original system call. After execution, the replacement function further records the return value and the output parameters. The driver logs this data in separate buffers for each thread. Therefore, we also have to record timing information in a separate synchronization log file to serialize the logged system calls after analysis. In order to save space, we do not record a timestamp for each system call. Instead, we log timestamps in fixed intervals and on thread scheduling.

Listing 3.1 summarizes our approach for the replacement function of `NtWriteFile` as an example. First, the driver checks if the current thread ID is in the list of logged threads. In this case the driver logs the input parameters and a timestamp, if necessary. Then the driver forwards the call to the original function. After the execution the driver checks again if another thread was scheduled in the meantime and if logging a timestamp is necessary. It then logs the return value and the output parameters and returns the return value to the caller.

Listing 3.1: Pseudocode of the `NtWriteFile` replacement function.

```
NTSTATUS NewNtWriteFile(IN HANDLE FileHandle, IN HANDLE Event,
                        IN IO_APC_ROUTINE *ApcRoutine, IN VOID *ApcContext,
                        OUT IO_STATUS_BLOCK *IoStatusBlock, IN VOID *Buffer,
                        IN ULONG Length, IN LARGE_INTEGER *Offset,
                        IN ULONG *Key) {

    ...
    syscall_nr = 274;
    tid = PsGetCurrentThreadId();
    hook = IsHookedThread(tid);

    if (hook) {
        //log input parameters
        WriteLogData(syscall_nr, tid, FileHandle, sizeof(FileHandle));
        ...
        //check if we need synchronization data
        CheckSyncData(tid);
    }

    ntStatus = ((OldNtWriteFile)(FileHandle, Event, ApcRoutine, ApcContext,
                                 IoStatusBlock, Buffer, Length, Offset, Key));

    if (hook) {
        //check if we need synchronization data
        CheckSyncData(tid);
        //log ntStatus and output parameters
        WriteLogData(syscall_nr, tid, ntStatus, sizeof(ntStatus));
        WriteLogData(syscall_nr, tid, IoStatusBlock, sizeof(IoStatusBlock));
    }

    return ntStatus;
}
```

Monitoring the creation of services requires special instrumentation. Applications start and stop Windows services by calling functions from the Services API. This API is part of the Windows API, but, unlike most other Windows API functions, does not call the Windows Native API. Instead, each function of the Services API executes a remote procedure call (RPC) to the Service Control Manager (SCM). The `services.exe` executable implements the SCM and provides service-related functionalities, such as starting a service as a process on `CreateService` API calls [58]. Therefore, we include the `services.exe` process in our analysis when we observe RPC connection requests to the SCM endpoint `\RPC Control\ntsvcs`. Subsequently, we monitor service creations as process creations by the SCM.

Transparent malware analysis requires an analyzer to operate at a higher privilege level than the maximum privilege level a malware sample can gain [17]. We therefore maintain the integrity of our system by prohibiting the loading of any other kernel drivers after analysis start. We achieve this by intercepting and not forwarding Windows Native API calls to `NtLoadDriver` as well as `NtSetSystemInformation`, which can be used to load drivers with the `SystemLoadAndCallImage` system information class. Another undocumented way for loading drivers is calling the internal kernel function `MmLoadSystemImage` directly [47]. This function is not exported by the kernel and its address changes in different Windows and service pack versions. We determine this address manually once for each Windows image used in our analysis with the Windows debugger WinDbg[1] and also intercept calls to this function.

User Space Control

We control the analysis from user space with a component responsible for loading and unloading the kernel driver. Furthermore, the kernel driver does not write any logs directly to files for performance reasons. Therefore, the user space component contacts the kernel driver in fixed intervals to transfer the logged data from kernel space to user space and then writes the system call logs to binary files. Binary data takes less space than human-readable data and further enhances performance during analysis. This approach was already used by Balzarotti et al. [9]. We extended their mechanism with automatic resizing of buffers to ensure the completeness of the binary log as well as with an additional synchronization log to serialize log entries from different thread logs.

Each binary log entry describes one system call parameter or the return value and contains the fields listed in Fig. 3.4. The *Call ID* is an incremental index for each system call, which is unique for each thread and allows us to assign a parameter to a specific call. The *System Call Number* represents the number of the system call as assigned in the SSDT, e.g. 274 for `NtWriteFile`. The fields *Param Name Length* and *Param Name* represent the length of the parameter name and the parameter name respectively. *Data Type* specifies the type of the parameter, e.g. if it is an integer or a string. Finally, the fields *Data Length* and *Data* store the length of the parameter value and the value itself.

As already mentioned, we only log synchronization data in fixed intervals and on thread scheduling. Figure 3.5 lists the fields of such a synchronization entry. The first entry is the

[1] `http://www.microsoft.com/whdc/devtools/debugging/default.mspx`

current *Timestamp*. The next entry specifies the *Process ID* of the currently executing process. The field *Call | Return* specifies whether the synchronization entry is recorded before or after the execution of a system call. The last four fields let us determine if the execution of the system call was interrupted by another thread. They specify the *Thread ID* and *Call ID* before and at the time of recording the synchronization entry.

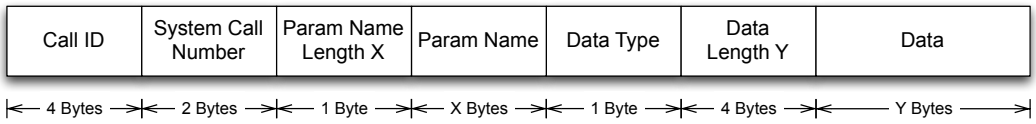

Figure 3.4: Log format of system call parameters.

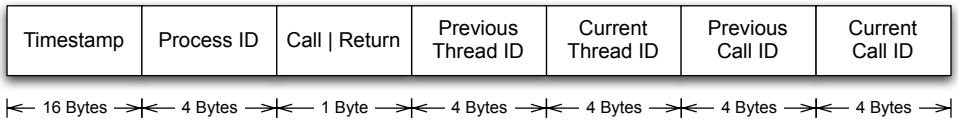

Figure 3.5: Log format of synchronization data.

Log Conversion

After the analysis we perform a log conversion to obtain a common basis for comparison of behavior from both our monitoring technologies (see Section 4.1 for a discussion of those behavior representations). We reassemble the binary logs and synchronization logs and enhance them with pseudo information in order to resemble the text and XML logs produced by Anubis. Based on these logs we are then able to produce further execution artifacts that are similar to those from Anubis.

During the log conversion we reconstruct the sequence of system calls from the synchronization logs. As each synchronization entry contains thread and call IDs, we can use these IDs to locate the corresponding system call number and parameters in the system call logs. We iterate over all calls in the binary log of a specific thread until a synchronization entry indicates that another thread was scheduled. In that case, we continue with the binary log of the new thread. For each call, we assemble all parameters into two lines in the text log: The first line represents the input parameters before the execution of the system call. The second line represents the output parameters after the return of the system call. Representing a call with two lines in the log file is necessary because parameters can be input and output parameters at the same time.

Listing 3.2 demonstrates the format of the converted log entries. Each line starts with a pseudo-timestamp obtained from the synchronization log, which we incremented with each call ID to make up for the fact that we do not have actual timestamps for each call. Each entry then contains “C” or “R” indicating the call or return, the name of the calling process and the index of the process and the thread. As the text log should be readable for a human analyst, each entry maps the system call number from the binary log to the name of the system call. It then

Listing 3.2: Example of system calls in the text log.

```
0:00:01.109031 C "sample.exe" 1 1 NtCreateFile(FileHandle: NULL, ...,
                                  ObjectAttributes: {"C:\foo.exe"}, ...)

0:00:01.109039 R "sample.exe" 1 1 NtCreateFile(FileHandle: **{42}** &23, ...,
                                  ObjectAttributes: {"C:\foo.exe"}, ...,
                                  mode: create): 0

0:00:01.218004 C "sample.exe" 1 1 NtWriteFile(FileHandle:
                                  **<42;NtCreateFile;FileHandle>** 23,
                                  ...)

0:00:01.218004 R "sample.exe" 1 1 NtWriteFile(FileHandle: 23, ...): 0

0:00:01.218008 C "sample.exe" 1 1 NtClose(Handle:
                                  **<42;NtCreateFile;FileHandle>** 23)

0:00:01.218008 R "sample.exe" 1 1 NtClose(Handle: 23): 0
```

lists all parameters including parameter name and value as well as the return value. We further enhance system calls with additional pseudo information derived from the actual parameters. For example, `NtCreateFile` either creates a new resource or opens an existing resource depending on the specified create options. We therefore add the additional pseudo argument *mode* specifying whether a resource was opened or created.

Inside the Windows kernel, data that needs to be shared, protected, named or made visible to user space programs via system calls is represented as objects [58]. When a process requires access to an object it has to possess a reference, called a *handle*, to this object. A process receives a handle, implemented as an integer value, when it creates or opens an object by its name or inherits it from its parent process. Listing 3.2 illustrates the access to a file object. A process retrieves a handle to the file `C:\foo.exe` by creating it and then writes to the file referring to it only by its handle. In order to assign system calls to operating system objects, we need to track handles and the corresponding object names. When a process closes an object, the system discards the reference and reassigns the handle value to other objects. Therefore, we cannot directly map handle values to object names. Instead, we assign synthetic labels to handles and track the use of these labels. We also implemented these labels as integers, but in contrast to handles they are unique during the whole analysis process. In Listing 3.2 we assign the label "`**{42}**`" to a handle on file creation. Whenever this handle is used as an input parameter, we refer to it with the label "`**<42;NtCreateFile;FileHandle>** 23`" including the name of the system call creating this handle and the type of the handle.

3.4 Monitoring Workflow

We automate the dynamic analysis with both monitoring technologies by integrating our in-the-box monitoring technology with the existing infrastructure of the out-of-the-box monitoring system Anubis. We start the analysis by submitting malware samples to the web interface of

the analysis server. The server creates an analysis task in its database, performs pre-processing activities and starts the analysis with either monitoring technology. After the analysis end, either by termination of the sample or by a timeout, the server performs the post-processing and stores the analysis results in the database. In detail the analysis server performs the following steps during the analysis of a malware sample:

1. The analysis server creates a new analysis task and starts an analysis script to process it.
2. The analysis script starts a sandbox using the in-the-box or out-of-the-box monitoring technology.
3. The analysis script pings the sandbox to determine when the Windows inside the sandbox has booted.
4. The analysis script copies the malware sample and the necessary analysis tools to the sandbox using the Common Internet File System (CIFS) protocol.[2]
5. The analysis script starts capturing the network traffic.
6. The sandbox starts the malware sample and signals the analysis start to the analysis script.
7. The analysis script sets a timer for a fail-safe timeout and starts a heartbeat to ensure the analysis is still ongoing and the sandbox has not crashed.
8. When the analysis is finished, the sandbox signals the analysis script the termination reason (either timeout, termination or error).
9. When the analysis did not end within the fail-safe timeout or the sandbox is unreachable, the analysis script terminates the sandbox.
10. The analysis script copies log files from the sandbox either using the CIFS protocol or from the mounted Windows image in case the network connection is lost.
11. The analysis script starts post-processing scripts that convert the log files from the in-the-box monitoring.
12. The analysis script enriches the log files with information from the network dump.
13. The analysis script terminates the sandbox and reverts it to a clean state.
14. The analysis script archives the log files and saves the results to the database.

[2]We do not include the analysis tools in the Windows installation of the sandboxes in order to facilitate fast deployment of new versions.

CHAPTER 4

Behavior Comparison

In this chapter we describe our behavior comparison approach as illustrated in Fig. 4.1. Behavior representations produced by our execution monitoring serve as the basis for comparison. In order to eliminate spurious differences we first perform a behavior normalization on these behavior representations. We then measure distances between behavior representations and calculate an evasion score. Based on this evasion score and a threshold, which we identify during our evaluation, we can classify a malware sample as showing the same or different behavior. We implemented all parts of this behavior comparison in Python[1], unless stated otherwise.

For each malware sample our approach requires behavior representations from different sandboxes as well as from multiple executions in the same sandbox. As we will discuss in Section 4.3, the behavior normalization determines random names based on names differing between the executions in the same sandbox. Furthermore, as we will discuss in Section 4.4, we consider the distance between multiple executions in the same sandbox as a baseline for the distance in behavior between different sandboxes.

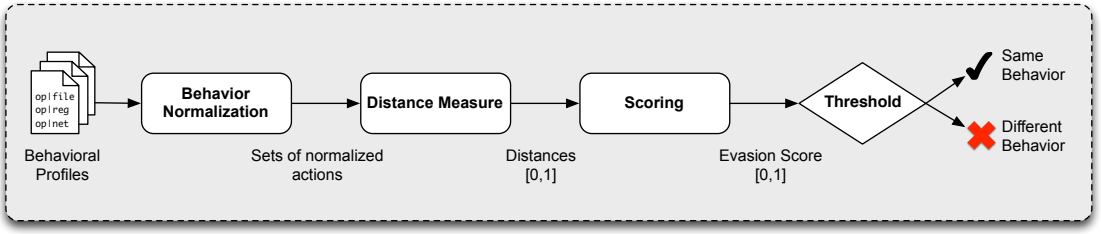

Figure 4.1: Behavior comparison overview.

[1] http://www.python.org/

4.1 Behavior Representation

The analysis of samples with either monitoring technology leads to the creation of a number of analysis artifacts such as a detailed text log of system calls, a human-readable XML report summarizing the observed behavior and a network traffic trace of all network communication performed by the malware that we capture with libpcap[2]. Furthermore, we extract a behavioral profile [10, 12] from the system call and network traces. This profile represents a malware's system and network-level behavior as a set of features. Each feature describes an action on an operating system (OS) resource, and is identified by the type and name of the resource, the type of action, optional arguments and a boolean flag representing the success or failure of the action:

```
<resource type>|<resource name>|<operation>[(<arguments)]:[<success>]
```

Examples for such an action are writing to a file, creating a process, a network connection attempt or setting a registry value:

```
file|C:\foo.exe|write:1
process|C:\Windows\foo.exe|create:0
network|tcp_conn_attempt_to_host|www.foobar.com
registry|HKLM\System\CurrentControlSet\Services|set_value('x'):1
```

All these analysis artifacts represent the malware's behavior at different levels of abstraction. Table 4.1 provides a comparison of the analysis artifacts and evaluates their suitability as the basis for behavior comparison. For the purpose of this work we chose to compare a malware's behavior based on behavioral profiles. The profiles provide a complete representation of all the malware's actions on OS resources and contain no execution-specific artifacts such as handles. As a profile is represented as a set, we can use standard set operations for comparison. Furthermore, each feature is tagged with a timestamp representing the offset into the analysis run when the feature was first observed [12]. As we will see in Section 4.2, this is essential to be able to compare behavior across monitoring technologies with vastly different performance overheads.

Nevertheless, we had to extend the current implementation of the behavioral profiles. Most network actions are extracted from the system call log that includes timestamps for each call. Some network actions, however, are added to the behavioral profiles from the libpcap network trace. In our current implementation no timestamps are available for this trace. In this case we have two options: We can assign the lowest possible timestamp and thereby add these actions at the beginning of a profile. This means that we always include them in our comparison. If we assign the highest possible timestamp and thereby add the actions at the end of a profile, it is likely that we will ignore them in most comparisons (for more details see Section 4.2). As network actions are an important indicator of the malware's behavior we chose the first approach.

We further had to modify the representation of network-related features because fast-flux service networks [64] or DNS-based load balancing may cause malware to contact different IP addresses in different executions. Behavioral profiles contain either a domain name or an IP address representing the network endpoint that the malware sample is communicating with. We extended the behavioral profiles by a mapping of $< IP\ address, domain\ name >$ obtained

[2] http://www.tcpdump.org/

	System call log	XML report	Behavioral profile
Contents	all system calls with input and output parameters and the return value	all successful file, registry, process, service and network operations	all operations on OS resources
Ordering	ordered by time	ordered by process, no timing information	ordered by OS resources, also includes timestamps
Processes	each call has information about the calling process and thread	operation are assigned to processes, no information about threads	no information about executing process or thread
Stability	unstable across executions (contains execution-specific artifacts such as handles)	relatively stable across executions	relatively stable across executions
Evaluation	• also includes irrelevant system calls • operations on handles, not resources	• failed operations are missing • no timestamps	• no comparison per process possible • represented as sets

Table 4.1: Comparison of analysis artifacts produced by the execution monitoring.

from the DNS queries in the libpcap log. We consider two network resources to be the same if *either one* of the IP address or the domain name used to resolve the IP address are the same.

Another challenge was posed by incomplete registry keys. When we include an already running process in the analysis, we sometimes miss the creation of resource handles. Furthermore, the OS provides handles to registry hive root keys to a process at its creation. Consequently, our system is sometimes unable to resolve the full name of a registry resource. We therefore automatically complete registry keys, that do not start with one of the registry hive root keys, either from other profiles that contain the full key, or from the information gathered from the Windows image used for analysis itself (see Section 4.2). Lanzi et al. [36] encountered the same problem and proposed the following approach: At the inclusion of a new process they retrieve already open resources by querying the open handle table for pre-existing handles. Alternatively, we could implement this feature in our driver in the future.

The behavioral profiles used in [10] also include dependencies that represent data-flow between OS resources. These dependencies can be used to determine execution-specific artifacts such as random names, that are derived from the random number generator or time sources. However, to maintain compatibility with light-weight monitoring technologies that cannot track the data-flow within the monitored programs, we do not consider dependencies in this work. Instead, we implemented the generalization of random names in our behavior normalization, which we describe in Section 4.3. This generalization can be applied to traces from any monitoring technology and does not require data-flow information.

4.2 Comparison Prerequisites

Termination Recognition

When comparing behavioral profiles produced by different monitoring technologies, it is highly unlikely that they will contain the same amount of features. The reason is that each monitoring technology is likely to have significantly different runtime overheads, so a sample will not be able to execute the same number of features on each system within a given amount of time. Nor can we simply increase the timeout on the slower system to compensate for this, since monitoring overheads may vary depending on the type of load. Thus, given two sandboxes α and β and the behavioral profiles consisting of n_α and n_β features respectively, DISARM only takes into account the first $min(n_\alpha, n_\beta)$ features from each profile, ordered by timestamp. In a few cases, however, this approach is not suitable. If the sample terminated on both sandboxes, or it terminated in sandbox α and $n_\alpha < n_\beta$, we have to compare all features. This is necessary to identify samples that detect the analysis sandbox and immediately exit. Samples that detect a sandbox may instead choose to wait for the analysis timeout without performing any actions. We therefore also compare all features in cases where the sample exhibited "not much activity" in one of the sandboxes. For this, we use a threshold of 150 features, which covers the typical amount of activity performed during program startup. This is the threshold observed by Bayer et al. [11], who in contrast observed 1,465 features in the average behavioral profile.

Feature Selection

As already mentioned, actions in the behavioral profiles are arranged in feature groups according to the OS resource they are performed on. Currently behavioral profiles describe actions on 15 different types of OS resources plus an additional *crash* resource that we added for our purposes:

- *file*: actions on filesystem objects
- *registry*: actions on registry objects
- *section*: actions on named as well as unnamed section objects
- *process*: actions on processes
- *job*: actions on job objects
- *thread*: actions on threads
- *random*: requests to the random number generator
- *time*: get and set system time
- *network*: network actions from the Anubis log as well as from libpcap
- *sync*: actions on named as well as unnamed semaphores and mutexes
- *driver*: loading and unloading of drivers
- *service*: actions on Windows services
- *popups*: popup titles and description from the `popupKiller.exe` application
- *info*: querying of system information and write access to the console
- *pseudo*: additional pseudo information
- *crash*: represents user-mode faults that cause Dr. Watson [40] to start

For each resource our profiles describe several different actions, e.g. open, create, write, delete or query. All resource types and actions are present in profiles from both monitoring technologies, except for actions on *service* resources. As discussed in Section 3.3, our driver currently does not monitor the Services API. However, service actions are represented as process creations, which are extracted from monitoring `services.exe`.

Not all features are of equal value for characterizing a malware's behavior, e.g. we consider file writes a more meaningful indicator for malicious behavior than file reads. In order to lastingly infect a system or leak private information of the user, malware has to write to the filesystem or the network. As a consequence, DISARM only takes into account features that correspond to persistent changes to the system state as well as network activity. This includes writing to the filesystem, registry or network as well as starting and stopping processes and loading drivers. This is similar to the approach used in previous work [8, 16] and, as we will show in Section 5.3, leads to a more accurate detection of semantically different behavior.

Furthermore, we do not consider the success of an action a difference in behavior. The action itself describes the intended behavior of the malware, while the success or failure of the action can result from environmental differences of the sandboxes.

We also considered including all actions in the comparison and assigning lower and higher weights to less and more interesting actions respectively. We experimented with assigning the inverse document frequency (IDF) [57] to each pair of $< resource\ type, action\ name >$. The IDF measures the importance of a term (in our case the pair of resource type and action) as a ratio of total documents in a dataset to documents that contain that term. However, as our evaluation in Section 5.3 shows, this system failed in representing the importance of individual types of actions accurately.

Image Information

As a prerequisite for the behavior normalization, we collect information about a Windows installation's filesystem and registry. For this purpose, we implemented a Windows batch script. For most parts of this script, we used built-in Windows command line tools such as `systeminfo`, `tasklist` and `regedit`. For other parts we used ports of GNU/Linux tools provided by GnuWin [2]. For example, the `file` command is a convenient way to determine file types, while, in contrast to `dir`, the `ls` command also resolves file shortcuts. We packaged the batch script in a PE assembly containing all additional tools. Therefore, we can submit this executable to analysis just like a normal sample and gather the following information about any Windows installation utilized in an analysis sandbox:

- complete file listing containing the full path, short DOS-path, file size, bzip2 compressed size, MD5 of the file's content, file type, ownership flags and destination of file shortcuts
- complete list of registry keys and their values (the registry dump)
- Windows environment variables such as the current user and the path to the program files
- output of the `systeminfo` command
- list of running processes

Furthermore, we implemented a parser for the output of the file listing and the registry dump. The parser presents the filesystem and the registry as a tree, with each node containing the various attributes collected by our batch script. This tree can be queried for the existence of elements as well as their attributes. It is thereby possible to map elements from different environments to each other that vary in path names, but represent the same resources. The obtained information also contains various installation-specific identifiers that we can query and generalize during behavior normalization.

Ultimately, we do not use all parts of the collected image information for behavior normalization. Nevertheless, the image information also proved to be useful for interpreting different behavior during our evaluation, discussed in Chapter 5, that was caused by environmental differences.

4.3 Behavior Normalization

In order to meaningfully compare behavioral profiles from different executions of a malware sample, we need to perform a number of normalization steps, mainly for the following two reasons: The first reason is that significant differences in behavior occur even when running an executable multiple times within the same sandbox. Many analysis runs exhibit non-determinism not only in malware behavior but also in behavior occurring inside Windows API functions, executables or services. The second reason is that we compare behavioral profiles obtained from different Windows installations. This is necessary to identify samples that evade analysis by detecting a specific installation. Differences in the filesystem and registry, however, can result in numerous differences in the profiles. These spurious differences make it harder to detect semantically different behavior. Therefore, we normalize each profile before comparing it to other profiles.

Figure 4.2 illustrates the order in which we perform the various normalization steps. We perform most steps individually on each profile, except for the randomization detection and the repetition detection, which we perform on profiles from multiple executions. Firstly, this is necessary to detect instances of random names that vary from execution to execution. Secondly, we propagate resource generalizations to other profiles in which our generalization missed instances of random behavior or repetitions.

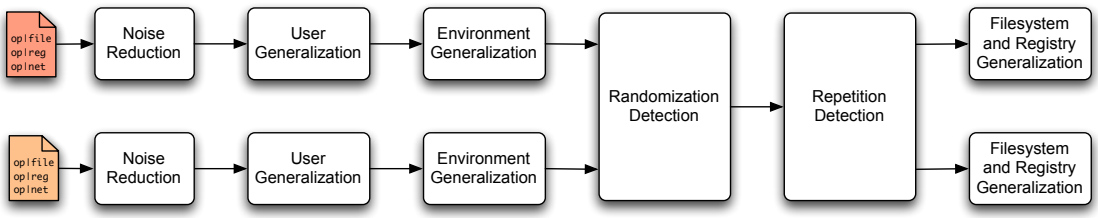

Figure 4.2: Sequence of behavior normalization steps.

Noise Reduction

In our experience even benign programs cause considerable differences when comparing profiles from different sandboxes. As a consequence, we captured the features generated by starting four benign Windows programs (`notepad.exe`, `calc.exe`, `winmine.exe`, `mspaint.exe`) on each sandbox, and consider them as "noise". We filter these features out of all behavioral profiles. Similarly, we filter out the startup behavior of `explorer.exe`, `iexplore.exe` and `cmd.exe` when we detect the creation of any of these processes in a profile. Furthermore, we filter actions by Dr. Watson and replace them with a single action representing the *crash*.

This "noise" behavior contains a lot of similar actions across all sandboxes and removing these actions might decrease overall similarity. This is acceptable as we are not interested in the highest possible similarity but in our ability to distinguish semantically different behavior. We achieve this by eliminating a number of differences in these actions that are not caused by malware behavior.

User Generalization

Programs can write to the home directory `C:\Documents and Settings\<username>` of the user logged in during analysis without needing special privileges. Malware samples therefore often write files to this directory. In the registry user-specific data is stored in the key `HKEY_CURRENT_USERS`, which actually points to `HKEY_USERS\<SID>`. The SID is a secure identifier created by the Windows setup program. It is unique for every user and system. Profiles from different systems certainly differ in the users SID and may also contain different usernames. We therefore generalize these values.

Environment Generalization

Other system-specific values include hardware identifiers and cache paths in the filesystem and the registry. Furthermore, names of folders commonly accessed by malware include the user home directory `C:\Documents and Settings` and the program directory `C:\Program Files` as well as their respective subfolders. The names of these folders depend on the language of the Windows installation. We generalize these identifiers and paths to eliminate differences caused by different Windows installations and not by the malware's behavior itself.

Randomization Detection

Malware samples often use random names when creating new files or registry keys. The use of random names can drastically decrease the similarity between different runs in the same sandbox and therefore also decreases the system's sensibility to detect differences between executions in different sandboxes. Random names can either be generated by using the Windows random number generator or time sources. Furthermore, a malware sample might randomly select a name from a internal list of possible combinations.

Since DISARM executes each sample multiple times in each sandbox, we can detect random behavior by comparing profiles obtained in the same sandbox. Like the authors of MIST [65], we assume that the path and extension of a file are more stable than the filename. As a consequence,

we detect all created resources (in the filesystem or registry) that are equal in path and extension but differ in name. If the same set of actions is performed on these resources in all executions, we assume that the resource names are random. We can thus generalize the profiles by replacing the random names with a special token. In some cases, names are only random in one sandbox but equal on others. Therefore, we compare created resources that our random generalization missed in one sandbox to already generalized resources in another sandbox. If we encounter created resources that match generalized names in the other sandbox in path and extension as well as the performed actions, then we also generalize them.

Repetition Detection

Some types of malware perform the same actions on different resources over and over again. For instance, file infectors perform a scan of the filesystem to find executables to infect. This behavior leads to a high number of actions, but in reality only represents one malicious behavior. Furthermore, these actions are highly dependent on a sandbox's filesystem and registry structure. To generalize this behavior, we look for actions that request directory listings or enumerate registry keys. We also consider the arguments that are passed to the enumeration action, for example queries for files with the extension ".exe". For each such query, we examine all actions on resources that were performed after the query and that match the query's arguments. If we find any actions (such as file writes) that are performed on three or more such resources, we create a generalized resource in the queried path instead of the individual resources and assign these actions to it.

Furthermore, we also have to consider repetitions when comparing profiles from different executions and different sandboxes. We consider actions that were detected as repetitions in only one sandbox, but match the enumeration query in the other sandbox without reaching our threshold of three resources, as equal. For example, if there are actions on three or more resources detected as repetitions in the first sandbox, but there are only actions on one or two resources that match the same query detected in the second sandbox, then we generalize the actions in the second sandbox as the same repetition.

Some actions are not directly recognizable as repetitions and require a different approach to generalization: Malware can query the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\CURRENTVERSION\APP PATHS` for paths to common applications. Therefore, whenever we encounter an enumeration of that registry key, we also generalize actions on the queried paths as repetitions.

Filesystem and Registry Generalization

We use the information we gather about a Windows image's filesystem and registry at analysis start (see Section 4.2) to view a profile obtained from one image in the context of another image. This allows us to remove actions that would be impossible or unnecessary in the other image. That is, we ignore the creation of a resource that already exists in the other image and, conversely, the modification or deletion of a resource that does not exist in the other image. We thereby further remove behavioral differences caused by differences in the environment.

4.4 Distance Measure and Scoring

As already mentioned, a behavioral profile represents actions as a set of string features. We thus compare two behavioral profiles by calculating the ratio of the number of actions in the intersection of sets a and b to the number of actions in the union of sets a and b. This ratio is defined as the Jaccard distance [28]:

$$J(a,b) = 1 - \frac{|a \cap b|}{|a \cup b|}. \tag{4.1}$$

This distance score provides results in the range [0,1], with zero indicating that two profiles are equal and one indicating that two profiles contain no common actions.

Balzarotti et al. [9] observed that two executions of the same malware program can lead to different execution runs. Our own experiments reveal that about 25 % of samples perform at least one different action between multiple executions in the same sandbox. Because of this, we cannot simply consider a high distance score as an indication of evasion. Instead, we consider the deviations in behavior observed within a sandbox as a baseline for variations observed when comparing behavior across different sandboxes. We can thereby mitigate the effect of varying behavior between executions in the same sandbox, e.g. caused by random behavior. For our evaluation we decided to execute each sample three times in each sandbox to reliably capture deviations in behavior. Thus, DISARM requires a total of six analysis runs to calculate an evasion score for a malware sample in two different sandboxes. We think this overhead is feasible when using DISARM as a verification tool for Anubis. In this case samples showing little or no activity in Anubis, as well as samples selected at random, can be subjected to further analysis using DISARM to verify the results or detect analysis evasions.

Thus, in order to decide whether a sample exhibits different behavior in any of the sandboxes, we calculate an evasion score in two steps: First, we compare all profiles from the same sandbox to compute the variation between executions. Then we compare all profiles from different sandboxes to get the maximum variation between analysis sandboxes. Based on these two distances we obtain an evasion score for each sample:

$$E = \max_{1<i<n} \left\{ \max_{1<j<n, i \neq j} \Big\{ \mathrm{distance}(i,j) - \max\{\mathrm{diameter}(i), \mathrm{diameter}(j)\} \Big\} \right\}. \tag{4.2}$$

Here, the $\mathrm{diameter}(i)$ is the maximum distance between executions in the sandbox i, while $\mathrm{distance}(i,j)$ is the maximum distance between all executions in the sandboxes i and j. Thus, the evasion score is the difference between the maximum *inter-sandbox distance* and the maximum *intra-sandbox distance*. We calculate evasion scores in the interval [0,1], with zero representing the same behavior and one representing completely different behavior. If this score exceeds an evasion threshold, that we identified during evaluation, DISARM declares that the malware has performed semantically different behavior in one of the sandboxes.

We also considered calculating our evasion score based a modified version of the Dunn index [19], which is commonly used for behavior clustering (described in Section 2.4). Per definition, the Dunn index calculates the ratio between the minimum inter-cluster distance to the maximum intra-cluster distance. For our purposes, we replace cluster distances with sandbox distances and define the Dunn index as follows:

$$D = \max_{1<i<n} \left\{ \max_{1<j<n, i \neq j} \left\{ \frac{\text{distance}(i,j)}{\max\{\text{diameter}(i), \text{diameter}(j)\}} \right\} \right\}. \tag{4.3}$$

In our experiments, executions on the same image often yielded a diameter of zero, which leads to a division by zero in Eq. 4.3. As a consequence, and to limit the result range to [0,20], we defined a minimum diameter of 0.05. This approach, however, was too sensitive to high intra-image distances and was unable to detect some samples with different behavior. Consequently, we abandoned the ratio in favor of the overall distance score as described in Eq. 4.2.

CHAPTER 5

Evaluation

In our evaluation we demonstrate the effectiveness of our approach to detecting deviations in malware behavior. To this end, we collected and compared execution traces from different monitoring technologies as well as Windows environments. We first evaluated DISARM on a small training dataset to determine a threshold for our evasion scores. We then applied this threshold to the results of a large scale test dataset, which led to the detection of a large number of evasive samples. We manually analyzed a number of these samples and we will discuss our findings in this chapter. We illustrate the results of our experiments with figures produced by Matplotlib[1].

5.1 Setup

We use two different analysis systems: We run Anubis as out-of-the-box monitoring system and our kernel driver as in-the-box monitoring system. To simplify deployment and the integration into our existing analysis infrastructure, we run the driver-based sandboxes inside an unmodified Qemu emulator (version 0.11), rather than on a physical system. This limits our ability to detect evasion techniques targeted against Qemu CPU emulation bugs that may be present in both monitoring technologies. Nevertheless, we avoid the overhead of resetting the hard disk of a physical machine to a clean state after analysis and minimize environmental differences caused by hardware drivers etc. Our results showed that we effectively eliminate these differences during normalization. Therefore, we are planning to extend our driver-based monitoring system to analyze samples on a physical system in the future. We could use one of the hardware or software solutions discussed in [69] or the approach used by CWSandbox [27] to restore the state of the physical environment.

In this setup we can use the same Windows image in Anubis as well as with our driver by deploying the same Qemu snapshot file. Therefore, we can monitor evasion techniques targeting different monitoring technologies. Comparing executions from different Windows images allows us to detect differences caused by configuration and installation characteristics.

[1] `http://matplotlib.sourceforge.net/`

Sandboxes

We used four different sandboxes and analyzed each sample three times in each of the sandboxes, resulting in a total of 12 executions per sample. We then calculated the evasion score for each pair of sandboxes using Eq. 4.2 in Section 4.4. The overall evasion score of a sample is the maximum evasion score calculated for any pair of sandboxes.

Table 5.1 summarizes the most important characteristics of these sandboxes. In the following we will refer to each sandbox by the names shown in the first column. The first image, used in the *Anubis* and *Admin* sandboxes, was an image recently used in the Anubis system. We selected two additional images that differ in the language localization, the username under which the malware is running (all users possess administrative rights), as well as the available software. These differences allow us to evaluate the effectiveness of our normalization.

Table 5.1: Sandboxes used for evaluation.

Sandbox	**Monitoring Technology**	**Image Characteristics**		
		Software	**Username**	**Language**
Anubis	Anubis	Windows XP Service Pack 3, Internet Explorer 6	Administrator	English
Admin	Driver	same Windows image as Anubis		
User	Driver	Windows XP Service Pack 3, Internet Explorer 7, .NET framework, Java Runtime Environment, Microsoft Office	User	English
German	Driver	Windows XP Service Pack 2, Internet Explorer 6, Java Runtime Environment	Administrator	German

Performance Comparison

Timing is an important measure by which malware can differentiate an analysis sandbox from a physical host. Although we deploy our driver in Qemu, there is a measurable performance difference between Anubis and the driver. To quantify this difference, we tested our sandboxes against the following benchmarks:

1. *io*: IO bound test, achieved through checksumming the performance test binary
2. *io7z*: IO bound test, achieved through the 7-Zip[2]compression of the test binary
3. *cpu*: CPU bound test, achieved through a `sprintf` loop
4. *cpu7z*: CPU bound test, achieved through the 7-Zip benchmark
5. *syscall*: CPU bound system call test, achieved through iteration of the registry

[2] `http://www.7-zip.org/`

Figure 5.1 shows the average results of 30 test executions in each sandbox as well as in an unmodified version of Qemu without any analysis tool running. For each of the tests, our driver was about ten times faster than Anubis. We can attribute this difference to the heavy-weight instrumentation and data tainting capabilities of Anubis. This performance comparison demonstrates that our setup should be capable of detecting timing-sensitive malware. Furthermore, we can measure the performance overhead caused by our driver when comparing the performance of our sandboxes running the driver with the unmodified Qemu. On average, the execution of the benchmarks with the driver was about two times slower than in Qemu without the driver. The worst-case scenario is the *syscall* benchmark, which is achieved by invoking system calls and therefore causes our driver to log every one of these calls.

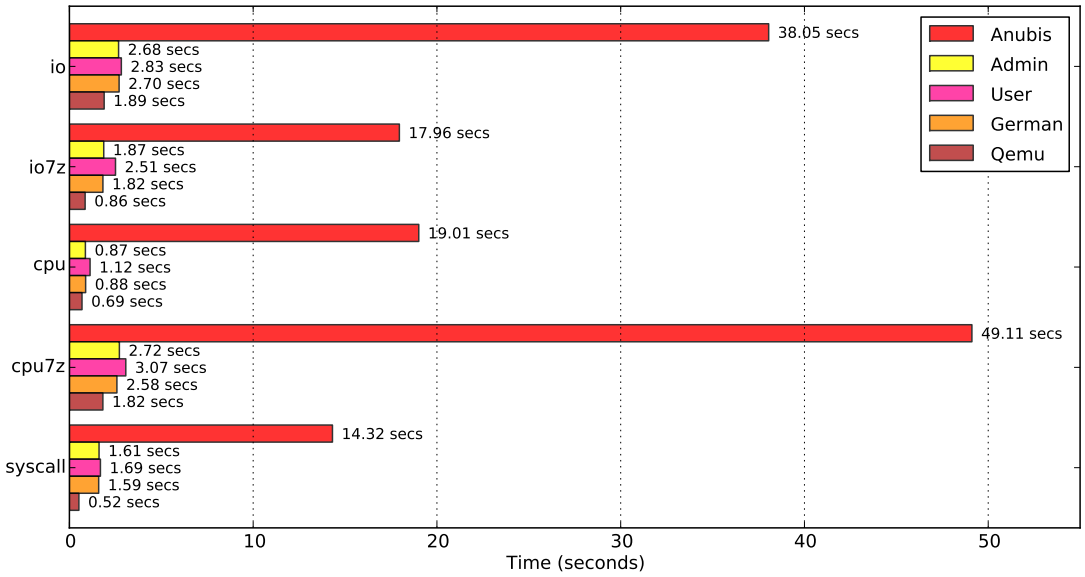

Figure 5.1: Performance comparison of our sandboxes.

5.2 Sample Collection

We randomly selected malware samples among those submitted to Anubis between August 2010 and March 2011. To ensure our datasets were as diverse as possible, we selected samples belonging to different malware families. For the training dataset only one sample per malware family was used. For the test dataset we allowed a maximum of five samples per family. Malware families were assigned based on virus labels from Kaspersky Anti-Virus, that we obtained from VirusTotal [7]. Kaspersky's naming rules [61] state that threats are named according to

```
[Prefix:]Behaviour.Platform.Name[.Variant]
```

where *name* defines the family of a malware sample. Samples were differentiated by their family name only without the specific variant suffix.

At the time of analysis, each malware sample was not older than four days to ensure the availability of network resources such as C&C servers. To make sure that at all times only the most recent malware samples were submitted for analysis in our sandboxes, a Python script monitored the queue of remaining analysis tasks and selected and submitted one malware sample at a time as soon as the queue was empty.

5.3 Training Dataset

To develop our techniques and select a threshold for evasion detection, we created a small, labeled training dataset. For this, we selected 175 malware samples among those submitted to Anubis in August 2010. Furthermore, we included ten additional samples based on comments from Anubis users, that suggested the samples were evading analysis. Thus, the training dataset consisted of a total of 185 samples.

Classification

To establish a "ground truth" for the training dataset we manually inspected the generated behavioral profiles and in a few cases even the binaries themselves. Manual classification of these samples showed that 43 (23.24 %) displayed different behavior in at least one of the sandboxes. We further split these samples into five groups: samples evading Anubis, samples evading our driver, samples requiring the .NET framework and therefore only working in the *User* sandbox (see Table 5.1), samples not working in the *German* sandbox due to hardcoded English paths and samples showing different behavior due to various other installation characteristics (e.g. the missing installation of Java in the Anubis image).

Table 5.2: Observed behavioral differences in the training dataset.

Percentage	#	Description of behavior
76.76 %	142	same behavior
9.19 %	17	evasion of Anubis
5.41 %	10	.NET environment required
3.24 %	6	evasion of our driver
3.24 %	6	different behavior due to other characteristics
2.16 %	4	not working in the German environment

Table 5.2 summarizes the amount of samples for each class and Figure 5.2 visualizes the size of the different classes. It has to be noted that samples requiring the .NET environment as well as samples showing different behavior due to installation characteristics also can be characterized as not working in Anubis. This amounts to a total of 14.60 % of samples, which cannot be analyzed in our current Anubis environment.

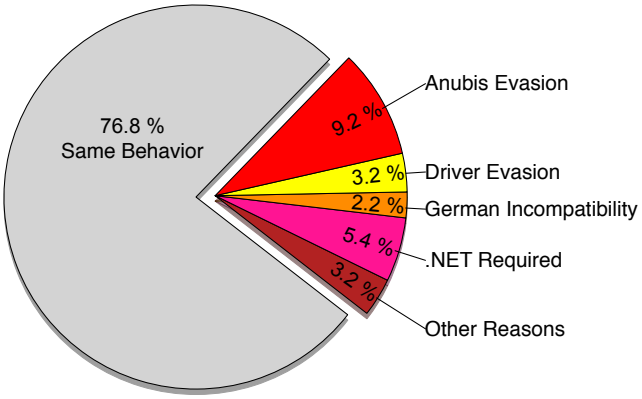

Figure 5.2: Visualization of classes of observed behavioral differences in the training dataset.

Quantitative Results

Our techniques succeeded in assigning low evasion scores to samples with the same behavior and high evasion scores to samples displaying different behavior. To visualize these results, we plotted the maximum diameter (intra-sandbox distance) against the maximum distance (inter-sandbox distance), as discussed in Section 4.4, in Fig. 5.3. The overall evasion score can be seen as the distance of each point from the diagonal. Points close to the diagonal represent samples with low evasion scores, while points farther away from the diagonal represent samples with high evasion scores. Points close to the y-axis are samples exhibiting little variation between analysis runs in the same sandbox. This is the case for the larger part of our training dataset, confirming the effectiveness of our normalization techniques. Only 8.11 % display a maximum intra-sandbox variation greater than 0.1 as a result of non-deterministic behavior such as crashes that occur only in some executions.

Figure 5.3 displays samples labeled as exhibiting different behavior as filled points, and those with the same behavior as empty points. The samples with different behavior all received evasion scores above 0.4. The samples with the same behavior all received evasion scores below 0.4, except for one sample. We could attribute this behavior to the lack of timestamps for network actions, that we add after analysis from an external network trace. In order to not always ignore these actions, we had to add them at the beginning of each profile (see Section 4.1). Nevertheless, in this case these actions account for a substantial difference in behavior, although they are not always relevant for the selected comparison timeframe.

The dashed lines, which are parallels to the diagonal, represent threshold candidates. For the training dataset a threshold of 0.4 results in detecting all samples with different behavior, while incorrectly classifying one sample with the same behavior. Eliminating this false positive would mean choosing a threshold greater than 0.53, which leads to six false negatives. Consequently, the threshold has to be chosen according to the detection strategy, either minimizing false positives or maximizing the detection rate for samples with different behavior. In this case the false positive was caused by a flaw in our behavioral representation, which can be fixed. Therefore, we propose using 0.4 as our threshold for the experiments on the test dataset.

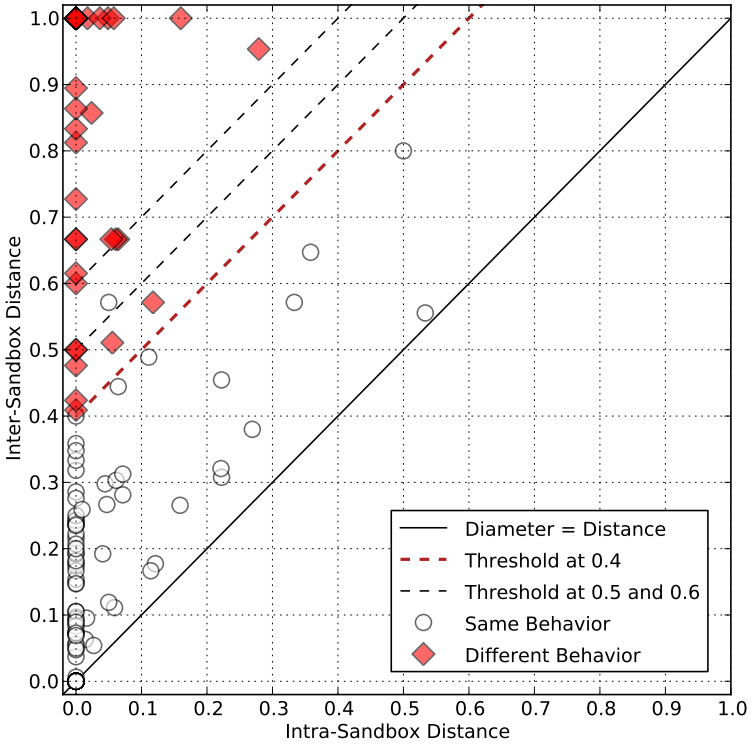

Figure 5.3: Maximum diameter (*intra-sandbox* distance) vs. maximum distance (*inter-sandbox* distance) with thresholds (0.4,0.5,0.6).

We visualize our results for each pair of sandboxes in Fig. 5.4. The figure includes all possible pairs and therefore the same results twice, e.g. the results for *Anubis vs. Admin* are the same results as for *Admin vs. Anubis*. Nevertheless, this depiction allows us to directly compare one sandbox to the other three sandboxes by listing all possible comparisons for one sandbox in a row. Note that the classification as evasion is displayed in all comparisons but is only applicable to specific pairs of sandboxes, e.g. a sample classified as *Anubis Evasion* can only be detected as an evasion in pairs of sandboxes including the *Anubis* sandbox. Figure 5.4 demonstrates that our techniques succeeded in this regard. Samples classified as *Anubis Evasion* or *Driver Evasion* received high evasion scores in at least one of the pairs including the *Anubis* sandbox or sandboxes running the driver respectively, while receiving evasion scores below the threshold in the other sandbox pairs. One sample classified as *Anubis Evasion* also received an evasion score above the threshold for comparisons between the *Admin* sandbox and the other two sandboxes running our driver. This indicates that the sample shows different behavior because of characteristics of the Windows environment in *Anubis* and not the monitoring technology. Samples classified as requiring .NET only received high evasion scores in pairs including the *User* sandbox, while showing the same behavior and therefore receiving low evasion scores in other comparisons. Samples classified as not working in the German image correctly received high evasion scores in pairs including the *German* sandbox.

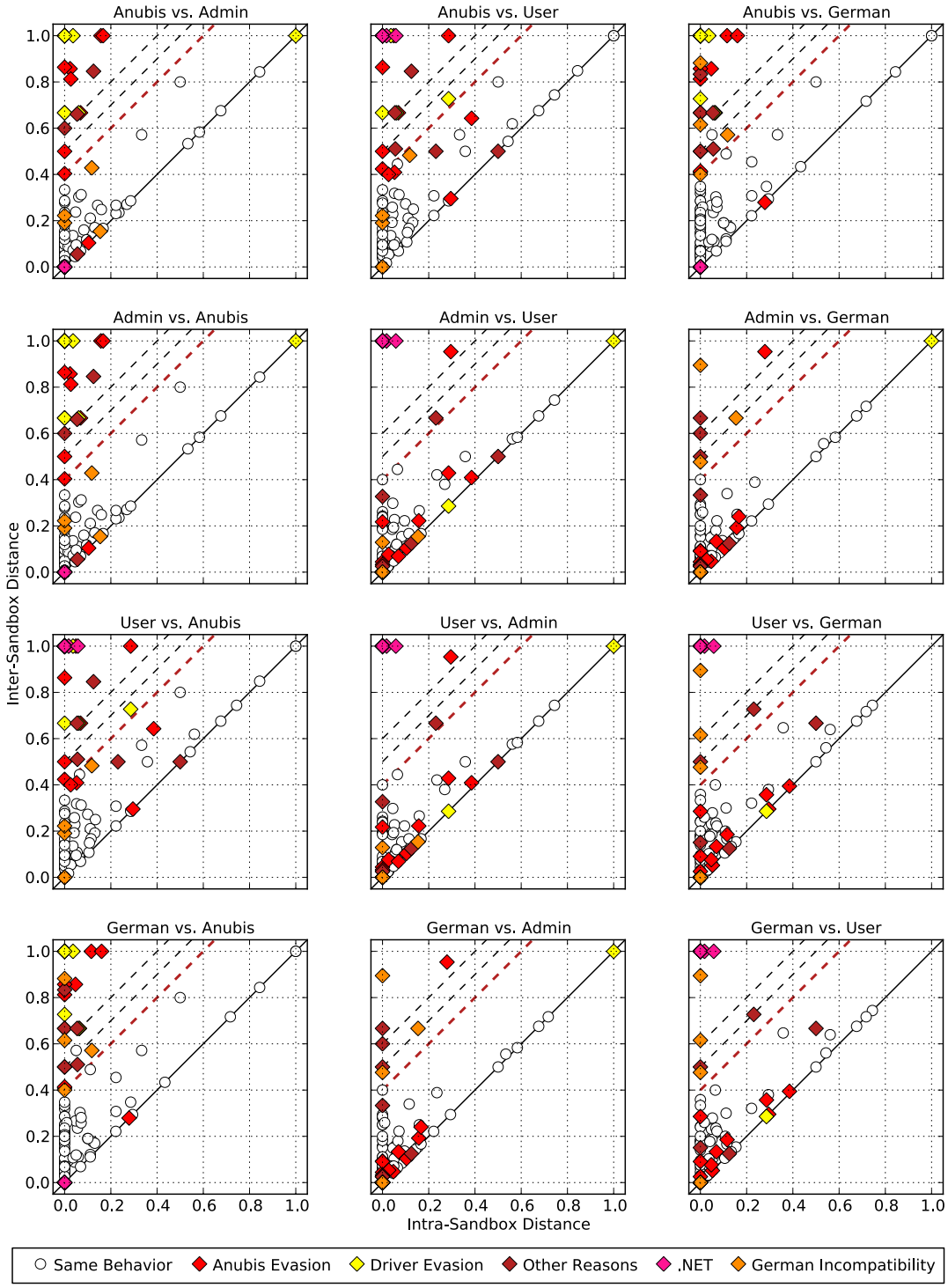

Figure 5.4: Maximum diameter (*intra-sandbox* distance) vs. maximum distance (*inter-sandbox* distance) with thresholds (0.4,0.5,0.6) for each pair of sandboxes.

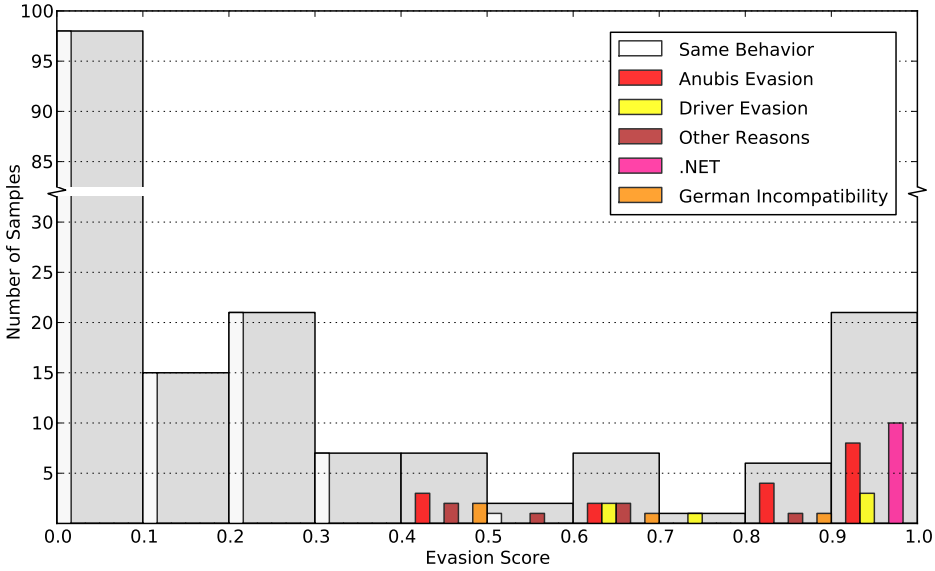

Figure 5.5: Histogram of evasion scores for the training dataset.

While Fig. 5.3 and 5.4 demonstrate the evasion score distribution and possible thresholds, the number of samples for each evasion score is not obvious from this visualization. Therefore, we show a histogram of the scores for the training dataset in Fig. 5.5. This figure illustrates the number of samples that received an evasion score of [0, 0.1],]0.1, 0.2], ... and]0.9, 1.0]. The y-axis is broken into two parts to accommodate the large number of samples that received a score between 0 and 0.1. Again, this figure shows that all samples with different behavior received scores above our determined threshold of 0.4, while, as already shown in Fig. 5.3, one sample with the same behavior received a score between 0.5 and 0.6. Apart from this one false positive, Fig. 5.5 proves that our results for samples with the same behavior and different behavior are well separated. Thus, our system succeeded in distinguishing between these two classes.

Result Accuracy

We further wanted to measure the effect of the various normalization steps on the results and verify our design decision to only compare persistent actions. For this purpose we calculate the proportion of correctly classified samples in the training dataset for each normalization step and types of actions at all possible thresholds. This metric is called accuracy and is defined as follows:

$$accuracy = \frac{|True\ Positives| + |True\ Negatives|}{|All\ Samples|} \cdot 100. \tag{5.1}$$

For our purposes, we use the following definitions for the variables in Eq. 5.1:

- True positives (TP) are samples exhibiting different behavior, that are correctly classified.
- True negatives (TN) are samples exhibiting the same behavior, that are correctly classified.

- False positives (FP) are samples exhibiting the same behavior, that are incorrectly classified as having different behavior.
- False negatives (FN) are samples exhibiting different behavior, that are incorrectly classified as having the same behavior.

We applied the normalization steps, as described in Section 4.3, in ascending order and calculated the accuracy for each step (see Fig. 5.6): no normalization (*default*), the removal of noise (*noise*), the generalization of user-specific artifacts (*user*), the generalization of environment-specific artifacts (*environment*), the detection and generalization of random names (*random*), the detection of repetitions (*repetitions*), and the generalization of missing filesystem and registry resources (*missing*).

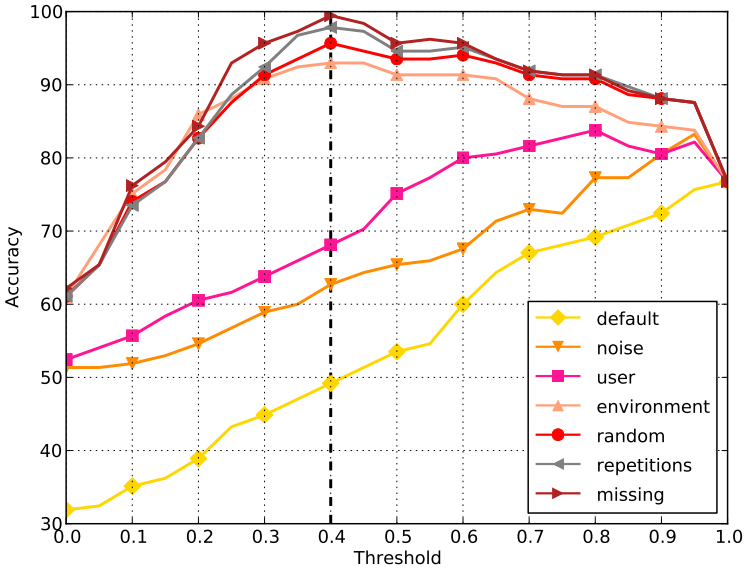

Figure 5.6: Overall accuracy for each normalization step at thresholds [0,1].

Overall, we achieved an accuracy of more than 95 % for thresholds between 0.3 and 0.6, with the highest accuracy of 99.5 % at the chosen threshold of 0.4. Every normalization step improved the accuracy by some degree, with the removal of noise and the generalization of the user and the environment yielding the largest improvements.

We further calculated the accuracy of our final results (with all normalization steps applied) when comparing only persistent actions, all actions in a behavioral profile or all actions in a profile with IDF weights assigned to each action. Figure 5.7 demonstrates that if we consider all actions instead of only persistent actions, we obtain a maximum accuracy of 89.2 % for thresholds between 0.6 and 0.7. Also the use of IDF weights only marginally improves the results, which shows that these weights do not accurately represent the importance of individual types of actions.

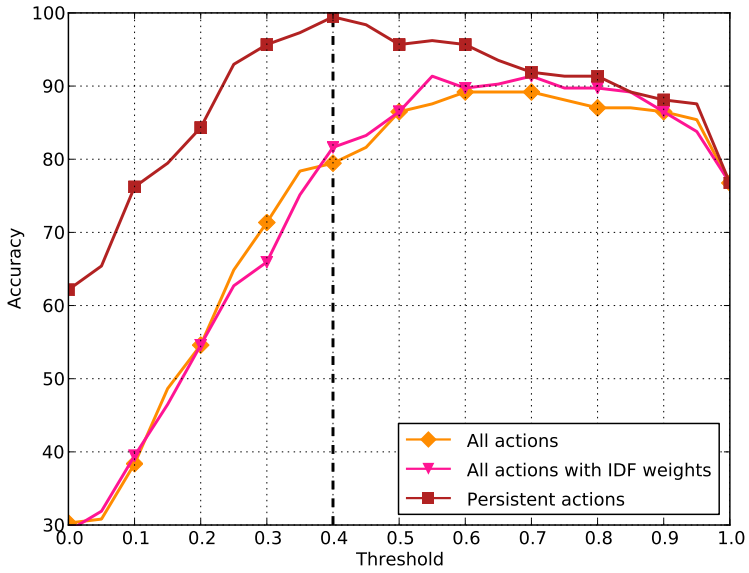

Figure 5.7: Overall accuracy for persistent/all/weighted actions at thresholds [0,1].

We summarize these results in Table 5.3 for the different comparison types. The comparison types include the comparison of persistent actions with all normalization steps applied in ascending order, as well as the comparison of all actions and the use of IDF weights with all normalization steps applied. For each comparison type we considered the results at the respective threshold t that yielded the highest result accuracy. For example, without any normalization the best accuracy was obtained at $t = 1.0$, which means that we do not detect any evasive samples. These results demonstrate that our system works as a whole and that all of the normalization techniques, as well as our decision to only compare persistent actions, play an important role in achieving the highest possible classification accuracy.

Comparison Type		t	**TP**	**TN**	**FP**	**FN**	**FP Rate**	**FN Rate**	**Accuracy**
Persistent	*Default*	1.00	0	142	0	43	0.00 %	100.00 %	76.76 %
	Noise	0.95	17	137	5	26	3.52 %	60.47 %	83.24 %
	User	0.80	23	132	10	20	7.04 %	46.51 %	83.78 %
	Environment	0.40	38	134	8	5	5.63 %	11.63 %	92.97 %
	Random	0.40	42	135	7	1	4.93 %	2.33 %	95.68 %
	Repetitions	0.40	43	138	4	0	2.82 %	0.00 %	97.84 %
	Missing	0.40	43	141	1	0	0.70 %	0.00 %	99.46 %
All		0.60	32	133	9	10	6.34 %	25.58 %	89.19 %
IDF Weights		0.55	38	131	11	5	7.75 %	11.63 %	91.35 %

Table 5.3: Result comparison at thresholds t in the training dataset.

5.4 Test Dataset

We tested DISARM on a larger test dataset of 1,686 samples collected between December 2010 and March 2011. We used the evasion threshold of 0.4 selected in our experiments on the training dataset. Note that, in contrast to the training dataset, we do not have a ground truth for this dataset and therefore are unable to provide a complete discussion of the quantitative results for each class of behavior as well as an accuracy score.

Quantitative Results

In this dataset, a total of 431 (25.56 %) samples resulted in an evasion score above the threshold. Figure 5.8 illustrates the distribution of evasion scores in total for all sandboxes as well as for each pair of sandboxes by providing the cumulative distribution function (CDF). The CDF calculates the percentage of samples with evasion scores lower than or equal to each possible score.

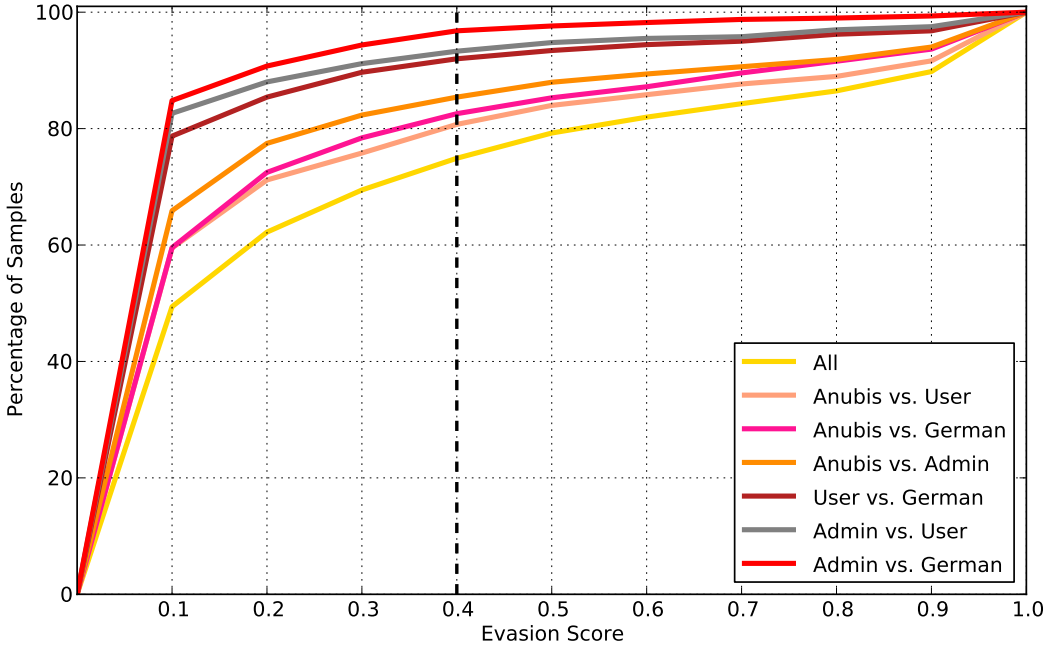

Figure 5.8: CDF of evasion scores for the test dataset overall and per sandbox pairs.

As discussed in Section 5.1, for a sample's overall evasion score we consider the maximum evasion score obtained by comparing executions from any pair of sandboxes. Therefore, we observed the largest amount of samples resulting in evasion scores above the threshold when comparing all sandboxes, which considers all classes of different behavior. We further observed considerably more samples with higher evasion scores for comparisons between the *Anubis* sandbox and sandboxes running the driver. The least amount of samples resulted in higher evasion scores for comparisons between the *Admin* and *German* sandbox. We use the following reasoning to

explain these results: The sandboxes *Anubis* and *Admin* use the same Windows image. We can therefore solely attribute these results to evasions of one of the monitoring technologies employed. Conversely, different behavior for any combination of *Admin*, *User* and *German* indicates evasion or environment sensitivity related to differences in the Windows environment. The results for *Anubis* vs. *User* and *Anubis* vs. *German* are a combination of both factors and therefore produce the highest rates of evasion.

Table 5.4: Samples with evasion scores above the threshold of 0.4 for each pair of sandboxes.

	Anubis	**Admin**	**User**	**German**
Anubis	-	252 (14.95 %)	333 (19.75 %)	303 (17.97 %)
Admin	-	-	121 (7.18 %)	58 (3.44 %)
User	-	-	-	138 (8.19 %)
German	-	-	-	-

Table 5.5: Samples with at least one different persistent action for each pair of sandboxes.

	Anubis	**Admin**	**User**	**German**
Anubis	391 (23.19 %)	729 (43.24 %)	733 (43.48 %)	755 (44.78 %)
Admin	-	440 (26.10 %)	727 (43.12 %)	730 (43.30 %)
User	-	-	443 (26.28 %)	780 (46.26 %)
German	-	-	-	435 (25.80 %)

Table 5.4 provides the total number and percentage of samples with evasion scores above the threshold of 0.4 for each pair of sandboxes. Note that as we calculate our evasion scores based on the difference between inter-sandbox and intra-sandbox distances, our approach assigns an evasion score of exactly 0 for executions in the same sandbox. We use these results to compare our system to another approach proposed to detect analysis evasion. Chen et al. [16] detect analysis evasion by considering any difference in persistent behavior as a sign of evasion. Table 5.5 demonstrates that simply assuming a malware sample is evading an analysis sandbox whenever two executions differ by as little as a single persistent action leads to misleading results. Chen et al. employed the same Windows image in two different sandboxes and only allowed DNS queries. Consequently, we applied our user and environment normalization and ignored network-related actions to allow for a fair comparison across different Windows images and to their methodology. Nevertheless, close to one quarter of samples display some difference in behavior even among executions in the same sandbox. The problem is further exacerbated when considering different sandboxes: Overall 990 samples (58.72 %) behave differently in at least one sandbox according to this criterion.

5.5 Qualitative Results

To get a feeling for differences causing a high ranking in our behavior comparison, we manually examined randomly selected samples with an evasion score above our threshold. The following examples illustrate how DISARM succeeded in uncovering new methods used by malware to detect the presence of Anubis as well as pointing out ways to improve the configuration of the Windows images in our analysis sandboxes.

Anubis Evasion

We found several timing-sensitive samples, that evade analysis by detecting that the environment in which they are running is much slower than a real system. For this, they use the `GetTickCount` API call before and after executing a loop of several benign calls such as `NtDelayExecution` (the native sleep function) or `GetCommandLine`. If the difference exceeds a threshold, these samples either exit immediately or sleep until the analysis timeout occurs without exhibiting malicious behavior. We are able to detect these samples because our in-the-box monitoring technology, despite running inside Qemu, is much faster than Anubis, as we demonstrated in Section 5.1. Therefore, it does not exceed the specified thresholds. This further highlights the importance of evasion detection techniques that are compatible with light-weight instrumentation. Among these samples were representatives of several well known malware families such as Zeus, Virut, Agent and Rbot.

Win32.Packed.Krap.ag.5 and *Win32.TrojanSpy.Banker.9* evade Anubis by checking if the parent process is `explorer.exe`. In Anubis samples are started by an analysis daemon, which terminates afterwards. Thus, this condition is not met and these samples exit without performing any malicious activity.

To render known evasion techniques ineffective, Anubis already randomizes a number of identifiers at each execution. These identifiers are characteristics of the Anubis image that were already detected as being used for evasion. The randomization is implemented by intercepting the calls to these known sources for Anubis-specific characteristics and forging the results:

- `NtQueryValueKey`:
  Windows product ID, computer name, hardware profile GUID
- `NtQueryVolumeInformationFile`:
  volume serial number
- `NtDeviceIoControlFile`:
  disc product ID and serial, MAC address
- `NtQuerySystemInformation`:
  running processes specific to the analysis environment, e.g. `popupKiller.exe`[3]

[3]We run this application in all analysis sandboxes to automatically close popups created by the malware that could otherwise interrupt the malware's execution. Furthermore, this application creates screenshots that enrich our analysis reports.

Inspecting samples detected by DISARM, however, reveals that this feature needs to be improved. Several samples query the installation-specific machine GUID stored in the registry key `HKLM\SOFTWARE\MICROSOFT\CRYPTOGRAPHY\MACHINEGUID`, which is not yet covered by our randomization.[4] In other cases, the randomization was insufficient to prevent evasion. *Trojan-Clicker.Win32.Wistler.d* detected Anubis by querying the computer name. Currently our computer names have the format "pc" followed by a random number. As AV Tracker's [33] listing of publicly available sandboxes also includes the computer name, it is easy for an attacker to deduce this mechanism. Clearly, we need to implement stronger randomization of this identifier or use a name that is too generic to be targeted by evasion methods. Finally, malware can also detect Anubis by checking the hard drive manufacturer information. The randomization feature of Anubis already intercepts the device control code `IOCTL_STORAGE_QUERY_PROPERTY` and the system call `NtQueryVolumeInformationFile` and forges the return information. Some samples, however, were able to bypass this randomization by instead using the device control code `DFP_RECEIVE_DRIVE_DATA` to retrieve the hard drive serial number and manufacturer.

DISARM also detected further samples that performed no malicious actions in Anubis, while performing malicious actions in sandboxes running the driver. We were unable to locate the cause of this behavior in the logs. These samples included additional anti-debugging mechanisms that hindered static analysis. A more thorough investigation would exceed the scope of this evaluation and this thesis. In the future, we could leverage a number of recent techniques [9, 29], which are specifically designed to find the root cause of deviations in behavior.

Environment Sensitivity

The results of our evaluation also exposed various configuration flaws in the image currently used in Anubis. In this image, third party extensions for Internet Explorer are disabled. *AdWare.Win32.InstantBuzz* queries this setting and terminates with a popup asking the user to enable browser extensions. Four samples, e.g. *Trojan.Win32.Powp.gen*, infect the system by replacing the Java Update Scheduler. Clearly, they can only show this behavior in the sandboxes in which the Java Runtime Environment is installed. Microsoft Office is only installed in one of our sandboxes and is targeted by *Worm.Win32.Mixor*. *P2P-Worm.Win32.Tibick.c* queries the registry for the presence of a file-sharing application and fails on images where the Kazaa file-sharing program is not installed. Using this insight we are able to modify the image used in Anubis in order to observe a wider variety of malware behavior.

We already observed in the training dataset, that various samples depend on hardcoded English paths and therefore do not work on the German environment. Samples explicitly looking for `C:\Program Files` exit and do not perform malicious behavior. Furthermore, the German image used for our evaluation uses an older Microsoft Visual C++ Run-Time than the other environments. Samples depending on the newer version therefore do not work with this image.

[4] Note that this is a different identifier than the hardware GUID, which Anubis already randomizes.

Driver Evasion

We prevent samples from loading drivers in order to maintain the integrity of our kernel module. Nonetheless, we found samples that not only detect our logging mechanism, but also actively tamper with our SSDT hooks. At least 20 samples employ mechanisms to restore the hooks to their original addresses and therefore disable the logging in the driver. This can be done from user space by directly accessing `\device\physicalmemory` and restoring the values in the SSDT with the original values read from the `ntoskrnl.exe` disk image [63]. Another ten samples achieve the same effect by using the undocumented function `NtSystemDebugControl` to directly access kernel memory [70]. These techniques are employed by several popular malware families such as Palevo/Butterfly, Bredolab, GameThief and Bifrose, probably as a countermeasure against Anti-Virus solutions.

By disabling access to kernel memory, not granting the `SeDebugPrivilege` necessary for calls to `NtSystemDebugControl` and instrumenting additional system calls, we can harden our driver against such techniques, as long as the kernel is not vulnerable to privilege-escalation vulnerabilities.

False Positives

False positives were caused by samples from the Sality family. This virus creates registry keys and sets registry values whose name depends on the currently logged in user:

- `HKCU\SOFTWARE\AASPPAPMMXKVS\A1_0` for “Administrator”
- `HKCU\SOFTWARE\APCR\U1_0` for “User”

This behavior is not random and not directly related to the username and therefore undetected by our normalization. This behavior, however, is specific to this malware family and can easily be filtered from our results.

CHAPTER 6

Future Work

Our results have shown that DISARM is capable of detecting evasion techniques in current real-world malware samples. However, a determined attacker could build samples that evade the analysis in ways our current system cannot detect. In this chapter we discuss this limitation and propose possible countermeasures. We further discuss enhancements to our approach that can be addressed in future work.

First of all, DISARM works under the premise that a malware sample exhibits its malicious behavior in at least one of the sandboxes. A sample that is able to detect and evade the analysis in all sandboxes raises no suspicion. Therefore, it is crucial to eliminate sandbox characteristics that are shared across all sandboxes. For this purpose, we deployed sandboxes with two different monitoring technologies and three different Windows environments.

One immutable characteristic all sandboxes in our evaluation share is Qemu as the underlying emulation technique. Thus, we are presently unable to detect malware leveraging CPU emulation bugs [54] that are present in both of the Qemu versions we deployed. Furthermore, Qemu imposes a detectable performance overhead and is therefore vulnerable to timing attacks. As we showed in Section 5.5, our in-the-box monitoring technology, even running inside Qemu, was fast enough to escape timing-based detection from some malware samples. However, it is likely that malware can detect the presence of all sandboxes with more aggressive timing attacks. Currently, we are using Qemu in all our sandboxes to facilitate the deployment and integration in our existing analysis infrastructure. Nevertheless, as the proposed in-the-box monitoring approach works independently of the emulation technique, adding new environments to DISARM is straight forward. Therefore, we can deploy our in-the-box monitoring technology on a physical host in the future in order to thwart evasion techniques targeting emulation.

A second immutable characteristic of all sandboxes is the network configuration. We perform the analysis in a very restricted network environment to prevent malware from engaging in harmful activities such as sending spam mails, performing Denial of Service attacks or exploiting vulnerable hosts. A sophisticated attacker could detect these network restrictions and evade the analysis in all sandboxes. Furthermore, evasion techniques that are based on identifying and

blacklisting our public IP addresses [68] would be currently successful against DISARM. To address this problem, we plan to configure our sandboxes to employ a large and dynamic pool of public IP addresses. These IP addresses can be obtained from commercial proxy services or from ISPs that provide dynamic IP addresses to their customers. We could also consider the usage of an anonymity network such as The Onion Router (TOR) [18]. Experimenting with analysis runs without network connectivity could further reveal a malware's sensitivity to restrictions in the network environment.

Malware authors aware of the specifics of our system could also attack DISARM by trying to decrease the evasion score, either by increasing the intra-sandbox distance or decreasing the inter-sandbox distance. The former can be achieved by adding non-deterministic, i.e. randomized, behavior. However, implementing truly randomized behavior might lead to reliability and robustness issues for malware authors. Unstable malware installations are likely to raise suspicion, lead to fast removal from a system or increase attention from malware analysts — three outcomes unfavorable to an attacker. Conversely, malware authors could try to add a number of identical features to the execution on all sandboxes. This would decrease the inter-sandbox distance since our distance measure considers the ratio of the intersection to the union of two profiles. To defeat this attack, we could experiment with distances calculated from the set difference of each pair of profiles, rather than from their Jaccard distance. Furthermore, Anti-Virus solutions incorporate dynamic behavior detectors into their engines, which classify malware based on known malicious behavioral patterns. A large number of irrelevant behavior could thus greatly simplify detection and is not in favor of the attacker.

The current implementation of the in-the-box monitoring system relies solely on program inspection from the kernel's perspective. In our experiments, we discovered malware samples that were, despite our precautions against driver loading, able to access kernel memory and circumvent our kernel driver. We can harden our system against these attacks by instrumenting additional system calls to disable access to the kernel. We could further patch the original SSDT addresses in the disk image of `ntoskrnl.exe` at run time in order to conceal the presence of our SSDT hooks.

Currently, the generated behavioral profiles are imprecise in a number of situations. For example, the driver currently lacks the ability to intercept calls to the Services API, which is implemented by means of complex RPC communication that our kernel driver is unable to understand. We already extract important information following Services API invocations, such as the process of a service being started. Nevertheless, the driver should be extended to intercept RPC communication from the kernel level, or employ user-land hooking techniques [26, 67] to improve profile precision. Furthermore, the driver currently does not intercept API calls that are used by keyloggers to record pressed keys, nor do we represent these functions in our behavioral profiles. We could extend the definition of our profiles and also intercept these calls with user-land hooking techniques. Another feature we currently do not consider in our comparison is the creation of mutexes. Malware samples might create a mutex with a specific name to mark its presence on a system and thereby avoid multiple infections of the same system or even to vaccinate a system against infection by other malware strains [60]. We could extend the definition of synchronization resources in the behavioral profiles to differentiate between semaphores and named or unnamed mutexes and consider the creation of named mutexes as a persistent feature.

In this thesis we focused on detecting the presence of evasion techniques in malware without automatically investigating the nature of these techniques. We could automatically run samples that DISARM identified as evading the analysis with tools designed to detect the root cause of deviations in behavior such as the tools presented in [9, 29]. We could also enhance DISARM to automatically classify samples by detecting patterns in behavior. We can extract patterns from malware behavior that we already detected as evasions and automatically classify new samples that show these patterns as evasions. We can also define sources for information that can be used by malware to identify a sandbox and flag samples that access this sources as suspicious. For example, as we discussed in Chapter 2, malware samples use the Windows product ID to detect analysis sandboxes. As benign applications are unlikely to query this information, we can classify the access to registry keys containing the Windows product ID as signs of evasive behavior.

We demonstrated the capability of DISARM to detect evasion techniques in current malware samples. Additionally to applying our findings to Anubis in order to prevent these evasion techniques in the future, we could also maintain a deployment of DISARM as a permanent verification tool for Anubis. Thereby, we could detect new evasion techniques as they emerge and further harden Anubis against these attacks.

CHAPTER 7

Conclusion

Dynamic malware analysis systems are widely used by security researchers and Anti-Virus vendors in the fight against the vast amount of malware samples emerging every day. As a consequence, malware samples try to evade analysis and thereby detection by refusing to perform malicious activities when they are running inside an analysis sandbox instead of on a real user's system. These "*environment-sensitive*" malware samples detect the presence of an analysis sandbox either by detecting the monitoring technology employed by the sandbox or by identifying characteristics of a specific Windows environment that is used for analysis. In the absence of an "undetectable", fully transparent analysis sandbox, defense against sandbox evasion is mostly reactive: Sandbox developers and operators tweak their systems to thwart individual evasion techniques as they become aware of them, leading to a never-ending arms race.

In order to provide more automation in this arms race, we introduced DISARM, a system that automatically screens malware samples for evasive behavior. By comparing the behavior of malware across multiple analysis sandboxes that employ different monitoring technologies and Windows environments, DISARM can detect analysis evasion irrespective of the root cause of the divergence in behavior. We introduced novel techniques for normalizing and comparing behavior observed in different sandboxes, which discard spurious differences. In order to accurately detect samples exhibiting semantically different behavior, we further proposed an evasion score that uses behavior variations within a sandbox as well as between sandboxes. Furthermore, we implemented a light-weight in-the-box execution monitoring system that can be applied to any Windows XP environment. Nevertheless, DISARM is compatible with any in-the-box or out-of-the-box monitoring technology as long as it is able to detect persistent changes to the system state.

We evaluated DISARM against over 1,500 malware samples in four different analysis sandboxes using two different monitoring technologies. As a result, we discovered several new evasion techniques currently in use by malware. We can use our findings to prevent these kinds of evasion techniques against Anubis in the future. We further discovered ways in which we can improve the configuration of Anubis to observe a wider variety of malware behavior.

DISARM succeeded in automatically detecting deviations in malware behavior with a high degree of accuracy. Our behavior normalization allows us to filter out differences that are unrelated to malware behavior. Considering variations within a sandbox as a baseline for behavior variations between sandboxes enables us to mitigate differences caused by non-deterministic behavior. By implementing a portable in-the-box execution monitoring technology we can further extend our sandbox setup and also perform execution monitoring on a physical host in the future. This would allow us to detect even more evasion techniques. We therefore propose DISARM as a permanent verification tool for Anubis to automatically detect new evasion techniques as they emerge.

APPENDIX A

Appendix

A.1 Hooked System Calls

```
 0 NtAcceptConnectPort
 1 NtAccessCheck
 2 NtAccessCheckAndAuditAlarm
 3 NtAccessCheckByType
 4 NtAccessCheckByTypeAndAuditAlarm
 5 NtAccessCheckByTypeResultList
 6 NtAccessCheckByTypeResultListAndAuditAlarm
 7 NtAccessCheckByTypeResultListAndAuditAlarmByHandle
 8 NtAddAtom
 9 NtAddBootEntry
10 NtAdjustGroupsToken
11 NtAdjustPrivilegesToken
12 NtAlertResumeThread
13 NtAlertThread
14 NtAllocateLocallyUniqueId
15 NtAllocateUserPhysicalPages
16 NtAllocateUuids
17 NtAllocateVirtualMemory
18 NtAreMappedFilesTheSame
19 NtAssignProcessToJobObject
20 NtCallbackReturn
21 NtCancelDeviceWakeupRequest
22 NtCancelIoFile
23 NtCancelTimer
24 NtClearEvent
25 NtClose
26 NtCloseObjectAuditAlarm
27 NtCompactKeys
28 NtCompareTokens
29 NtCompleteConnectPort
30 NtCompressKey
31 NtConnectPort
32 NtContinue
33 NtCreateDebugObject
34 NtCreateDirectoryObject
35 NtCreateEvent
36 NtCreateEventPair
37 NtCreateFile
38 NtCreateIoCompletion
39 NtCreateJobObject
40 NtCreateJobSet
41 NtCreateKey
42 NtCreateMailslotFile
43 NtCreateMutant
44 NtCreateNamedPipeFile
45 NtCreatePagingFile
46 NtCreatePort
47 NtCreateProcess
48 NtCreateProcessEx
49 NtCreateProfile
50 NtCreateSection
51 NtCreateSemaphore
52 NtCreateSymbolicLinkObject
53 NtCreateThread
54 NtCreateTimer
55 NtCreateToken
56 NtCreateWaitablePort
57 NtDebugActiveProcess
58 NtDebugContinue
59 NtDelayExecution
60 NtDeleteAtom
61 NtDeleteBootEntry
62 NtDeleteFile
63 NtDeleteKey
64 NtDeleteObjectAuditAlarm
65 NtDeleteValueKey
66 NtDeviceIoControlFile
67 NtDisplayString
68 NtDuplicateObject
69 NtDuplicateToken
70 NtEnumerateBootEntries
71 NtEnumerateKey
72 NtEnumerateSystemEnvironmentValuesEx
73 NtEnumerateValueKey
74 NtExtendSection
75 NtFilterToken
76 NtFindAtom
77 NtFlushBuffersFile
78 NtFlushInstructionCache
79 NtFlushKey
80 NtFlushVirtualMemory
81 NtFlushWriteBuffer
```

```
 82 NtFreeUserPhysicalPages
 83 NtFreeVirtualMemory
 84 NtFsControlFile
 85 NtGetContextThread
 86 NtGetDevicePowerState
 87 NtGetPlugPlayEvent
 88 NtGetWriteWatch
 89 NtImpersonateAnonymousToken
 90 NtImpersonateClientOfPort
 91 NtImpersonateThread
 92 NtInitializeRegistry
 93 NtInitiatePowerAction
 94 NtIsProcessInJob
 95 NtIsSystemResumeAutomatic
 96 NtListenPort
 97 NtLoadDriver
 98 NtLoadKey
 99 NtLoadKey2
100 NtLockFile
101 NtLockProductActivationKeys
102 NtLockRegistryKey
103 NtLockVirtualMemory
104 NtMakePermanentObject
105 NtMakeTemporaryObject
106 NtMapUserPhysicalPages
107 NtMapUserPhysicalPagesScatter
108 NtMapViewOfSection
109 NtModifyBootEntry
110 NtNotifyChangeDirectoryFile
111 NtNotifyChangeKey
112 NtNotifyChangeMultipleKeys
113 NtOpenDirectoryObject
114 NtOpenEvent
115 NtOpenEventPair
116 NtOpenFile
117 NtOpenIoCompletion
118 NtOpenJobObject
119 NtOpenKey
120 NtOpenMutant
121 NtOpenObjectAuditAlarm
122 NtOpenProcess
123 NtOpenProcessToken
124 NtOpenProcessTokenEx
125 NtOpenSection
126 NtOpenSemaphore
127 NtOpenSymbolicLinkObject
128 NtOpenThread
129 NtOpenThreadToken
130 NtOpenThreadTokenEx
131 NtOpenTimer
132 NtPlugPlayControl
133 NtPowerInformation
134 NtPrivilegeCheck
135 NtPrivilegeObjectAuditAlarm
136 NtPrivilegedServiceAuditAlarm
137 NtProtectVirtualMemory
138 NtPulseEvent
139 NtQueryAttributesFile
140 NtQueryBootEntryOrder
141 NtQueryBootOptions
142 NtQueryDebugFilterState
143 NtQueryDefaultLocale
144 NtQueryDefaultUILanguage
145 NtQueryDirectoryFile
146 NtQueryDirectoryObject
147 NtQueryEaFile
148 NtQueryEvent
149 NtQueryFullAttributesFile
150 NtQueryInformationAtom
151 NtQueryInformationFile
152 NtQueryInformationJobObject
153 NtQueryInformationPort
154 NtQueryInformationProcess
155 NtQueryInformationThread
156 NtQueryInformationToken
157 NtQueryInstallUILanguage
158 NtQueryIntervalProfile
159 NtQueryIoCompletion
160 NtQueryKey
161 NtQueryMultipleValueKey
162 NtQueryMutant
163 NtQueryObject
164 NtQueryOpenSubKeys
165 NtQueryPerformanceCounter
166 NtQueryQuotaInformationFile
167 NtQuerySection
168 NtQuerySecurityObject
169 NtQuerySemaphore
170 NtQuerySymbolicLinkObject
171 NtQuerySystemEnvironmentValue
172 NtQuerySystemEnvironmentValueEx
173 NtQuerySystemInformation
174 NtQuerySystemTime
175 NtQueryTimer
176 NtQueryTimerResolution
177 NtQueryValueKey
178 NtQueryVirtualMemory
179 NtQueryVolumeInformationFile
180 NtQueueApcThread
181 NtRaiseException
182 NtRaiseHardError
183 NtReadFile
184 NtReadFileScatter
185 NtReadRequestData
186 NtReadVirtualMemory
187 NtRegisterThreadTerminatePort
188 NtReleaseMutant
189 NtReleaseSemaphore
190 NtRemoveIoCompletion
191 NtRemoveProcessDebug
192 NtRenameKey
193 NtReplaceKey
194 NtReplyPort
195 NtReplyWaitReceivePort
196 NtReplyWaitReceivePortEx
197 NtReplyWaitReplyPort
198 NtRequestDeviceWakeup
199 NtRequestPort
200 NtRequestWaitReplyPort
201 NtRequestWakeupLatency
202 NtResetEvent
203 NtResetWriteWatch
204 NtRestoreKey
205 NtResumeProcess
206 NtResumeThread
207 NtSaveKey
208 NtSaveKeyEx
209 NtSaveMergedKeys
210 NtSecureConnectPort
211 NtSetBootEntryOrder
212 NtSetBootOptions
213 NtSetContextThread
214 NtSetDebugFilterState
215 NtSetDefaultHardErrorPort
216 NtSetDefaultLocale
217 NtSetDefaultUILanguage
218 NtSetEaFile
219 NtSetEvent
220 NtSetEventBoostPriority
221 NtSetHighEventPair
222 NtSetHighWaitLowEventPair
223 NtSetInformationDebugObject
224 NtSetInformationFile
225 NtSetInformationJobObject
226 NtSetInformationKey
227 NtSetInformationObject
```

228 NtSetInformationProcess
229 NtSetInformationThread
230 NtSetInformationToken
231 NtSetIntervalProfile
232 NtSetIoCompletion
233 NtSetLdtEntries
234 NtSetLowEventPair
235 NtSetLowWaitHighEventPair
236 NtSetQuotaInformationFile
237 NtSetSecurityObject
238 NtSetSystemEnvironmentValue
239 NtSetSystemEnvironmentValueEx
240 NtSetSystemInformation
241 NtSetSystemPowerState
242 NtSetSystemTime
243 NtSetThreadExecutionState
244 NtSetTimer
245 NtSetTimerResolution
246 NtSetUuidSeed
247 NtSetValueKey
248 NtSetVolumeInformationFile
249 NtShutdownSystem
250 NtSignalAndWaitForSingleObject
251 NtStartProfile
252 NtStopProfile
253 NtSuspendProcess
254 NtSuspendThread
255 NtSystemDebugControl
256 NtTerminateJobObject
257 NtTerminateProcess
258 NtTerminateThread
259 NtTestAlert
260 NtTraceEvent
261 NtTranslateFilePath
262 NtUnloadDriver
263 NtUnloadKey
264 NtUnloadKeyEx
265 NtUnlockFile
266 NtUnlockVirtualMemory
267 NtUnmapViewOfSection
268 NtVdmControl
269 NtWaitForDebugEvent
270 NtWaitForMultipleObjects
271 NtWaitForSingleObject
272 NtWaitHighEventPair
273 NtWaitLowEventPair
274 NtWriteFile
275 NtWriteFileGather
276 NtWriteRequestData
277 NtWriteVirtualMemory
278 NtYieldExecution
279 NtCreateKeyedEvent
280 NtOpenKeyedEvent
281 NtReleaseKeyedEvent
282 NtWaitForKeyedEvent
283 NtQueryPortInformationProcess

Bibliography

[1] Anubis. http://anubis.iseclab.org/, 2010.

[2] GnuWin. http://gnuwin32.sourceforge.net/, 2010.

[3] Joe Sandbox. http://www.joesecurity.org/, 2011.

[4] Norman Sandbox. http://www.norman.com/business/sandbox_malware_analyzers/, 2011.

[5] QEMU. http://wiki.qemu.org/, 2011.

[6] ThreatExpert. http://www.threatexpert.com/, 2011.

[7] VirusTotal. http://www.virustotal.com/, 2011.

[8] Michael Bailey, Jon Oberheide, Jon Andersen, Zhuoqing 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)*, 2007.

[9] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[10] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[11] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A View on Current Malware Behaviors. In *Proceedings of the 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[12] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. Improving the Efficiency of Dynamic Malware Analysis. In *Proceedings of the 25th ACM Symposium on Applied Computing (SAC)*, 2010.

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

[14] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic Analysis of Malicious Code. *Journal in Computer Virology*, 2(1), 2006.

[15] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2005.

[16] Xu Chen, Jon Andersen, Zhuoqing Morley Mao, Michael Bailey, and Jose Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN)*, 2008.

[17] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.

[18] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[19] J.C. Dunn. Well Separated Clusters and Optimal Fuzzy Partitions. *Journal of Cybernetics*, 4, 1974.

[20] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A Survey on Automated Dynamic Malware Analysis Techniques and Tools. *ACM Computing Surveys Journal*, (to appear).

[21] Peter Ferrie. Attacks on Virtual Machine Emulators. Technical report, Symantec Research White Paper, 2006.

[22] Peter Ferrie. Attacks on More Virtual Machines. Technical report, Symantec Research White Paper, 2007.

[23] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS)*, 2007.

[24] Greg Hoglund and James Butler. *Rootkits: Subverting the Windows kernel*. Addison-Wesley Professional, 2005.

[25] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-Scale Malware Indexing Using Function-Call Graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[26] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, 1998.

[27] Internet Malware Analyse System (InMAS). NE1: Reliable Sandbox Setup. `http://mwanalysis.org/inmas/paper/2008-InMAS-Sandbox.pdf`, 2008.

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

[29] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, 2011.

[30] Vitaly Kamluk. A black hat loses control. `http://www.securelist.com/en/weblog?weblogid=208187881`, 2009.

[31] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A Hidden Code Extractor for Packed Executables. In *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM)*, 2007.

[32] Min Gyung Kang, Heng Yin, Steve Hanna, Steve McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec)*, 2009.

[33] Peter Kleissner. Antivirus tracker. `http://avtracker.info/`, 2009.

[34] McAfee Labs. McAfee Threats Report: Third Quarter 2010. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q3-2010.pdf`, 2010.

[35] McAfee Labs. McAfee Threats Report: Fourth Quarter 2010. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2010.pdf`, 2011.

[36] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. AccessMiner: Using System-Centric Models for Malware Protection. In *Proceedigns of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[37] Boris Lau and Vanja Svajcer. Measuring virtual machine detection in malware using DSD tracer. *Journal in Computer Virology*, 6(3), 2010.

[38] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[39] Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. A Framework for Behavior-Based Malware Analysis in the Cloud. In *Proceedings of the 5th International Conference on Information Systems Security (ICISS)*, 2009.

[40] Microsoft. Description of the Dr. Watson for Windows (Drwtsn32.exe) Tool. http://support.microsoft.com/kb/308538, 2007.

[41] Microsoft. Windows API Reference. http://msdn.microsoft.com/en-us/library/aa383749(v=VS.85).aspx, 2010.

[42] Microsoft. Microsoft Windows Software Development Kit. http://msdn.microsoft.com/en-us/windowsserver/bb980924.aspx, 2011.

[43] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the 28th IEEE Symposium on Security and Privacy*, 2007.

[44] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of Static Analysis for Malware Detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[45] Gery Nebbett. *Windows NT/2000 Native API Reference*. New Riders Publishing, 2000.

[46] NTinternals. The Undocumented Functions: Microsoft Windows NT/2K/XP/2003. http://undocumented.ntinternals.net, 2008.

[47] NVlabs. Loading drivers and Native applications from kernel mode, without touching registry. http://www.nvlabs.in/archives/6-Loading-drivers-and-Native-applications-from-kernel-mode,-without-touching-registry.html, 2007.

[48] Tavis Ormandy. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. http://taviso.decsystem.org/virtsec.pdf, 2008.

[49] Roberto Paleari, Lorenzo Martignoni, Emanuele Passerini, Drew Davidson, Matt Fredrikson, Jon Giffin, and Somesh Jha. Automatic Generation of Remediation Procedures for Malware Infections. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[50] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[51] PandaLabs. Creation of New Malware Increases by 26 Percent to Reach More than 73,000 Samples Every Day, PandaLabs reports. http://press.pandasecurity.com/news/creation-of-new-malware-increases-by-26-percent-to-reach-more-than-73000-samples-every-day-pandalabs-reports, 2011.

[52] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. nEther: In-guest Detection of Out-of-the-guest Malware Analyzers. In *Proceedings of the 4th ACM European Workshop on System Security (EUROSEC)*, 2011.

[53] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010.

[54] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting System Emulators. In *Proceedings of the 10th Information Security Conference (ISC)*, 2007.

[55] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and Classification of Malware Behavior. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[56] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic Analysis of Malware Behavior using Machine Learning. Technical Report 18–2009, Berlin Institute of Technology, December 2009.

[57] Stephen E. Robertson and Karen Sparck Jones. Relevance Weighting of Search Terms. *Journal of the American Society for Information Science*, 27(3):129–146, 1976.

[58] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, 2004.

[59] Joanna Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://invisiblethings.org/papers/redpill.html`, 2004.

[60] Andreas Schuster. Searching for mutants. `http://computer.forensikblog.de/en/2009/04/searching_for_mutants.html`, 2009.

[61] Securelist. Rules for naming detected objects. `http://www.securelist.com/en/threats/detect?chapter=136`, 2011.

[62] Brett Stone-Gross, Andreas Moser, Christopher Kruegel, Kevin Almaroth, and Engin Kirda. FIRE: FInding Rogue nEtworks. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, 2009.

[63] Chew Keong Tan. Defeating Kernel Native API Hookers by Direct Service Dispatch Table Restoration. Technical report, SIG2 G-TEC Lab, 2004.

[64] The Honeynet Project. Know Your Enemy: Fast-Flux Service Networks. `http://www.honeynet.org/papers/ff`, 2007.

[65] Philipp Trinius, Carsten Willems, Thorsten Holz, and Konrad Rieck. A Malware Instruction Set for Behavior-Based Analysis. Technical Report 07–2009, University of Mannheim, 2009.

[66] Amit Vasudevan and Ramesh Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized-executions. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, 2006.

[67] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2), 2007.

[68] Katsunari Yoshioka, Yoshihiko Hosobuchi, Tatsunori Orii, and Tsutomu Matsumoto. Your Sandbox is Blinded: Impact of Decoy Injection to Public Malware Analysis Systems. *Journal of Information Processing*, 19, 2011.

[69] Lenny Zeltser. Tools for restoring a system's state. `http://isc.sans.edu/diary.html?storyid=4147`, 2008.

[70] Jiayuan Zhang, Shufen Liu, Jun Peng, and Aijie Guan. Techniques of user-mode detecting System Service Descriptor Table. In *Proceedings of the 13th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 2009.

[71] Jianwei Zhuge, Thorsten Holz, Chengyu Song, Jinpeng Guo, Xinhui Han, and Wei Zou. Studying Malicious Websites and the Underground Economy on the Chinese Web. In *Proceedings of the 7th Workshop on the Economics of Information Security (WEIS)*, 2008.