SEVENTH FRAMEWORK PROGRAMME
Theme ICT-1-1.4 (Secure, dependable and trusted infrastructures)

WORLDWIDE OBSERVATORY OF
MALICIOUS BEHAVIORS AND ATTACK THREATS

D16 (D4.2) Analysis Report of Behavioral Features

Contract No. FP7-ICT-216026-WOMBAT

Workpackage	WP4 - Data Enrichment and Characterization
Author	Herbert Bos
Version	1.0
Date of delivery	M27
Actual Date of Delivery	M28
Dissemination level	Public
Responsible	VU

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n°216026.

SEVENTH FRAMEWORK PROGRAMME

Theme ICT-1-1.4 (Secure, dependable and trusted infrastructures)

The WOMBAT Consortium consists of:

France Telecom	Project coordinator	France
Institut Eurecom		France
Technical University Vienna		Austria
Politecnico di Milano		Italy
Vrije Universiteit Amsterdam		The Netherlands
Foundation for Research and Technology		Greece
Hispasec		Spain
Research and Academic Computer Network		Poland
Symantec Ltd.		Ireland
Institute for Infocomm Research		Singapore

Contact information:
Dr. Marc Dacier
2229 Route des Cretes
06560 Sophia Antipolis
France

e-mail: Marc_Dacier@symantec.com
Phone: +33 4 93 00 82 17

Contents

Abstract

This deliverable provides a discussion of the features used to characterize the behavior of code, and a discussion of preliminary results of applying these features to a set of malicious code. It discusses the project's results in behavior-based clustering, malware detection at end hosts in different ways, system call analysis, but also our work on shellcode behavior.

1 Introduction

This deliverable provides a discussion of the features used to characterize the behavior of code, and a discussion of preliminary results of applying these features to a set of malicious code.

We describe techniques to enrich and characterize the malicious code that is collected in WP3. The main idea is to enrich the collected code - with the help of metadata that might provide insights into the origin of the code, as well as the intentions of those that created, released or used it. In this deliverable and indeed the entire work package, we deliberately use the term 'code' in a relatively broad sense. It is not limited to malware samples that are available as binary executables, but also takes into account any other element that is sent as part of an attack process. The reason is that all these elements share a common characteristic. They aim at having certain actions performed for the benefit of an adversary, without the conscious consent of the use. The form in which the malicious code is delivered is not particularly relevant, and our techniques can be equally applied to malicious code in different forms.

In this deliverable we are interested in the code's behavior. Models of program behavior are used frequently in host-based intrusion detection - e.g., in the form of system call anomaly detection. In this context, behavioral models are used to protect the program against code-injection attacks, by comparing the program execution with the expected behavior. Models that embody the permitted behavior of the program can be derived through static analysis [39, 50, 94] or learning from dynamic traces [40, 44]. As current system typically do not take into account order, dependency, or data flow information regarding system calls is, it is difficult to distinguish between malicious and benign programs in many circumstances. This is confirmed by the high false positive rate that is reported in these approaches.

We define the behavior of a piece of code as the observable effect this code has on its environment. Code behavior can be defined at different levels of granularity, depending on what is considered the program's environment. We have looked at two approaches, one that only considers the effects of code within a process, and one that monitors the interaction of a process and the underlying operating system. Note that in both cases, it is desirable to formalize code behavior (using a formal model or language) in order to precisely describe and reason about malicious actions.

Intra-process Behavior The first approach is to define behavior as the effects a code sequence has on the state of a process. For our purpose, the state of a process is defined as the content of the memory address space. The effect on this state are modifications to the contents of the address space. Characterizing intra-process behavior is useful for the following reasons:

- **Exploit payload detection:** By analyzing the actions of a piece of code on the process memory space, we can identify the actual, nefarious actions within a given exploit payload. This allows us to generate more precise signatures, also in case of polymorphic attacks.
- **Malware characterization:** We can analyze a code sequence for the occurrence of instructions that modify certain memory locations in a fashion that is indicative of a nefarious purpose. For example, we can identify decryption routines in which a number of consecutive locations in memory are modified, a technique often used by viruses and worms.

Interaction with Operating System An alternative approach to specify code behavior is to view the program as a black-box, focusing only on its interaction with the operating system. Such monitoring can be performed by the observation of the operating system calls invoked by this process. One way to use system call information is to characterize code behavior based on the type of system calls this code performs, as well as an analysis of the function arguments.

In addition to the knowledge of the type of operating system calls that are invoked and the arguments that are used, it is also important to have access to a mechanism that provides precise information about data flow dependencies between arguments. To see why this might be important, consider a worm that spreads by sending mails with its own executable as attachment. When analyzing the system calls that this worm performs, we will observe a set of calls that read a file (i.e., the worm program) and a set of calls that send data over the network (to the mail server). Given the additional information that the data sent over the network was previously read from the program's own executable provides additional information that helps to distinguish this suspicious worm behavior from the behavior of a mail client. The reason is that a mail client might also read files (e.g., to load attachments) and send data over the network. Thus, the mail client and the worm cannot be distinguished by looking at the system calls alone.

Outline The remainder of this deliverable is structured as follows. In Chapter 2, we will present a survey of current malware behavior, which serves both as concrete result and as context for the remaining chapters. In Chapter 3, we present evaluation results for

the behavioral malware clustering techniques introduced in Deliverable D08 (D4.1). In Chapter 4, we discuss how we improved a scanner and detection system that worked by way of information flow analysis in the end host. Chapter 5 explains how we monitor the behavior of malware by analysis of the system calls. In Chapter 6, we present malware behavioral detection by grammar-based signatures. Besides the eventual malware we also study the behavior of shellcode. Our progress in this area is presented in Chapter 7. Finally, we conclude this deliverable in Chapter 8.

2 A Survey of Current Malware Behavior

2.1 Introduction

Malicious software (or malware) is one of the most pressing and major security threats facing the Internet today. Anti-virus companies typically have to deal with tens of thousands of new malware samples every day. Because of the limitations of static analysis, dynamic analysis tools are typically used to analyze these samples, with the aim of understanding how they behave and how they launch attacks. This understanding is important to be able to develop effective malware countermeasures and mitigation techniques.

In this survey, we set out to provide insights into common malware behaviors. Our analysis and experiences are based on the malicious code samples that were collected by Anubis [12, 21], our dynamic malware analysis platform. When it receives a sample, Anubis executes the binary and monitors the invocation of important system and Windows API calls, records the network traffic, and tracks data flows. This provides a comprehensive view of malicious activity that is typically not possible when monitoring network traffic alone.

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

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

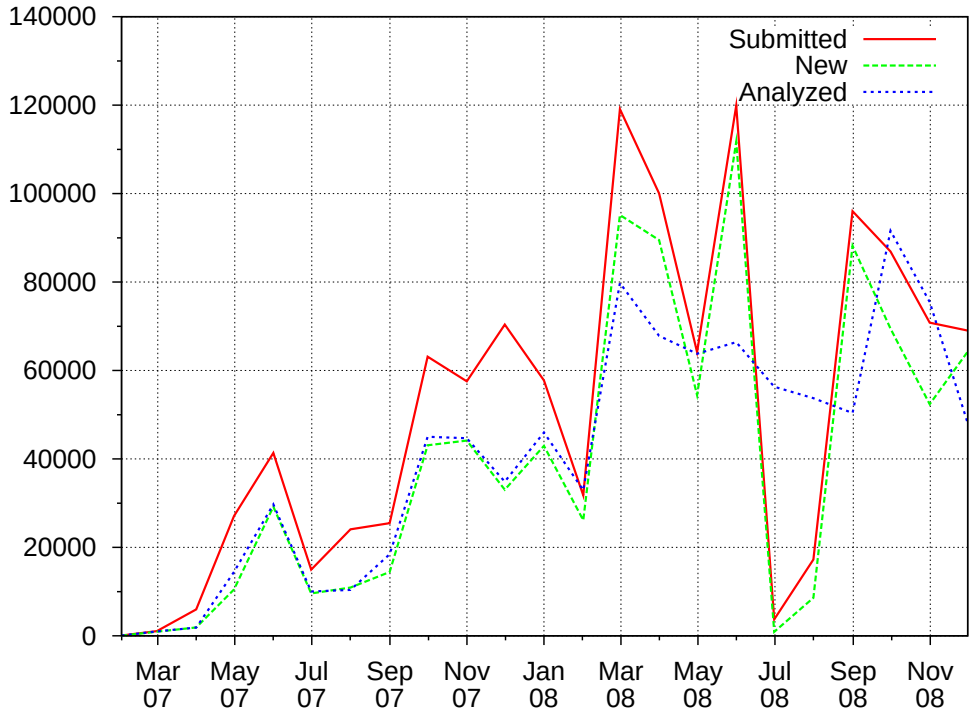

Figure 2.1: Anubis submission statistics.

based on malware families (clusters).

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

2.2 Dataset

In this section, we give a brief overview of the data that Anubis collects. As mentioned previously, a binary under analysis is run in an emulated operating system environment (a modified version of Qemu [23]) and its (security-relevant) actions are monitored. In particular, we record the Windows native system calls and Windows API functions that

the program invokes. One important feature of our system is that it does not modify the program that it executes (e.g., through API call hooking or breakpoints), making it more difficult to detect by malicious code. Also, our tool runs binaries in an unmodified Windows environment, which leads to good emulation accuracy. Each sample is run until all processes are terminated or a timeout of four minutes expires. Once the analysis is finished, the observed actions are compiled in a report and saved to a database.

Our dataset covers the analysis reports of all files that were submitted to Anubis in the period from February 7th 2007 to December 31st 2008, and that were subsequently analyzed by our dynamic analysis system in the time period between February 7th 2007 and January 14th 2009. This dataset contains 901,294 unique samples (based on their MD5 hashes) and covers a total of 1,167,542 submissions. Typically, a given sample is only analyzed once by our analysis system. That is, when a sample is submitted again, we return the already existing analysis report without doing an actual analysis.

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

Naturally, the Anubis tool has evolved over time. We fixed bugs in later versions or added new features. Given that there is a constant stream of new malware samples arriving and the analysis process is costly, we typically do not rerun old samples with each new Anubis version. Unfortunately, this makes it a bit more difficult to combine analysis results that were produced by different versions of Anubis into consolidated statistics. In some cases, it is possible to work around such differences. In other cases (in particular, for the analysis of anti-sandbox techniques presented Section 2.3.6), we had to confine ourselves to results for a smaller subset of 330,088 analyzed PE files. The reason is that necessary information was not present in older reports.

2.2.1 Submissions

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

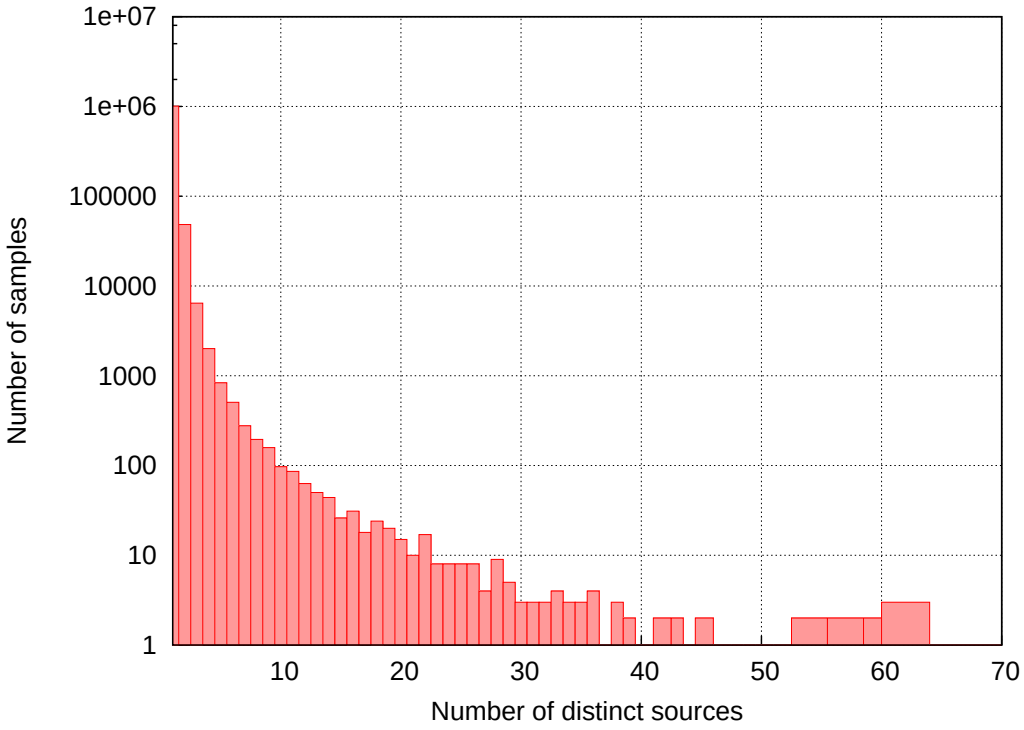

Figure 2.2: Number of distinct sources for each sample.

We have made the experience that measuring the number of sources that submit a certain sample tends to indicate how widespread a certain malware sample is in the wild. In fact, this premise is supported by the results of the anti-virus scans that we run on each sample that we receive. For example, if we consider the samples submitted by one source, 73% of the submissions are categorized by two out of five anti-virus scanners as being malicious. In comparison, 81% of the submissions that are submitted by at least 3 different sources are identified as being malicious by anti-virus software. Furthermore, among the samples that are submitted by 10 or more sources, 91% are identified as being malicious.

2.2.2 Submitted file types

One problem with running an online, public malware analysis service is that one can receive all sorts of data, not only malware. In fact, users might even try to submit

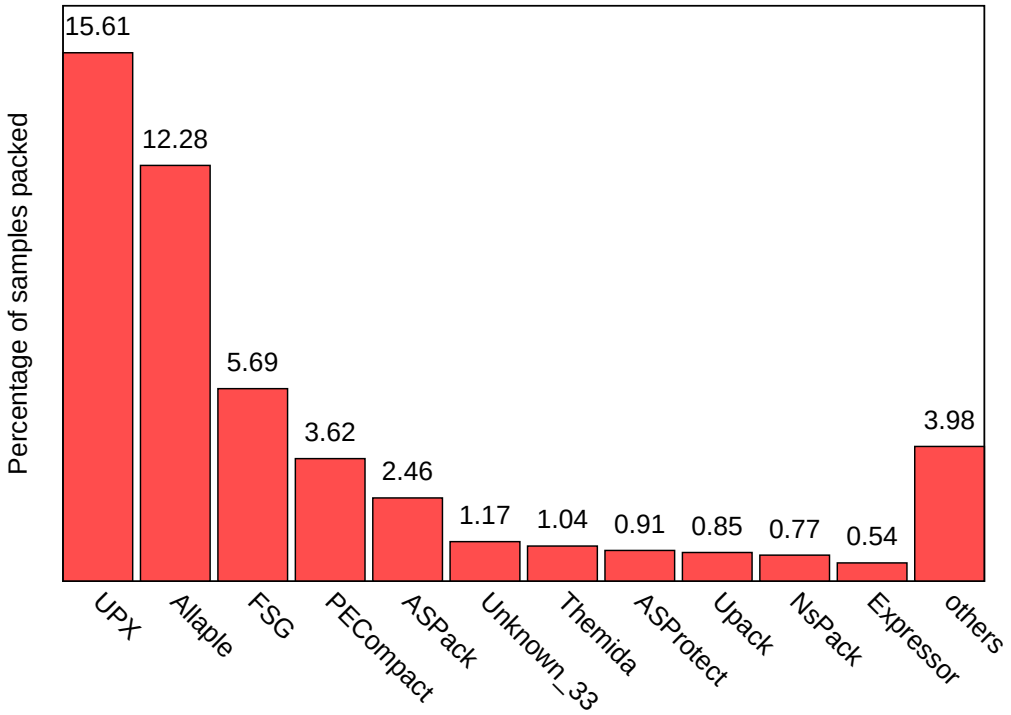

Figure 2.3: Overview of used packers

applications such as Microsoft Word or Microsoft Internet Explorer just to see how the system reacts. Furthermore, unfortunately, not all the submitted samples are valid Windows PE executables [71] (around 14% are not). Table 2.1 shows a breakdown of the different file types submitted to Anubis. As can be seen from this table, fortunately for us, most of the files that are sent to Anubis can be analyzed. The category of non PE files includes mostly different archive formats (ZIP and RAR archives) and MS Office documents (such as Word and Excel), but also a small number of shell scripts and executables for different operating systems (such as DOS, Linux). According to SigBuster, a signature-based scanner for packers, 40.64% of the analyzed PE files are packed. Figure 2.3 provides an overview of the most common packers.

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

Table 2.1: File types submitted to Anubis.

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

Table 2.2: Overview of observed behavior.

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

Table 2.3: Submission sources.

2.2.3 Submission sources

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

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

2.3 Observed Malicious Behavior

In this section, we present detailed discussions on the file, registry, network, and botnet activity that we observed when analyzing the Anubis submissions. The goal is to provide insights into malicious behaviors that are common among a diverse range of malware programs. An overview of interesting activity is shown in Table 2.2. In this table, we show the fraction of samples that perform certain high-level activity. We also provide the behavioral information with respect to the number of malware families, approximated

Autostart Location	Percentage of Samples	Percentage of Clusters
HKLM\System\Currentcontrolset\Services\%\Imagepath	17.53%	11.67%
HKLM\Software\Microsoft\Windows\Currentversion\Run%	16.00%	17.80%
HKLM\Software\Microsoft\Active Setup\Installed Components%	2.50%	2.79%
HKLM\Software\Microsoft\Windows\Currentversion\Explorer\Browser Helper Objects%	1.72%	1.75%
HKLM\Software\Microsoft\Windows\Currentversion\Runonce%	1.60%	3.07%
HKLM\Software\Microsoft\Windows\Currentversion\Explorer\Shellexecutehooks%	1.30%	2.29 %
HKLM\Software\Microsoft\Windows NT\Currentversion\Windows\Appinit_Dlls	1.09%	0.89%
HKLM\Software\Microsoft\Windows NT\Currentversion\Winlogon\Notify%	1.04%	1.89%
HKLM\Software\Microsoft\Windows\Currentversion\Policies\Explorer\Run%	0.67%	1.04%
C:\Documents and Settings\%\Start Menu\Programs\Startup\%	0.20%	0.95%

Table 2.4: Top 10 Autostart locations.

as clusters of samples that exhibit similar behaviors [22]. It is interesting to observe that the differences are often not very pronounced. One reason is that the clustering process was using a tight threshold. That is, samples are only grouped when they exhibit very similar activity, resulting in a large number of clusters. Another reason is that the activity in Table 2.2 is quite generic, and there is not much difference at this level between individual samples and families. The situation changes when looking at activity at a level where individual resources (such as files, registry keys) are considered. For example, 4.49% of all samples create the file `C:\WINDOWS\system32\urdvxc.exe`, but this is true for only 0.54% of all clusters. This file is created by the well-known, polymorphic `allaple` worm, and many of its instances are grouped in a few clusters. Another example can be seen in Table 2.4. Here, 17.53% of all samples use a specific registry key for making the malware persistent. When looking at the granularity of clusters (families), this number drops to 11.67%. Again, the drop is due to the way in which `allaple` operates. It also demonstrates that using statistics based on malware clusters is more robust when large clusters of polymorphic malware samples are present in the dataset.

2.3.1 File system activity

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

When analyzing the *created* files in more detail, we observe that they mostly belong to two main groups: One group contains executable files, typically because the malware copies or moves its binary to a known location (such as the Windows system folder). Often, this binary is a new polymorphic variant. In total, 37.2% of all samples create

at least one executable file. Interestingly, however, only 23.2% of all samples (or 62% of those that drop an executable) choose the `Windows` directory or one of its sub-folders as the target. A large fraction – 15.1% – create the executable in the user's folder (under `Document and Settings`). This is interesting, and might indicate that, increasingly, malware is developed to run successfully with the permissions of a normal user (and hence, cannot modify the system folder).

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

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

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

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

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

2.3.2 Registry activity

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

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

2.3.3 Network activity

Table 2.5 provides an overview of the network activities that we observed during analysis. Figure 2.4 depicts the use of the HTTP, IRC, and SMTP protocols by individual samples over a one and a half year period. In contrast, Figure 2.5 shows the usage of the HTTP, IRC, and SMTP protocols once families of malware samples are clustered together (using our clustering approach presented in [22]). These two graphs clearly demonstrate the usefulness of clustering in certain cases. That is, when the first graph is observed, one would tend to think that there is an increase in the number of samples that use the HTTP protocol. However, after the samples are clustered, one realizes that the use

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

Table 2.5: Overview of network activities.

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

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

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

Note that over 70% of the samples that downloaded an executable actually downloaded more than one. In fact, we observed one sample that downloaded the same file 178 times

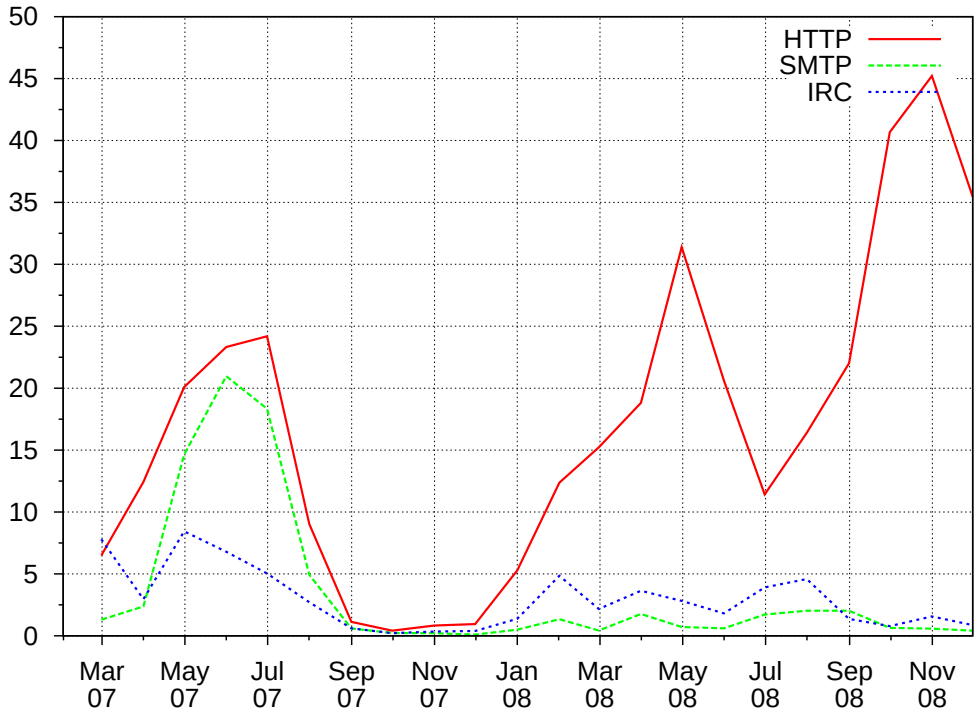

Figure 2.4: Network protocols (by samples).

during the analysis time of four minutes (i.e., the download was corrupted with each download, so the sample immediately attempted another download).

2.3.4 GUI windows

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

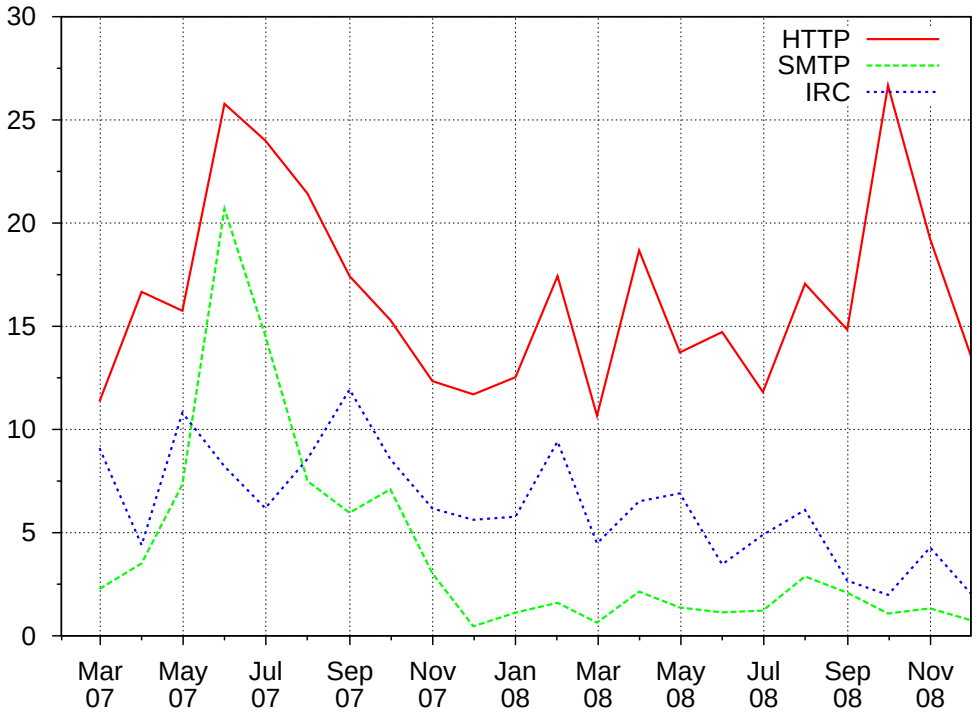

Figure 2.5: Network protocols (by families/clusters).

at all when being double clicked. For example, 1.7% of the samples show a fabricated message box that claims that a required DLL was not found. However, if this error message was authentic, it would be created on behalf of the `csrss.exe` process.

2.3.5 Botnet activity

Although a relative recent phenomenon, botnets have quickly become one of the most significant threats to the security of the Internet. Recent research efforts have led to mechanisms to detect and disrupt botnets [53]. To determine how prevalent bots are among our submissions, we analyzed the network traffic dumps that Anubis has recorded. For this, we were interested in detecting three bot families: IRC, HTTP, and P2P.

The first step in identifying a bot based on an analysis report is to determine the network protocol that is being used. Of course, the protocol detection needs to be done in a port-independent fashion, as a bot often communicates over a non-standard

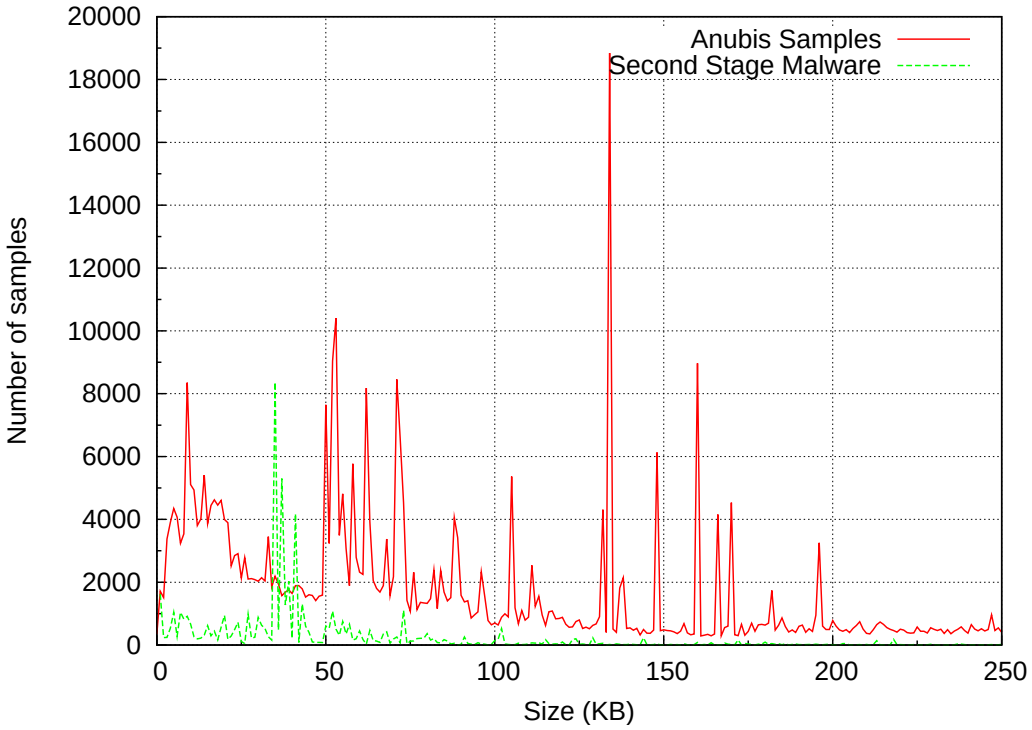

Figure 2.6: Sample sizes.

port. To this end, we implemented detectors for IRC, HTTP, and the following P2P protocols: BitTorrent, DirectConnect, EDonkey, EmuleExtension, FastTrack, Gnutella, and Overnet.

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

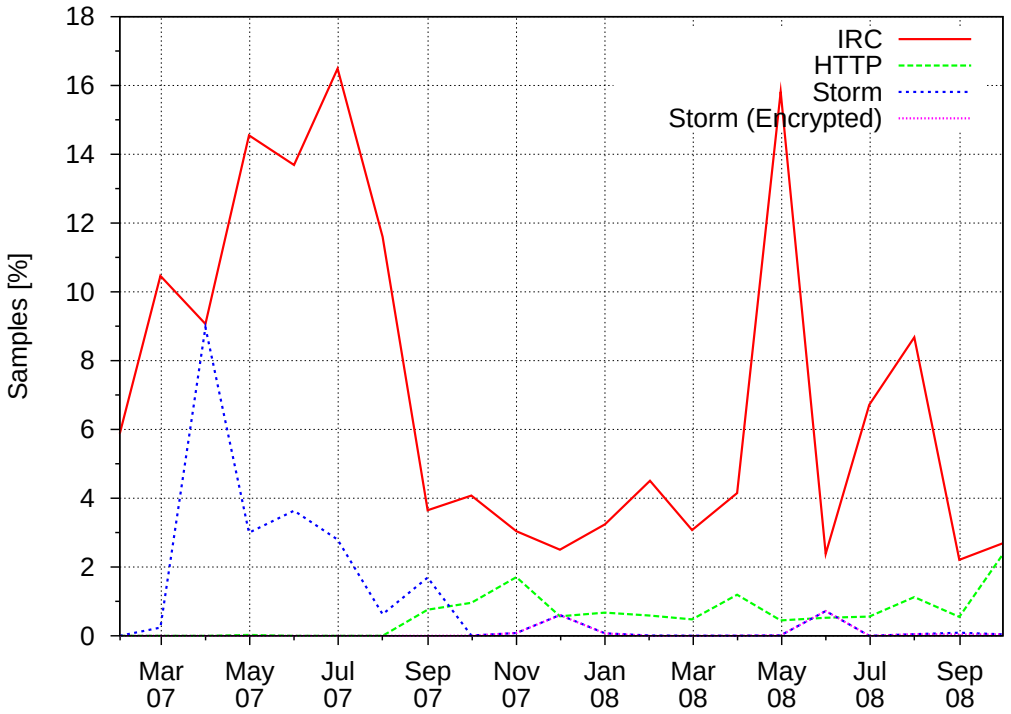

Figure 2.7: Botnet submissions (by samples).

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

Figure 2.7 and 2.8 show the bot submission (grouped by type) based on unique samples and unique clusters, respectively. By comparing the IRC botnet submissions in the two graphs, we can observe that, in 2007, most of IRC botnets were belonging to different clusters. In 2008 instead, we still received an high number of IRC bots, but they were mostly polymorphic variations of the same family. As an example, the peak that we observed in May 2008 is due to a large number of polymorphic variations of W32.Virut.

Interestingly, we are able to identify samples that, months after their first appearance, are still not recognized by any anti-virus software. This is probably due to the polymor-

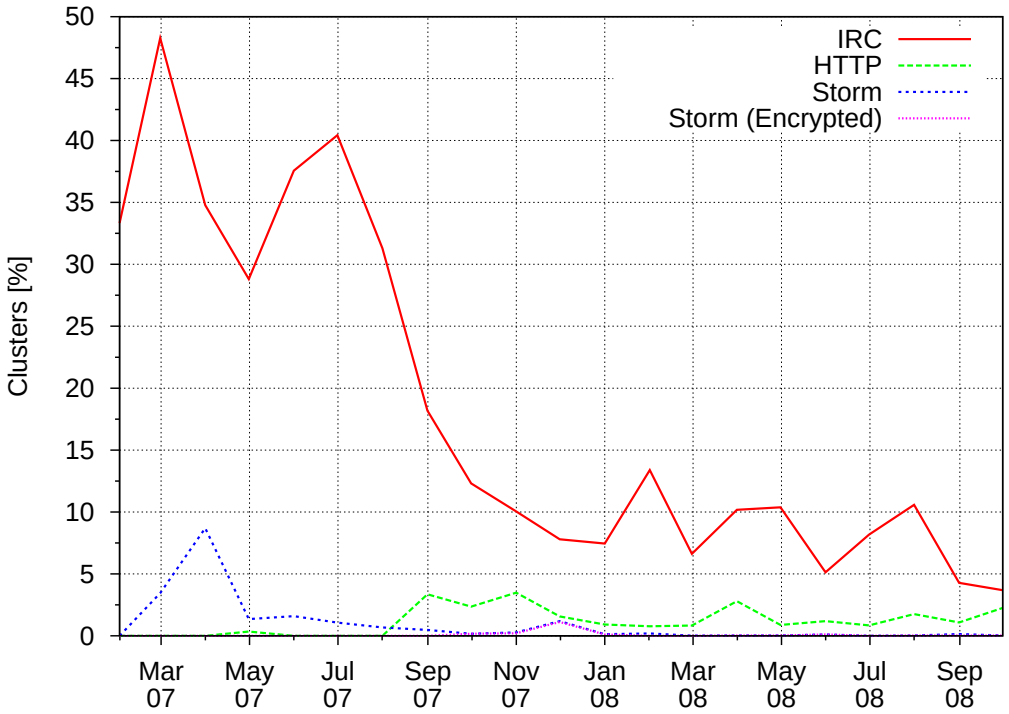

Figure 2.8: Botnet submissions (by families/clusters).

phism and metamorphism techniques used in the malware code. We also verified how many samples were identified by one anti-virus vendor as being a bot and cross-checked these samples with our detection technique. We missed 105 samples that the anti-virus software was able to detect. One reason for this could be the four-minute maximum runtime limit for the samples emulated in the Anubis system.

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

When IRC bots are analyzed in more detail, one observes that the channel topic is base64-encoded 13% of the time. During the time in which the samples were executed in Anubis, we also collected over 13,000 real commands that the bot master sent to malware under analysis. In 88% of the cases, the commands were instructing the client

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

Table 2.6: Overview of observed comparisons.

to download some file (e.g., `get` and `download` commands). Some other interesting commands that we observed were `ipscan`, `login`, `keylog`, `scan`, `msn`, and `visit`.

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

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

2.3.6 Sandbox detection

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

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

Unfortunately, we have currently no good way to detect *instruction-level* detection attempts. While it is theoretically possible to monitor single instructions in a full system emulator, for performance reasons, Anubis essentially works at the API (and system call) level. Since Anubis leverages Qemu for its full system emulation, it is susceptible to the same detection methods as Qemu is. In particular, several of the detection strategies (all of them are instruction-level detections) outlined in [41] and [80] also work for Anubis.

Since we do monitor Windows API calls, we are in a better position to check for possible *API-level detection* methods. At the time of writing this survey, several Anubis-specific detections have been published [15] on the Internet. All of them work by comparing the return value of a Windows API function such as `GetComputerName` to a a hard-coded value that is known to identify Anubis. Our system includes a data tainting component that allows us to detect when the output of an interesting function is subsequently used as an operand in a comparison operation [22]. Thus, we were able to analyze our sample set for specific comparisons that represent known Anubis checks published in forum postings or blogs. Table 2.6 summarizes our findings. A couple of checks deserve more discussion. Our system names each analyzed sample `sample.exe`. Thus, a well-known Anubis detection routine compares the return value of the Windows API function `GetCommandLine` with that string. Similarly, in the past, every sample was executed as a Windows user with the name "user." Consequently, it was possible to compare the result of `NtQuerySystemInformation` with the string "user" to detect Anubis (as a side note, we have since changed this username to "Administrator").

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

Under the assumption that a sample that detects Anubis (or a sandbox) does not perform much activity, we can also provide an upper bound for the samples that do sandbox detection. Based on our experience with Anubis, we consider a behavioral report (a profile [22]) to contain "not much activity" when it contains less than 150 features. For comparison, the average profile has 1,465 features. Using this definition,

we found that 12.45 % of the executable samples (13.57 % of the clusters) show not much activity.

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

2.4 Conclusion

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

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

3 Scalable, Behavior-Based Malware Clustering

In this chapter, we present evaluation results for the behavioral malware clustering techniques introduced in Deliverable D08 (D4.1): "Specification language for code behavior". Results from this chapter have been accepted for publication and presented at the Symposium on Network and Distributed System Security (NDSS) [20]. Furthermore, the results of applying these techniques to the Anubis [12] malware dataset have been made available to the public through a dedicated section of the Anubis web interface, that has been online since February 2009. Operation of this service has led us to perform further optimizations of the clustering implementation, to allow us to cluster ever-increasing numbers of malware samples. The final version takes advantage of parallelism and is able to cluster one million distinct malware samples in about six hours using 8 CPU cores..

To verify the effectiveness of our clustering approach, we used our system to cluster real-world malware data sets. In the next section, we discuss the quality of the generated clusters. Then, in Section 3.2, we compare our solution with previously-proposed clustering techniques [19, 64]. In Section 3.3, we present performance measurements of running our prototype on a very large data set. Finally, in Section 3.4, we discuss some examples of the clusters produced by our tool and of the insight they provide to the malware analyst.

3.1 Quality

Assessing the quality of the results that are produced by a clustering algorithm is an inherently difficult task. Obviously, it is possible to quantify the number of clusters, the average number of samples per cluster, or the relative sum of all pairwise distances for a cluster. Alternatively, one could randomly pick a few clusters and manually verify that the samples in these clusters are similar. The best option for demonstrating the correctness of a produced clustering, however, is to compare it with an existing reference clustering. Unfortunately, no such reference clustering exists for malware samples[1]. As

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

a result, to verify that our clustering approach is meaningful, we first needed to create a reference clustering.

Reference Clustering. To create a reference clustering, we took the following approach: First, we obtained a set of 14,212 malware samples that were submitted to ANUBIS [12] in the period from October 27, 2007 to January 31, 2008. These samples were contributed by a number of security organizations and individuals, spanning a wide range of sources (such as web infections, honeypots, botnet monitoring, peer-to-peer systems, and URLs extracted from other malware analysis services). Then, we scanned each sample with six different anti-virus programs. For the initial reference clustering, we selected only those samples for which the majority of the anti-virus programs reported the same malware family (this required us to define a mapping between the different labels that are used by different anti-virus products). This resulted in a total of 2,658 samples. For each sample, we examined the corresponding ANUBIS [12] report and manually corrected classification problems.

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

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

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

The overall precision value is:

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

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

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

The overall recall value is:

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

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

Clustering Results. We have run our clustering algorithm on the reference set of 2,658 samples. For this run, we selected a similarity threshold of $t = 0.7$. The value of this threshold was determined based on our experience with initial experiments on a small malware sample set with less than a hundred programs. Later in this section, we discuss in more detail the considerations for selecting an appropriate threshold. Moreover, we will show that the algorithm is quite robust with regard to the choice of the concrete threshold value.

Our system produced 87 clusters, while the reference clustering consists of 84 clusters. For our results, we derived a precision of 0.984 and a recall of 0.930. This demonstrates that our approach has produced a clustering that is very close to the reference set. The excellent precision shows that the system was able to differentiate well between different malware classes. The recall shows that, in almost all cases, samples of the same class were grouped in the same cluster. A quantitative comparison to other clustering techniques is presented in the following Section 3.2. In Section 3.4, we discuss a number of interesting, qualitative observations about the clustering that our system produced.

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

For our experiments, we decided to use a threshold value such that our results would differentiate between malware families (that is, only similar variants of the same family should be clustered). As mentioned previously, a concrete value of $t = 0.7$ was selected, based on our experience with initial, small-scale experiments. However, the selection of the correct value of t is quite robust. Figure 3.1 shows how precision and recall vary with respect to different choices of t. One can see that a broad range of choices for $t \in [0.6, 0.9]$ yield good results for both precision and recall.

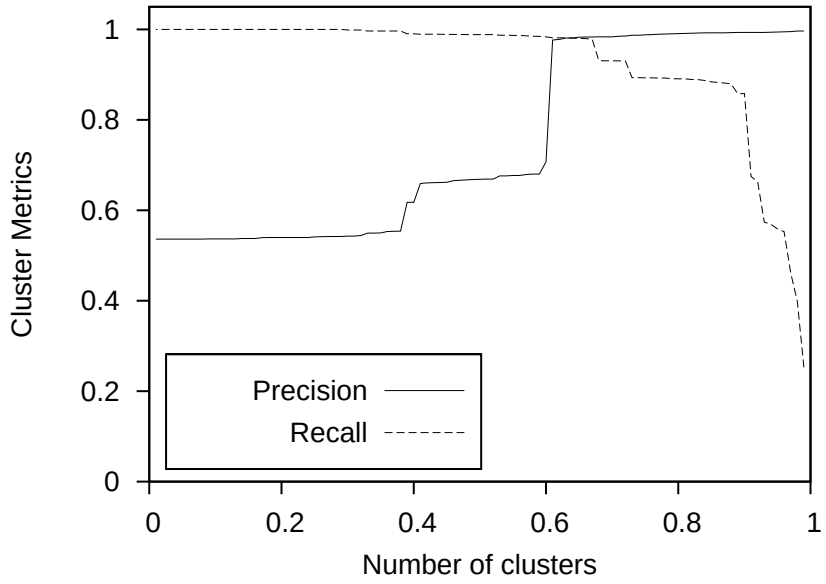

Figure 3.1: Precision and recall.

3.2 Comparative Evaluation

In the previous section, we have shown that our system has performed accurate clustering. However, we need to put these numbers into context with other approaches to be able to better assess the quality of our results. In this section, we present a comparative evaluation with the current state-of-the-art clustering approach, introduced by Bailey et al. [19]. Moreover, we analyze the impact of our behavioral abstraction and compare our clustering to one that is directly based on system call traces [64].

Bailey et al. [19] proposed a system for clustering malware based on the Normalized Compression Distance (NCD), using zlib-compression. NCD is based on the Kolmogorov complexity theory [66] and exploits the fact that similar data, when concatenated, compresses better than more differing data. Moreover, Bailey performs a coarse-grain abstraction from system calls and also uses profiles to represent malware behavior (we refer to these profiles as *Bailey-profiles* from now on). The difference to our approach is that Bailey-profiles contain only behavior in terms of non-transient state changes that a malware sample causes on the system (i.e., changes to the file-system, registry), as well as names of spawned processes and some basic information about network connections and scans. A detailed impression of the contents of Bailey-profiles can be gained from [17].

To evaluate Bailey's system on our reference data set, we adapted our dynamic analysis system to generate Bailey-profiles. Concerning NCD, we made use of the library provided by the Complearn-Toolkit [34].

A number of previous systems (e.g., [64]) based their behavioral profiles essentially on the raw system call traces. Thus, to evaluate the performance of such systems, and to obtain a baseline that shows the improvements due to generalized behavior profiles, we also performed clustering on the raw system call traces.

We used our reference clustering and the precision and recall metrics to directly compare the quality of the produced clusters for the different techniques. As an overall measure of clustering *quality*, we use the product of $precision * recall$. For each of the combinations of profile-types, similarity measures, and clustering methods presented in Table 3.1, we selected the threshold value which produces the highest quality score. In the clustering column, "exact" means that all $n*n/2$ distances between pairs of samples were computed, while "LSH" means that locality sensitive hashing was used. The last two rows show that the difference between exact and LSH-based clustering is minimal, demonstrating the effectiveness of LSH-based clustering as an approximation.

Behavioral Profile	*Similarity Measure*	*Clustering*	*Optimal Thresh*	*Quality*	*Precision*	*Recall*
Bailey-profile [19]	NCD	exact	0.75	**0.916**	0.979	0.935
Bailey-profile [19]	Jaccard Index	exact	0.63	**0.801**	0.971	0.825
Syscalls [64]	Jaccard Index	exact	0.19	**0.656**	0.874	0.750
Our profile	Jaccard Index	exact	0.61	**0.959**	0.977	0.981
Our profile	Jaccard Index	LSH	0.60	**0.959**	0.979	0.980

Table 3.1: Comparative evaluation of different clustering methods.

As can be seen in Table 3.1, the quality of our clustering approach (last two rows) outperforms the clustering proposed by Bailey et al. (first row). This is because our profiles represent the actual behavior of a malware sample in a more comprehensive and accurate way. For example, certain samples exhibit behavior that cannot be captured using Bailey-profiles. As a result, such profiles remain empty, or almost empty. Even more troublesome is the fact that Bailey's approach produces significantly worse results when using the Jaccard index as a similarity metric instead of NCD (second row). Unfortunately, a clustering algorithm based on NCD cannot take advantage of LSH to avoid computing all n^2 distances. Thus, a clustering approach that uses Bailey-profiles [19] either produces results that are significantly less precise than ours (by using the Jaccard

index and LSH), or it does not scale to real-world datasets (when using NCD). When analyzing the results for raw system call traces (third row), the results are significantly worse than for the other two techniques. This is not surprising, since the traces contain far too much noise to effectively find similarities between even closely-related malware instances.

3.3 Performance

To demonstrate the scalability of our clustering algorithm, we ran our system on a set of 75,692 malware samples (obtained from the complete database of ANUBIS). We performed our experiments on a XEN virtual machine that was hosted on a PowerEdge 2950 server equipped with two Quad-Core Xeon 1.86 GHz CPUs and 8 GB of RAM. We allocated about 7GB RAM and one physical CPU to the XEN VM.

As shown in Table 3.2, our prototype implementation succeeded to cluster the set of 75,692 samples in 2 hours and 18 minutes. This time could be further reduced by exploiting the inherent parallelism: Both the LSH hashing and the distance calculation step can be easily performed in parallel. The memory requirements of our prototype never exceeded 3.7 GB of virtual memory. For each sample, we store a behavioral profile on disk, which consumes about 96 KB of disk space on average. To load the samples, the clustering algorithm had to read and process $6.9GB$ of behavioral profiles.

We ran the clustering algorithm with the same threshold value $t = 0.7$. The LSH algorithm computed a set S, our approximation of the set of near pairs, that contained 66,528,049 pairs. Only 57,024,374 pairs were indeed above the similarity threshold t, i.e., LSH hashing resulted in about 14% false positives. Nevertheless, employing LSH hashing allowed us to calculate only 66,528,049 instead of $(75,692^2)/2 = 2,864,639,432$ distances. Thus, LSh reduced the number of necessary distance calculations by a factor of approximately 43.

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

Table 3.2: Runtime performance for 75K samples.

Compared to previous work, our prototype shows significantly improved performance. To classify malware based on NCD as in Bailey et al. [19], all of the $n^2/2$ distances between the n samples need to be computed. Moreover, it is possible to derive from the run-time graphs presented in their paper that a single distance calculation between two pairs takes about 1.25 milliseconds. As a result, the distance calculation step of their algorithm would require 995 hours (almost 6 weeks) to perform the necessary $75,692^2/2$ distance calculations. This is despite the fact that Bailey profiles are rather small (about 1KB on average). Applying our NCD implementation to the (much larger) behavioral profiles produced by our tool yields even more prohibitive computation times: a single NCD computation takes on average 43 milliseconds. Therefore, clustering 75,692 samples would take at least 6 months, even if the implementation were parallelized to run on 8 CPUs.

3.4 Qualitative Discussion of Clustering Results

In this section, we present a number of observations on the quality of our clustering techniques. First, we discuss the four largest clusters (with regard to the number of samples that they contain). These are Allaple.1 (1289 samples), Allaple.2 (717 samples), DOS (179 samples), and GBDialer.j (106 samples). Together, they account for 86% of all samples.

Allaple.1 and Allaple.2 are two different variants of the Allaple worm [14]. Allaple is a polymorphic malware, which explains why there are so many different samples in each cluster. It also demonstrates the ability of our system to quickly dispose of polymorphic malware instances that appear different but exhibit the same behavior. Interestingly, we found that virus scanners were inconsistently assigning different *variant names* to samples in both clusters (recall that we only used the malware *family names* that the virus scanners reported to perform the initial reference clustering). However, closer manual analysis showed that our clustering correctly identified two different Allaple variants. While all of the samples in both clusters perform ICMP scans, the Allaple.2 variant is much more aggressive at immediately attempting to exploit the target systems using a wider variety of propagation vectors. For instance, almost all Allaple.2 samples perform DNS lookups for the addresses of hosts they have successfully scanned, and attempt to connect to TCP port 9988, which corresponds to the Windows remote administration service. On the other hand, in none of the samples in the Allaple.1 cluster is there any DNS or port 9988 activity. Furthermore, all samples in Allaple.1 make a copy of themselves to the file "C:\WINDOWS\system32\urdvxc.exe," while none of the samples in Allaple.2 do. Moreover, in the Allaple.1 cluster, we observe the following, interesting

object dependences:

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

The first dependency indicates that the sample has succeeded in propagating itself over the network (to our nepenthes honeypot). Since our taint-system correctly handles memory-mapped files we see that the malware propagates by reading a memory-mapped file and writing it to the network. The second and third dependences provide a strong indication that this is polymorphic malware, since data from the malware sample and from a random number generation API is written to the new file "ccxebztz.exe." This shows how system call dependences can provide valuable insight on malware behavior.

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

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

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

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

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

According to the virus description database of AVG [13], the malware program checks whether the current date is greater then 2012/10/16, and if so, starts encrypting user documents. Our system was only partly able to find this date check. Our profile is shown below:

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

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

Of course, there are also malware programs for which our system did not produce the correct results. One common case is when a sample did not show any suspicious activity in our analysis environment. This could be because the malware program is damaged, or because it detects the presence of the analysis environment and exits prematurely. In

any case, it underlines the dependence of our system on the quality of the behavioral profiles. One cluster in particular is composed of 25 samples which belong to 10 different clusters according to the reference clustering. Manual analysis reveals that these samples all crash, which causes the Dr. Watson debugger application to be executed, generate a crash report and display a pop-up window asking the user permission to send the report to Microsoft. Clearly, this behavior is not specific to the malware family and it leads to misclassification.

3.5 Limitations and Future Work

Trace Dependence. A limitation of any dynamic approach to the analysis of malware is that it is trace-dependent. Analysis results will be based only on the sample's behavior during one (or more) specific execution runs. Unfortunately, some of a malware's behavior may be triggered only under specific conditions. A simple example of trigger-based behavior is a time-bomb. That is, a malware that only exhibits its malicious behavior on a specific date. Another examples is a bot, that only performs malicious actions when it receives specific commands through a command and control channel. Also, malware aimed at identity theft may only exhibit certain behavior when the user performs certain actions, such as logging in to specific electronic banking websites. Since we run malware samples automatically with no human interaction, such behavior will not occur in our traces.

Interestingly, our clustering may still succeed in grouping similar samples even when their most significant malicious behavior is not triggered, as is the case for the GBDialer.J cluster discussed in Section 3.4. The reason is that the behavioral features used for clustering encompass all malware behavior, not just malicious actions.

Evasion. Clearly, a malware author could manually modify a malware sample until its behavior is different enough from the original that the two are assigned to different clusters by our tool. We are not interested in this kind of labour-intensive, manual evasion. Instead, we consider an adversary who attempts to automatically produce an arbitrary number of mutations of a malware sample in such a way that all (or most) such mutations are assigned to different clusters by our tool. To this end, a malware author could randomly mutate parts of the malware's behavior that are not essential to its functionality. An example would be the often arbitrary file names under which the malware copies itself on the file system. These could be replaced with random strings, hardcoded into each malware instance. Nonetheless, adding enough randomness to make each mutation different is not a simple task. A sample in our dataset has more than one thousand features on average, many of which represent behavior from inside system

libraries that is only indirectly a consequence of the malware writer's intent. Also, since our tool discards features that are unique to a single malware instance, simple random variations would just lead to these features being discarded.

To make our tool more robust against this type of evasion, we could add more aggressive generalization to our algorithm for extracting behavioral profiles from execution traces. As an example we could consider the name of any file created by the malware irrelevant, and replace it with a special token as we do for the names of temporary files. Greater generalization of course loses some information, so it may be desirable to perform multiple clusterings using different levels of generalization.

Another issue is that dynamic data tainting of untrusted software is vulnerable to evasion. A malicious binary could easily inject fake data dependencies, using NOP-equivalent operations to taint clean data without modifying its value. Furthermore, it could hide data dependencies from our tool, using implicit flows to "clean" tainted data [31]. Unfortunately, there is no robust defense against such techniques. To address this issue we would therefore have to disable dynamic data tainiting, sacrificing some of the accuracy of our tool.

4 Effective and Efficient Malware Detection at the End Host

In Deliverable D08 (D4.1): "Specification language for code behavior", we already introduced the basics of our approach to detect malware based on information flow inside a malicious program. Furthermore, we presented implementation details of the prototype available at that time. Over the course of the project, we slightly modified the prototype. While the process of extracting behavior signatures (in the current system denoted as *behavior graphs*) through the means of binary slicing has not changed significantly, the design of the scanner/detection component has been improved in various ways.

In the following two sections, we briefly recapitulate the system overview and detail the new detection mechanism. In a third part, we present an evaluation we conducted using a set of six current malware families. Further, we show two sample behavior graphs extracted from malware used for the evaluation.

4.1 System Overview

The goal of our system is to effectively and efficiently detect malicious code at the end host. Moreover, the system should be general and not incorporate *a priori* knowledge about a particular malware class. Given the freedom that malware authors have when crafting malicious code, this is a challenging problem. To attack this problem, our system operates by generating detection models based on the observation of the execution of malware programs. That is, the system executes and monitors a malware program in a controlled analysis environment. Based on this observation, it extracts the behavior that characterizes the execution of this program. The behavior is then automatically translated into detection models that operate at the host level.

Our approach allows the system to quickly detect and eliminate novel malware variants. However, it is reactive in the sense that it must observe a certain, malicious behavior before it can properly respond. This introduces a small delay between the appearance of a new malware family and the availability of appropriate detection models. We believe that this is a trade-off that is necessary for a general system that aims to detect and mitigate malicious code with *a priori* unknown behavior. In some sense, the system can be compared to the human immune system, which also reacts to threats by

first detecting intruders and then building appropriate antibodies. Also, it is important to recognize that it is *not* required to observe every malware instance before it can be detected. Instead, the proposed system abstracts (to some extent) program behavior from a single, observed execution trace. This allows the detection of all malware instances that implement similar functionality.

Modeling program behavior. To model the behavior of a program and its security-relevant activity, we rely on system calls. Since system calls capture the interactions of a program with its environment, we assume that any relevant security violation is visible as one or more unintended interactions.

Of course, a significant amount of research has focused on modeling legitimate program behavior by specifying permissible sequences of system calls [44, 94]. Unfortunately, these techniques cannot be directly applied to our problem. The reason is that malware authors have a large degree of freedom in rearranging the code to achieve their goals. For example, it is very easy to reorder independent system calls or to add irrelevant calls. Thus, we cannot represent suspicious activity as system call sequences that we have observed. Instead, a more flexible representation is needed. This representation must capture true relationships between system calls but allow independent calls to appear in any order. For this, we represent program behavior as a *behavior graph* where nodes are (interesting) system calls. An edge is introduced from a node x to node y when the system call associated with y uses as argument some output that is produced by system call x. That is, an edge represents a data dependency between system calls x and y. Moreover, we only focus on a subset of interesting system calls that can be used to carry out malicious activity.

At a conceptual level, the idea of monitoring a piece of malware and extracting a model for it bears some resemblance to previous signature generation systems [76, 91]. In both cases, malicious activity is recorded, and this activity is then used to generate detection models. In the case of signature generation systems, network packets sent by worms are compared to traffic from benign applications. The goal is to extract tokens that are unique to worm flows and, thus, can be used for network-based detection. At a closer look, however, the differences between previous work and our approach are significant. While signature generation systems extract specific, byte-level descriptions of malicious traffic (similar to virus scanners), the proposed approach targets the semantics of program executions. This requires different means to observe and model program behavior. Moreover, our techniques to identify malicious activity and then perform detection differ as well.

Making detection efficient. In principle, we can directly use the behavior graph to detect malicious activity at the end host. For this, we monitor the system calls that an unknown program issues and match these calls with nodes in the graph. When enough of the graph has been matched, we conclude that the running program exhibits behavior that is similar to previously-observed, malicious activity. At this point, the running process can be terminated and its previous, persistent modifications to the system can be undone.

Unfortunately, there is a problem with the previously sketched approach. The reason is that, for matching system calls with nodes in the behavior graph, we need to have information about data dependencies between the arguments and return values of these systems calls. Recall that an edge from node x to y indicates that there is a data flow from system call x to y. As a result, when observing x and y, it is not possible to declare a match with the behavior graph $x \rightarrow y$. Instead, we need to know whether y uses values that x has produced. Otherwise, independent system calls might trigger matches in the behavior graph, leading to an unacceptable high number of false positives.

Previous systems have proposed dynamic data flow tracking (tainting) to determine dependencies between system calls. However, tainting incurs a significant performance overhead and requires a special environment (typically, a virtual machine with shadow memory). Hence, taint-based systems are usually only deployed in analysis environments but not at end hosts. In this paper, we propose an approach that allows us to detect previously-seen data dependencies by monitoring only system calls and their arguments. This allows efficient identification of data flows without requiring expensive tainting and special environments (virtual machines).

Our key idea to determine whether there is a data flow between a pair of system calls x and y that is similar to a previously-observed data flow is as follows: Using the observed data flow, we extract those parts of the program (the instructions) that are responsible for reading the input and transforming it into the corresponding output (a kind of program slice [96]). Based on this program slice, we derive a symbolic expression that represents the semantics of the slice. In other words, we extract an expression that can essentially pre-compute the expected output, based on some input. In the simplest case, when the input is copied to the output, the symbolic expression captures the fact that the input value is identical to the output value. Of course, more complicated expressions are possible. In cases where it is not possible to determine a closed symbolic expression, we can use the program slice itself (i.e., the sequence of program instructions that transforms an input value into its corresponding output, according to the functionality of the program).

Given a program slice or the corresponding symbolic expression, an unknown program can be monitored. Whenever this program invokes a system call x, we extract the

relevant arguments and return value. This value is then used as input to the slice or symbolic expression, computing the expected output. Later, whenever a system call y is invoked, we check its arguments. When the value of the system call argument is equal to the previously-computed, expected output, then the system has detected the data flow.

Using data flow information that is computed in the previously described fashion, we can increase the precision of matching observed system calls against the behavior graph. That is, we can make sure that a graph with a relationship $x \rightarrow y$ is matched only when we observe x and y, **and** there is a data flow between x and y that corresponds to the semantics of the malware program that is captured by this graph. As a result, we can perform more accurate detection and reduce the false positive rate.

4.2 System Details

In this section, we provide more details on the components of our system. In particular, we first discuss how we characterize program activity via behavior graphs. Then, we introduce our techniques to automatically extract such graphs from observing binaries. Finally, we present our approach to match the actions of an unknown binary to previously-generated behavior graphs.

4.2.1 Behavior Graphs: Specifying Program Activity

As a first step, we require a mechanism to describe the activity of programs. According to previous work [33], such a specification language for malicious behaviors has to satisfy three requirements: First, a specification must not constrain independent operations. The second requirement is that a specification must relate dependent operations. Third, the specification must only contain security-relevant operations.

The authors in [33] propose *malspecs* as a means to capture program behavior. A malicious specification (malspec) is a directed acyclic graph (DAG) with nodes labeled using system calls from an alphabet Σ and edges labeled using logic formulas in a logic $\mathcal{L}_{dep}$. Clearly, malspecs satisfy the first two requirements. That is, independent nodes (system calls) are not connected, while related operations are connected via a series of edges. The paper also mentions a function *IsTrivialComponent* that can identify and remove parts of the graph that are not security-relevant (to meet the third requirement).

For this work, we use a formalism called *behavior graphs*. Behavior graphs share similarities with malspecs. In particular, we also express program behavior as directed acyclic graphs where nodes represent system calls. However, we do not have unconstrained system call arguments, and the semantics of edges is somewhat different.

We define a system call $s \in \Sigma$ as a function that maps a set of input arguments $a_1, \ldots, a_n$ into a set of output values $o_1, \ldots, o_k$ For each input argument of a system call a_i, the behavior graph captures where the value of this argument is derived from. For this, we use a function $f_{a_i} \in F$. Before we discuss the nature of the functions in F in more detail, we first describe where a value for a system call can be derived from. A system call value can come from three possible sources (or a mix thereof): First, it can be derived from the output argument(s) of previous system calls. Second, it can be read from the process address space (typically, the initialized data section – the `bss` segment). Third, it can be produced by the immediate argument of a machine instruction.

As mentioned previously, a function is used to capture the input to a system call argument a_i. More precisely, the function f_{a_i} for an argument a_i is defined as $f_{a_i} : x_1, x_2, \ldots, x_n \rightarrow y$, where each x_i represents the output o_j of a previous system call. The values that are read from memory are part of the function body, represented by $l(addr)$. When the function is evaluated, l(addr) returns the value at memory location *addr*. This technique is needed to ensure that values that are loaded from memory (for example, keys) are not constant in the specification, but read from the process under analysis. Of course, our approach implies that the memory addresses of key data structures do not change between (polymorphic) variants of a certain malware family. In fact, this premise is confirmed by a recent observation that data structures are stable between different samples that belong to the same malware class [37]. Finally, constant values produced by instructions (through immediate operands) are implicitly encoded in the function body. Consider the case in which a system call argument a_i is the constant value 0, for example, produced by a `push $0` instruction. Here, the corresponding function is a constant function with no arguments $f_{a_i} :\rightarrow 0$. Note that a function $f \in F$ can be expressed in two different forms: As a (symbolic) formula or as an algorithm (more precisely, as a sequence of machine instructions – this representation is used in case the relation is too complex for a mathematical expression).

Whenever an input argument a_i for system call y depends on the some output o_j produced by system call x, we introduce an edge from the node that corresponds to x, to the node that corresponds to y. Thus, edges encode dependencies (i.e., temporal relationships) between system calls.

Given the previous discussion, we can define behavior graphs G more formally as: $G = (V, E, F, \delta)$, where:

- V is the set of vertices, each representing a system call $s \in \Sigma$
- E is the set of edges, $E \subseteq V \times V$
- F is the set of functions $\bigcup f : x_1, x_2, \ldots, x_n \rightarrow y$, where each x_i is an output arguments o_j of system call $s \in \Sigma$

- δ, which assigns a function f_i to each system call argument a_i

Intuitively, a behavior graph encodes relationships between system calls. That is, the functions f_i for the arguments a_i of a system call s determine how these arguments depend on the outputs of previous calls, as well as program constants and memory values. Note that these functions allow one to *pre-compute* the expected arguments of a system call. Consider a behavior graph G where an input argument a of a system call s_t depends on the outputs of two previous calls s_p and s_q. Thus, there is a function f_a associated with a that has two inputs. Once we observe s_p and s_q, we can use the outputs o_p and o_q of these system calls and plug them into f_a. At this point, we know the expected value of a, assuming that the program execution follows the semantics encoded in the behavior graph. Thus, when we observe at a later point the invocation of s_t, we can check whether its actual argument value for a matches our precomputed value $f_a(o_p, o_q)$. If this is the case, we have high confidence that the program executes a system call whose input is related (depends on) the outputs of previous calls. This is the key idea of our proposed approach: We can identify relationships between system calls without tracking any information at the instruction-level during runtime. Instead, we rely solely on the analysis of system call arguments and the functions in the behavior graph that capture the semantics of the program.

4.2.2 Extracting Behavior Graphs

As mentioned in the previous section, we express program activity as behavior graphs. These behavior graphs can be automatically constructed by observing the execution of a program in a controlled environment. For more detailed information on the extraction process, refer to Deliverable D08 (D4.1): "Specification language for code behavior".

Optimizing Functions

We would like to point out one improvement of the extraction process when compared to the previous deliverable in this section, however: Once we have extracted a slice for a system call argument and translated it into a corresponding function (program), we could stop there. However, many functions implement a very simple behavior; they copy a value that is produced as output of a system call into the input argument of a subsequent call. For example, when a system call such as `NtOpenFile` produces an opaque handle, this handle is used as input by all subsequent system calls that operate on this file. Unfortunately, the chain of copy operations can grow quite long, involving memory accesses and stack manipulation. Thus, it would be beneficial to identify and simplify instruction sequences. Optimally, the complete sequence can be translated into

a formula that allows us to directly compute the expected output based on the formula's inputs.

To simplify functions, we make use of symbolic execution. More precisely, we assign symbolic values to the input parameters of a function and use a symbolic execution engine developed previously [62]. Once the symbolic execution of the function has finished, we obtain a symbolic expression for the output. When the symbolic execution engine does not need to perform any approximations (e.g., widening in the case of loops), then we can replace the algorithmic representation of the slice with this symbolic expression. This allows us to significantly shorten the time it takes to evaluate functions, especially those that only move values around. For complex functions, we fall back to the explicit machine code representation.

4.2.3 Matching Behavior Graphs

For every malware program that we analyze in our controlled environment, we automatically generate a behavior graph. These graphs can then be used for detection at the end host. More precisely, for detection, we have developed a scanner that monitors the system call invocations (and arguments) of a program under analysis. The goal of the scanner is to efficiently determine whether this program exhibits activity that matches one of the behavior graphs. If such a match occurs, the program is considered malicious, and the process is terminated. We could also imagine a system that unrolls the persistent modifications that the program has performed. For this, we could leverage previous work [93] on safe execution environments.

In the following, we discuss how our scanner matches a stream of system call invocations (received from the program under analysis) against a behavior graph. The scanner is a user-mode process that runs with administrative privileges. It is supported by a small kernel-mode driver that captures system calls and arguments of processes that should be monitored. In the current design, we assume that the malware process is running under the normal account of a user, and thus, cannot subvert the kernel driver or attack the scanner. We believe that this assumption is reasonable because, for recent versions of Windows, Microsoft has made significant effort to have users run without root privileges. Also, processes that run executables downloaded from the Internet can be automatically started in a low-integrity mode. Interestingly, we have seen malware increasingly adapting to this new landscape, and a substantial fraction can now successfully execute as a normal user.

The basic approach of our matching algorithm is the following: First, we partition the nodes of a behavior graph into a set of *active* nodes and a set of *inactive* nodes. The set of active nodes contains those nodes that have already been matched with system

call(s) in the stream. Initially, all nodes are inactive.

When a new system call s arrives, the scanner visits all inactive nodes in the behavior graph that have the correct type. That is, when a system call `NtOpenFile` is seen, we examine all inactive nodes that correspond to an `NtOpenFile` call. For each of these nodes, we check whether all its parent nodes are active. A parent node for node N is a node that has an edge to N. When we find such a node, we further have to ensure that the system call has the "right" arguments. More precisely, we have to check all functions $f_i : 1 \leq i \leq k$ associated with the k input arguments of the system call s. However, for performance reasons, we do not do this immediately. Instead, we only check the *simple functions*. Simple functions are those for which a symbolic expression exists. Most often, these functions check for the equality of handles. The checks for *complex functions*, which are functions that represent dependencies as programs, are deferred and optimistically assumed to hold.

To check whether a (simple) function f_i holds, we use the output arguments of the parent node(s) of N. More precisely, we use the appropriate values associated with the parent node(s) of N as the input to f_i. When the result of f_i matches the input argument to system call s, then we have a match. When all arguments associated with simple functions match, then node N can be activated. Moreover, once s returns, the values of its output parameters are stored with node N. This is necessary because the output of s might be needed later as input for a function that checks the arguments of N's child nodes.

So far, we have only checked dependencies between system calls that are captured by simple functions. As a result, we might activate a node y as the child of x, although there exists a complex dependency between these two system calls that is *not* satisfied by the actual program execution. Of course, at one point, we have to check these complex relationships (functions) as well. This point is reached when an *interesting* node in the behavior graph is activated. Interesting nodes are nodes that are (a) associated with security-relevant system calls and (b) at the "bottom" of the behavior graph. With security-relevant system calls, we refer to all calls that write to the file system, the registry, or the network. In addition, system calls that start new processes or system services are also security-relevant. A node is at the "bottom" of the behavior graph when it has no outgoing edges.

When an interesting node is activated, we go back in the behavior graph and check all complex dependencies. That is, for each active node, we check all complex functions that are associated with its arguments (in a way that is similar to the case for simple functions, as outlined previously). When all complex functions hold, the node is marked as *confirmed*. If any of the complex functions associated with the input arguments of an active node N does not hold, our previous optimistic assumption has been invalidated.

Thus, we deactivate N as well as all nodes in the subgraph rooted in N.

Intuitively, we use the concept of interesting nodes to capture the case in which a malware program has demonstrated a chain of activities that involve a series of system calls with non-trivial dependencies between them. Thus, we declare a match as soon as any interesting node has been confirmed. However, to avoid cases of overly generic behavior graphs, we only report a program as malware when the process of confirming an interesting node involves at least one complex dependency.

Since the confirmed activation of a single interesting node is enough to detect a malware sample, typically only a subset of the behavior graph of a malware sample is employed for detection. More precisely, each interesting node, together with all of its ancestor nodes and the dependencies between these nodes, can be used for detection independently. Each of these subgraphs is itself a behavior graph that describes a specific set of actions performed by a malware program (that is, a certain behavioral trait of this malware).

4.3 Evaluation

We claim that our system delivers effective detection with an acceptable performance overhead. In this section, we first analyze the detection capabilities of our system. Then, we examine the runtime impact of our prototype implementation. In the last section, we describe two examples of behavior graphs in more detail.

Name	**Type**
Allaple	Exploit-based worm
Bagle	Mass-mailing worm
Mytob	Mass-mailing worm
Agent	Trojan
Netsky	Mass-mailing worm
Mydoom	Mass-mailing worm

Table 4.1: Malware families used for evaluation.

4.3.1 Detection Effectiveness

To demonstrate that our system is effective in detecting malicious code, we first generated behavior graphs for six popular malware families. An overview of these families

Name	Samples	Kaspersky variants	Our variants	Samples detected	Effectiveness
Allaple	50	2	1	50	1.00
Bagle	50	20	14	46	0.92
Mytob	50	32	12	47	0.94
Agent	50	20	2	41	0.82
Netsky	50	22	12	46	0.92
Mydoom	50	6	3	49	0.98
Total	300	102	44	279	0.93

Table 4.2: Training dataset.

is provided in Table 4.1. These malware families were selected because they are very popular, both in our own malware data collection (which we obtained from Anubis [1]) and according to lists compiled by anti-virus vendors. Moreover, these families provide a good cross section of popular malware classes, such as mail-based worms, exploit-based worms, and a Trojan horse. Some of the families use code polymorphism to make it harder for signature-based scanners to detect them. For each malware family, we randomly selected 100 samples from our database. The selection was based on the labels produced by the Kaspersky anti-virus scanner and included different variants for each family. During the selection process, we discarded samples that, in our test environment, did not exhibit any interesting behavior. Specifically, we discarded samples that did not modify the file system, spawn new processes, or perform network communication. For the `Netsky` family, only 63 different samples were available in our dataset.

Detection capabilities. For each of our six malware families, we randomly selected 50 samples. These samples were then used for the extraction of behavior graphs. Table 4.2 provides some details on the training dataset. The "Kaspersky variants" column shows the number of different variants (labels) identified by the Kaspersky anti-virus scanner (these are variants such as `Netsky.k` or `Netsky.aa`). The "Our variants" column shows the number of different samples from which (different) behavior graphs had to be extracted before the training dataset was covered. Interestingly, as shown by the "Samples detected" column, it was not possible to extract behavior graphs for the entire training set. The reasons for this are twofold: First, some samples did not perform any interesting activity during behavior graph extraction (despite the fact that they did show relevant behavior during the initial selection process). Second, for some malware programs, our

Name	Samples	Known variant samples	Samples detected	Effectiveness
Allaple	50	50	45	0.90
Bagle	50	26	30	0.60
Mytob	50	26	36	0.72
Agent	50	4	5	0.10
Netsky	13	5	7	0.54
Mydoom	50	44	45	0.90
Total	263	155	168	0.64

Table 4.3: Detection effectiveness.

system was not able to extract valid behavior graphs. This is due to limitations of the current prototype that produced invalid slices (i.e., functions that simply crashed when executed).

To evaluate the detection effectiveness of our system, we used the behavior graphs extracted from the training dataset to perform detection on the remaining 263 samples (the test dataset). The results are shown in Table 4.3. It can be seen that some malware families, such as `Allaple` and `Mydoom`, can be detected very accurately. For others, the results appear worse. However, we have to consider that different malware variants may exhibit different behavior, so it may be unrealistic to expect that a behavior graph for one variant always matches samples belonging to another variant. This is further exacerbated by the fact that anti-virus software is not particularly good at classifying malware (a problem that has also been discussed in previous work [18]). As a result, the dataset likely contains mislabeled programs that belong to different malware families altogether. This was confirmed by manual inspection, which revealed that certain malware families (in particular, the `Agent` family) contain a large number of variants with widely varying behavior.

To confirm that different malware variants are indeed the root cause of the lower detection effectiveness, we then restricted our analysis to the 155 samples in the test dataset that belong to "known" variants. That is, we only considered those samples that belong to malware variants that are also present in the training dataset (according to Kaspersky labels). For this dataset, we obtain a detection effectiveness of 0.92. This is very similar to the result of 0.93 obtained on the training dataset. Conversely, if we restrict our analysis to the 108 samples that do *not* belong to a known variant, we obtain a detection effectiveness of only 0.23. While this value is significantly lower, it still

demonstrates that our system is sometimes capable of detecting malware belonging to previously unknown variants. Together with the number of variants shown in Table 4.2, this indicates that our tool produces a behavior-based malware classification that is more general than that produced by an anti-virus scanner, and therefore, requires a smaller number of behavior graphs than signatures.

False positives. In the next step, we attempted to evaluate the amount of false positives that our system would produce. For this, we installed a number of popular applications on our test machine, which runs Microsoft Windows XP and our scanner. More precisely, we used Internet Explorer, Firefox, Thunderbird, putty, and Notepad. For each of these applications, we went through a series of common use cases. For example, we surfed the web with IE and Firefox, sent a mail with Thunderbird (*including* an attachment), performed a remote ssh login with putty, and used notepad for writing and saving text. No false positives were raised in these tests. This was expected, since our models typically capture quite tightly the behavior of the individual malware families. However, if we omitted the checks for *complex functions* and assumed all complex dependencies in the behavior graph to hold, *all* of the above applications raised false positives. This shows that our tool's ability to capture arbitrary data-flow dependencies and verify them at runtime is essential for effective detection. It also indicates that, in general, system call information alone (without considering complex relationships between their arguments) might not be sufficient to distinguish between legitimate and malicious behavior.

In addition to the Windows applications mentioned previously, we also installed a number of tools for performance measurement, as discussed in the following section. While running the performance tests, we also did not experience any false positives.

4.3.2 System Efficiency

As every malware scanner, our detection mechanism stands and falls with the performance degradation it causes on a running system. To evaluate the performance impact of our detection mechanism, we used 7-zip, a well-known compression utility, Microsoft Internet Explorer, and Microsoft Visual Studio. We performed the tests on a single-core, 1.8 GHz Pentium 4 running Windows XP with 1 GB of RAM.

For the first test, we used a command line option for 7-zip that makes it run a simple benchmark. This reflects the case in which an application is mostly performing CPU-bound computation. In another test, 7-zip was used to compress a folder that contains 215 MB of data (6,859 files in 808 subfolders). This test represents a more mixed workload. The third test consisted of using 7-zip to archive three copies of this same folder, performing no compression. This is a purely IO-bound workload. The next test

measures the number of pages per second that could be rendered in Internet Explorer. For this test, we used a local copy of a large (1.5MB) web page [35]. For the final test, we measured the time required to compile and build our scanner tool using Microsoft Visual Studio. The source code of this tool consists of 67 files and over 17,000 lines of code. For all tests, we first ran the benchmark on the unmodified operating system (to obtain a baseline). Then, we enabled the kernel driver that logs system call parameters, but did not enable any user-mode detection processing of this output. Finally, we also enabled our malware detector with the full set of 44 behavior graphs.

Test	**Baseline**	**Driver**		**Scanner**	
		Score	**Overhead**	**Score**	**Overhead**
7-zip (benchmark)	114 sec	117 sec	2.3%	118 sec	2.4%
7-zip (compress)	318 sec	328 sec	3.1%	333 sec	4.7%
7-zip (archive)	213 sec	225 sec	6.2%	231 sec	8.4%
IE - Rendering	0.41 page/s	0.39 pages/s	4.4%	0.39 page/s	4.4%
Compile	104 sec	117 sec	12.2%	146 sec	39.8%

Table 4.4: Performance evaluation.

The results are summarized in Table 4.4. As can be seen, our tool has a very low overhead (below 5%) for CPU-bound benchmarks. Also, it performs well in the I/O-bound experiment (with less than 10% overhead). The worst performance occurs in the compilation benchmark, where the system incurs an overhead of 39.8%. It may seem surprising at first that our tool performs worse in this benchmark than in the IO-bound archive benchmark. However, during compilation, the scanned application is performing almost 5,000 system calls per second, while in the archive benchmark, this value is around 700. Since the amount of computation performed in user-mode by our scanner increases with the number of system calls, compilation is a worst-case scenario for our tool. Furthermore, the more varied workload in the compile benchmark causes more complex functions to be evaluated. The 39.8% overhead of the compile benchmark can further be broken down into 12.2% for the kernel driver, 16.7% for the evaluation of *complex functions*, and 10.9% for the remaining user-mode processing. Note that the high cost of complex function evaluation could be reduced by improving our symbolic execution engine, so that less complex functions need to be evaluated. Furthermore, our prototype implementation spawns a new process every time that the verification of complex dependencies is triggered, causing unnecessary overhead. Nevertheless, we feel that our prototype performs well for common tasks, and the current overhead allows the

system to be used on (most) end user's hosts. Moreover, even in the worst case, the tool incurs significantly less overhead than systems that perform dynamic taint propagation (where the overhead is typically several times the baseline).

4.3.3 Examples of Behavior Graphs

To provide a better understanding of the type of behavior that is modeled by our system, we provide a short description of two behavior graphs extracted from variants of the Agent and Allaple malware families.

Agent.ffn.StartService. The `Agent.ffn` variant contains a resource section that stores chunks of binary data. During execution, the binary queries for one of these stored resources and processes its content with a simple, custom decryption routine. This routine uses a variant of XOR decryption with a key that changes as the decryption proceeds. In a later step, the decrypted data is used to overwrite the Windows system file `C:\WINDOWS\System32\drivers\ip6fw.sys`. Interestingly, rather than directly writing to the file, the malware opens the `\\.\C:` logical partition at the offset where the `ip6fw.sys` file is stored, and directly writes to that location. Finally, the malware restarts Windows XP's integrated IPv6 firewall service, effectively executing the previously decrypted code.

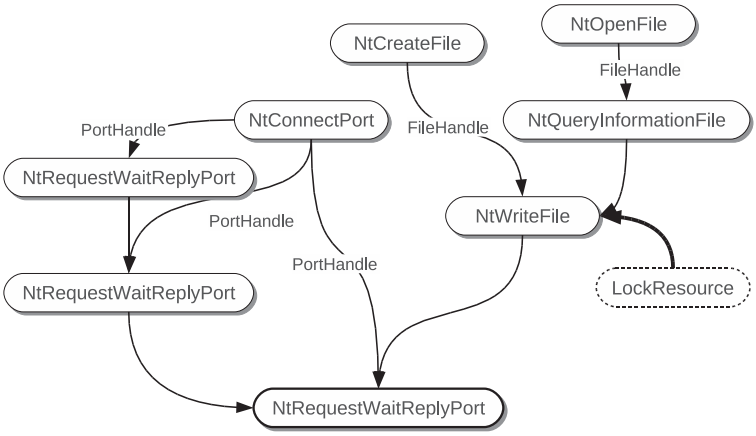

Figure 4.1: Behavior graph for `Agent.fnn`.

Figure 4.1 shows a simplified behavior graph that captures this behavior. The graph contains nine nodes, connected through ten dependencies: six simple dependencies rep-

resenting the reuse of previously obtained object handles (annotated with the parameter name), and four complex dependencies. The complex dependency that captures the previously described decryption routine is indicated by a bold arrow in Figure 4.1. Here, the `LockResource` function provides the body of the encrypted resource section. The `NtQueryInformationFile` call provides information about the `ip6fw.sys` file. The `\\.\C:` logical partition is opened in the `NtCreateFile` node. Finally, the `NtWriteFile` system call overwrites the firewall service program with malicious code. The check of the complex dependency is triggered by the activation of the last node (bold in the figure).

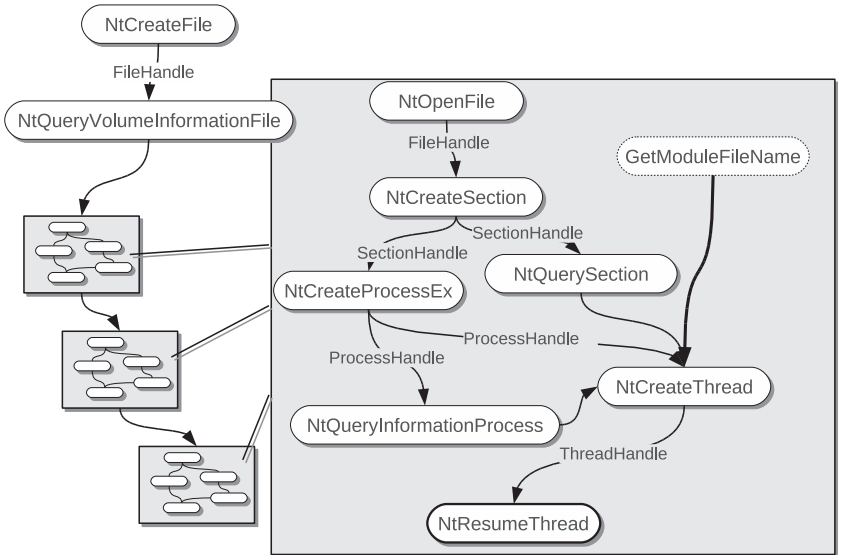

Figure 4.2: Behavior graph for `Allaple.b`.

Allaple.b.CreateProcess. Once started, the `Allaple.b` variant copies itself to the file `c:\WINDOWS\system32\urdvxc.exe`. Then, it invokes this executable various times with different command-line arguments. First, `urdvxc.exe /installservice` and `urdvxc.exe /start` are used to execute stealthily as a system service. In a second step, the malware tries to remove its traces by eliminating the original binary. This is done by calling `urdvxc.exe /uninstallservice patch:<binary>` (where `<binary>` is the name of the originally started program).

The graph shown in Figure 4.2 models part of this behavior. In the `NtCreateFile` node, the `urdvxc.exe` file is created. This file is then invoked three times with different arguments, resulting in three almost identical subgraphs. The box on the right-hand side

of Figure 4.2 is an enlargement of one of these subgraphs. Here, the `NtCreateProcessEx` node represents the invocation of the `urdvxc.exe` program. The argument to the uninstall command (i.e., the name of the original binary) is supplied by the `GetModuleFileName` function to the `NtCreateThread` call. The last `NtResumeThread` system call triggers the verification of the complex dependencies.

5 System Call Analysis

5.1 Motivation and introduction

Most of the anomalous actions that an aggressor would try to perform on a system through uploaded malware (e.g., accessing the host file system, sending or receiving packets over the network, executing other programs on the host, etc.) require the use of one or more system calls. Thus, it is reasonable to monitor such calls in order to analyze the behavior of a process. In particular, we propose to use anomaly detection techniques to flag anomalous or suspicious executions and record them for review in order to create a trail (i.e., the alert logs) that would otherwise be lost. We use S^2A^2DE, a tool which we developed in [67, 99], which makes use of both the sequence and the content of system calls to detect anomalies. This has been shown to be more efficient than using sequences of syscalls only, something which has been studied for a long time since the seminal work [45].

S^2A^2DE is a next-generation evolution of the seminal works in the field by Vigna et al. [63, 73], and uses a Markovian model of the sequence (as in, e.g., [61]) complemented with an analysis of the arguments of the system calls to detect intrusions - and malware activity.

One of the reasons why this type of detection is deemed useful in the context of WOMBAT is that, nowadays, skilled attackers are wary of writing anything on the hard drive of an attacked machine. Thus, if we wish to preserve malware samples, we need to take into account in-memory execution, which is a widely known and used "definitive anti-forensic" [47, 24, 55] technique.

There are two wide classes of anti-forensics techniques: *transient* techniques make the acquired evidence difficult to analyze with a specific tool or procedure, but not impossible to analyze in general. *Definitive* anti-forensics techniques instead effectively deny once and forever any access to the evidence. In this case, the evidence may be destroyed by the attacker, or may simply not exist on the media. The final objective of anti-forensics is to reduce the quantity and spoil the quality [51] of the evidence that can be retrieved.

Examples of transient anti-forensics techniques are the fuzzing and abuse of filesystems in order to create malfunctions or to exploit vulnerabilities of the tools used by the analyst, or the use of log analysis tools vulnerabilities to hide or modify certain information

[46, 51]. In other cases, entire filesystems have been hidden inside the metadata of other filesystems [51], but techniques have been developed to cope with such attempts [83]. Other examples are the use of steganography [60], or the modification of file metadata in order to make filetype not discoverable. In these cases the evidence is not completely unrecoverable, but it may escape any quick or superficial examination of the media: a common problem today, where investigators are overwhelmed with cases and usually undertrained, and therefore overly reliant on tools.

Definitive anti-forensics, on the other hand, effectively denies access to the evidence. The attackers may encrypt it, or securely delete it from filesystems (this process is sometimes called "counter-forensics") with varying degrees of success [49, 48]. Access times may be rearranged to alter the activity timeline that is usually exploited by analysts to correlate events. The final anti-forensics methodology is not to leave a trail: for instance, modern attack tools (commercial or open source) such as Metasploit [4], Mosdef or Core IMPACT [36] focus on pivoting and in-memory injection of code: in this case, nothing or almost nothing is written on disk, and therefore information on the attack will be lost as soon as the system is powered down, which is usually standard operating procedure on compromised machines. These techniques are also known as "disk-avoiding" procedures.

Memory dump and analysis operations have been advocated in response to this, and tools are being built to cope with the complex tasks of reliable acquisition [27, 89] and analysis [27, 88, 78] of a modern system's memory. However, even if the memory can be acquired and examined, if the injected process has already terminated, no trace of the attack will be found: these techniques are much more useful against in-memory resident backdoors and rootkits, which by definition are persistent.

In [68] we used S^2A^2DE to detect malicious code in-memory, generating a forensic audit trail even if an attack does not write code to disk.

5.2 Architecture and implementation of S^2A^2DE

We have briefly described the architecture of S^2A^2DE in Deliverable D08, but we recapitulate here for a better understanding of its usage in the following.

The architecture of S^2A^2DE is shown in Figure 5.1. Each execution of an application is modeled as a sequence of system calls, $S = [s_1, s_2, s_3, \ldots]$, logged by the operating system auditing facilities. Each system call s_i is characterized by a *type* (e.g. `read`, `write`, `exec`, etc.) and a list of *arguments* (e.g., the path of the file to be opened by `open`). We ignore instead the return value of the system call.

Please note that, while for simplicity in our study this system was developed and tested on Linux and FreeBSD, nothing in this setup is unusual, so the concept may be

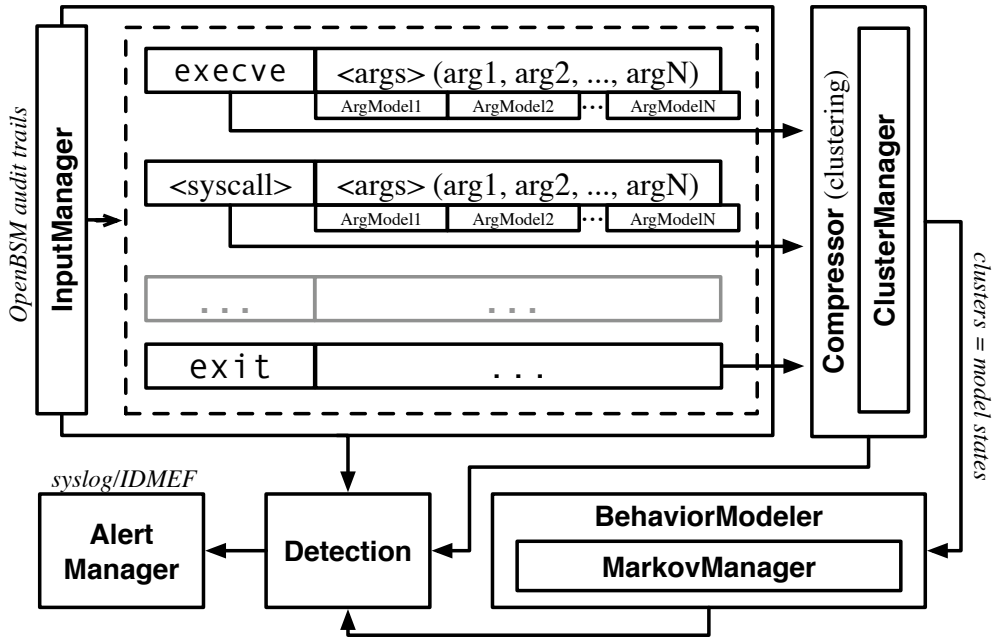

Figure 5.1: The architecture of our HIDS prototype

easily ported to any operating system

S^2A^2DE must be trained in order to "learn" a model of the normal behavior of the monitored applications. During this phase, the system builds a distinct profile for each application (e.g. `sendmail`, `telnetd`, etc.). A two-phase process of machine learning is then applied to each type of system call separately. Firstly, a single-linkage, bottom-up agglomerative hierarchical clustering algorithm [54] is used to find, for each type of system call, sub-clusters of invocations with similar arguments. We are interested in creating models on these clusters, and not on the general system call, in order to better capture normality and deviations on a more compact input space. This is important because some system calls, most notably `open`, are used in very different ways. Indeed, `open` is probably the most used system call on UNIX-like systems, since it opens files or devices in the file system creating a descriptor for further use. Only by careful aggregation over its parameters (i.e., the file path, a set of flags indicating the type of operation, and an opening mode) we can de-multiplex the general system call into "sub-groups" that are specific to a single function. In order to do this, we must define a way to measure "distance" among arguments, as we will show.

Afterwards, the system builds models of the parameters inside each cluster. The type of models, as well as the type of distances used for agglomeration, depend on the type of parameter, as shown in Table 5.1. In our framework, the distance among two system calls, s_i and s_j, is the sum of distances between corresponding arguments $D(s_i, s_j) = \sum_{a \in \mathrm{A}_s} d_{\mathrm{model}(a)}(s_i^a, s_j^a)$ (being A_s the shared set of system call arguments). For each couple of corresponding arguments a we compute the distance as:

$$d_a = \begin{cases} K_{(\cdot)} + \alpha_{(\cdot)}\delta_{(\cdot)} & \text{if the elements are different} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

where $K_{(\cdot)}$ is a fixed quantity which creates a "step" between different elements, while the second term is the real distance between the arguments $\delta_{(\cdot)}$, normalized by a parameter $\alpha_{(\cdot)}$. We use "$(\cdot)$" to denote that such variables are parametric w.r.t. the type of argument.

Since hierarchical clustering does not offer a concept analogous to the "centroid" of partitioning algorithms that can be used for classifying new inputs, we also created, for each cluster, a stochastic model that can be used to classify further inputs. These models generate a *probability density function* that can be used to state the probability with which the input belongs to the model. It is not strictly necessary for such model, or its distance or probability functions, to be the same as the distance functions that are used for clustering purposes.

As can be seen in Table 5.1, at least 4 different types of arguments are passed to system calls: path names and file names, discrete numeric values, arguments passed to

Table 5.1: Association of models to Syscall arguments in our prototype

Syscall	Model used for the arguments
`open`	`pathname` → Path Name `flags, mode` → Discrete Numeric
`execve`	`filename` → Path Name `argv` → Execution Argument
`setuid, setgid`	`uid, gid` → User/Group
`setreuid, setregid`	`ruid, euid` → User/Group
`setresuid, setresgid`	`ruid, euid, suid` → User/Group
`symlink, link,rename`	`oldpath,newpath` → Path Name
`mount`	`source, target` → Path Name `flags` → Discrete Numeric
`umount`	`target,flags` → Path Name
`exit`	`status` → Discrete Numeric
`chown` `lchown`	`path` → Path Name `group, owner` → User/Group
`chmod, mkdir` `creat`	`path` → Path Name `mode` → Discrete Numeric
`mknode`	`pathname` → Path Name `mode, dev` → Discrete Numeric
`unlink, rmdir`	`pathname` → Path Name

programs for execution, users and group identifiers (`UID`s and `GID`s).

Path names and file names are very frequently used in system calls. They are complex structures, rich of useful information, and therefore difficult to model properly. For the clustering phase, we chose to use a very simple model, the directory tree depth. This is easy to compute, and experimentally leads to fairly good results. Thus, in Equation 5.1 we set δ_a to be the difference in depth. The stochastic model for path names is a probabilistic tree which contains all the directories involved with a probability weight for each. Filenames are often too variable to be considered, so if the leaves of the tree are too different we simply ignore them for that specific model.

Discrete numeric values such as flags, opening modes, etc. are usually chosen from a limited set. Therefore we can store all of them along with a discrete probability. Since in this case two values can only be "equal" or "different", we set up a binary distance model for clustering, where the distance between x and y is:

$$d_a = \begin{cases} K_{disc} & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases}$$

and K_{disc}, as usual, is a configuration parameter. In this case, the generation of the probability density function is straightforward.

We also noticed that *execution arguments* (i.e. the arguments passed to the `execve` syscall) are difficult to model, but we found the length to be an extremely effective indicator of similarity of use. Therefore we set up a binary distance model, where the distance between x and y is:

$$d_a = \begin{cases} K_{arg} & \text{if } |x| \neq |y| \\ 0 & \text{if } |x| = |y| \end{cases}$$

denoting with $|x|$ the length of x and with K_{arg} a configuration parameter. In this way, arguments with the same length are clustered together. For each cluster, we compute the minimum and maximum value of the length of arguments. Fusion of models and incorporation of new elements are straightforward. The probability for a new input to belong to the model is 1 if its length belongs to the interval, and 0 otherwise.

We developed an ad-hoc model for *user and group* identifiers. These discrete values have three different meanings: `UID` 0 is reserved to the super-user, low values usually are for system special users, while real users have `UID`s and `GID`s above a threshold (usually 1000). So, we divided the input space in these three groups, and computed the distance for clustering using the following formula:

$$d_a = \begin{cases} K_{uid} & \text{if belonging to different groups} \\ 0 & \text{if belonging to the same group} \end{cases}$$

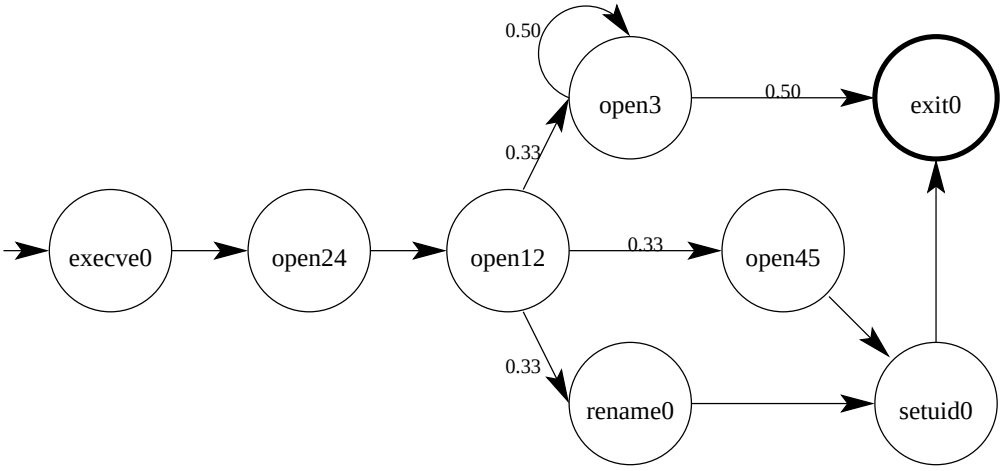

Figure 5.2: A sample of the resulting Markov model with the clusters of system calls as states

and K_{uid}, as usual, is a user-defined parameter. Since `UID`s are limited in number, they are preserved for testing, without associating a discrete probability to them. Fusion of models and incorporation of new elements are straightforward. The probability for a new input to belong to the model is 1 if the UID belongs to the learned set, and 0 otherwise.

In order to take into account the execution *context* of each system call, we use a Markov chain (i.e. a first order Markov model) to represent the program flow. The model states represent the system calls, or better they represent the various clusters of each system call, as detected during the clustering process. For instance, if we detected three clusters in the `open` syscall, and two in the `execve` syscall, then the model will have five states: $\texttt{open}_1$, $\texttt{open}_2$, $\texttt{open}_3$, $\texttt{execve}_1$, $\texttt{execve}_2$. Each transition will reflect the probability of passing from one of these groups to another through the program. A sample of such a model is shown in Figure 5.2. This approach was investigated in former literature [28, 30, 56, 81, 59, 61], but never in conjunction with the handling of parameters and with a clustering approach.

During training, each execution of the program in the training set is considered as a sequence of observations. Using the output of the clustering process, each syscall is classified into the correct cluster, by computing the probability value for each model and choosing the cluster whose models give out the maximum composite probability along all known models: $\max(\prod_{i \in \mathrm{M}} P_i)$. The probabilities of the Markov model are

then straightforward to compute.

Since training should happen, ideally, on the machine which will be monitored, it is important to notice that the prototype is resistant to the presence of a limited number of outliers (e.g. abruptly terminated executions or attacks) in the training set, because the resulting transition probabilities will drop near zero. For the same reason, it is also resistant to the presence of any cluster of anomalous invocations created by the clustering phase. Therefore, the presence of a minority of attacks in the training set will not adversely affect the learning phase, which in turn does not require an attack-free training set, and thus it can be performed on the deployment machine.

During the detection phase, each system call is considered in the context of the process. The cluster models are once again used to classify each syscall into the correct cluster: the probability value for each model is computed and the stored cluster whose models give out the maximum composite probability ($P_c = \max(\prod_{i \in \mathrm{M}} P_i)$) is chosen as the "system call class". Three distinct probabilities can be taken into account in order to build anomaly thresholds:

- P_s, the probability of the *execution sequence* to fit the Markov model up to now;
- P_c, the probability of the *system call* to belong to the best-matching cluster;
- P_m, the *latest transition* probability in the Markov model.

We fuse the last two into a probability value of the single syscall, $P_p = P_c \cdot P_m$. A second, separate value for the *sequence probability* P_s is kept. Using the training data, appropriate threshold values are calculated by considering the lowest probability over all the dataset for that single program (for both P_s and P_p). We then choose a sensitivity parameter for scaling such value, giving the final *anomaly threshold*. A process is flagged as malicious if either P_s or P_p are lower than the anomaly threshold. For avoiding a P_s which quickly decreases to zero for long sequences, we introduced a "scaling" of the probability calculation based on the geometric mean, by introducing a sort of "forgetting factor": $P_s(l) = \sqrt[2l]{\prod_{i=1}^{l} P_p(i)^i}$ (where l is the sequence length). In this case, we demonstrated [67] that $\mathrm{P}\left[\lim_{l \to +\infty} P_s(l) = 0\right] = 1$, but it converges more slowly. Experimentally, this latter scaling function leads to much better results in terms of false positive rate.

5.3 Experimental setup

In order to show that our system is capable of detecting the in-memory injection of code, and of creating an audit trail which can be used for forensics purposes, while at the same time reducing the logged data to the bare minimum that is needed, we generated an experimental dataset.

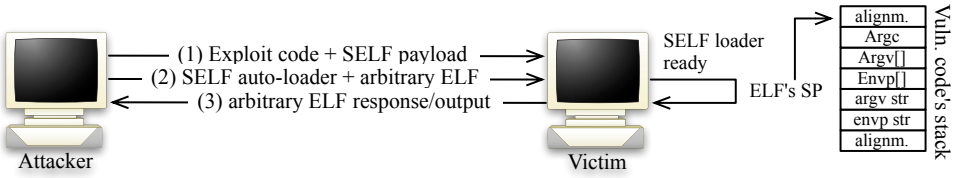

Figure 5.3: An illustration of the in-memory execution technique we developed and used for this work

We developed code injection attacks against applications on an Intel x86 machine running FreeBSD 6.2. We recompiled the kernel enabling auditing capabilities, and used OpenBSM [95] to collect audit trails (i.e. system calls sequences and their details).

We found vulnerabilities [74, 75] in two versions of `eject` and `bsdtar`, namely `mcweject 0.9` (which is an alternative to the `eject` command bundled with FreeBSD 6.2) and the release of `bsdtar` distributed with FreeBSD 6.2.

In order to train our system, we need a reasonable set of normal executions of all the involved commands. Using a process similar to the one used for creating the IDEVAL [2] dataset, and in fact used also in other works such as [90], we prepared shell scripts to emulate pseudo-random behaviors of a user.

We developed the exploits for the vulnerabilities, and used a specifically crafted payload, which implements a technique known as "Userland Exec", i.e., by overwriting the program headers of any statically linked ELF binary, and by building a specially-crafted stack it allows an attacker to load and run that ELF in the memory space of a target process without calling the kernel and, more importantly, without leaving any trace on the hard disk of the attacked machine. The idea was based on a tool named SELF [11], but our payload is a shellcode which can be executed through code injection, as opposed to the previously available POCs.

Our technique employs a two-stage attack where a shellcode is injected in the vulnerable program, and then retrieves a modified ELF from a remote machine, and subsequently injects it into the memory space of the running target process, as shown schematically in Figure 5.3.

In the setup detailed above, we performed several experiments with both `eject` and `bsdtar`. We trained our anomaly detector with ten different execution of `eject` and more than a hundred executions of `bsdtar`, randomized as described above. We also audited eight instances of the activity of `eject` under attack, while for `bsdtar` we logged

Details	*Detection accuracy*		
PROGRAMS	DR % = $\frac{TP}{TP+FN}$%		FPR % = $\frac{FP}{FP+TN}$%
Test env.:	**(a) w/o SELF**	**(b) w/ SELF**	**(c) Attack-free data**
`eject` (no. of execs.)	**75%** = $\frac{6}{6+2}$%	**100%** = $\frac{8}{8+0}$%	0% = $\frac{0}{0+404}$%
`bsdtar` (no. of execs.)	**70.6%** = $\frac{12}{12+5}$%	**100%** = $\frac{4 \cdot 2}{4 \cdot 2+0}$%	7.81% = $\frac{20}{20+236}$%

Table 5.2: Experimental results for DR with a (a) regular shellcode (without userland execution) and (b) with our userland exec implementation based on SELF. Test environment (c) is related to the data used to compute the FPR.

seven malicious executions. We repeated the tests both with a simple shellcode which opens a root shell (a simple `execve` of `/bin/sh`) and with our implementation of the userland exec technique. In the latter we injected four different statically built payloads (`sash`, `links`, `fget`, and the `sudoku` command line game); in the case of `bsdtar` we gathered 8 executions by invoking `bsdtar` with two different command line option sets; in the case of `eject` we injected 8 different payload (the same used for `bsdtar` plus `portsentry`, `tree`, `pstree`, `less`) and we audited eight different executions.

The overall results are summarized in Table 5.2. Let us consider the effectiveness of the detection of the attacks themselves. The attacks against `eject` are detected with no false positive at all. The exploit is detected in the very beginning: since a very long argument is passed to the `execve`, this triggers the argument model. The detection accuracy is similar in the case of `bsdtar`, even if in this case there are some false positives. The detection of the shellcode happens with the first `open` of the unexpected special file `/dev/tty`. It must be underlined that most of the true alerts are correctly fired at system call level; this means that malicious *calls* are flagged by our IDS because of their unexpected arguments, for instance. It must be noted that, in the case of userland execution with SELF, we were able of reaching 100% because our IDS is easily triggered by in memory attacks; in fact, executing the injected payload significantly modifies the normal behavior of the process more than a classic exploit does. Also note that to test the accuracy of the prototype we used attack-free data.

On the other hand, exploiting the "Userland Exec" an attacker launches an otherwise normal executable, but of course such executable has different system calls, in a different order, and with different arguments than the ones expected in the monitored process. This reflects in the fact that we achieved a 100% detection rate with no increase in false positives, as each executable we have run through SELF has produced a Markov model

which significantly differs from the learned one for the exploited host processes.

We profiled the code with `gprof` and `valgrind` for CPU and memory requirements. The throughput for the training phase varies between 6120 and 10228 syscalls per second. The training phase is also memory consuming, with a worst-case peak during our tests of about 700 MB. The performance observed in the detection phase varies between 12395 and 22266 syscalls/sec. Considering that the kernel of a typical machine running services such as HTTP/FTP on average executes system calls in the order of thousands per second (e.g., around 2000 system calls per second for `wu-ftpd` [73]), and that in the context of WOMBAT we are thinking about the use of S^2A^2DE in the context of honeypot operations (where system load is arguably much lower), the overhead introduced will not impact system operations in our target context.

Of course, using the prototype in a real honeypot introduces the issue of survivability, i.e. wheteher or not an intruder can compromise the auditing system. This is a common issue for any type of logging system: as soon as the host is compromised at root level, any running auditing program cannot be trusted anymore. However, compromising our prototype would entail uploading a training file which accepts the attacker's actions as normal. This is definitely nontrivial: the best choice for an attacker would probably be to deactivate our system altogether. However, this can happen only in the post-exploitation phase, whereas detection hopefully happens during exploitation. If the logger is configured to forward alerts to a remote syslog server, the attacker would not be able to easily circumvent it.

6 Behavioral detection by grammar-based signatures

Behavioral detection should theoretically be able to detect, if not innovative malware, at least unknown malware reusing variations of known techniques. However, most of the current behavioral detectors rely on specific characteristics, allowing evasion through simple modifications at the functional level. In Deliverable D08 (D4.1): Specification language for code behavior [98], a generative grammar for the *Abstract Malicious Behavioral Language (AMBL)* has been provided to model malicious behaviors, describing their generic principle rather than their technical implementations. This chapter shows the usage of behavioral signatures declared in the *AMBL* to build efficient and resilient parsing automata for detection.

In a detection context, deterministic finite automata are attractive because their linear complexity remains acceptable for operational deployment. Already in 1995, [32] used automata to describe the alternative sequences of operations making up malicious behaviors. Detection was then restricted to behaviors described by classes of grammars insensitive to the context. Since then, a focus on data flow has led to the apparition of tainting techniques to detect malicious uses of data [77]. After significant successes, control of the data flow is now broadly used, in intrusion detection [25] or malware behavior extraction [33]. The data-flow being context-sensitive, it requires more evolved automata, such as pushdown automata, to be handled. In practice, the automata embed the sequences of system calls constituting respectively attacks and behaviors. The data flow is then captured by analysis of the parameters collected along the system calls. Following this principle, [72] focuses on self-reproduction as the discriminating behavior for detection whereas [70] focuses on bots behaviors. The approach of behavioral detection that we present in this chapter also combines automata and data flow control. But, according to the declarative approach of [86], behavior signatures are first declared within the *AMBL* instead of being directly embedded into automata like the previously mentioned articles.

Starting from the declared signatures, parsing automata are built for behavioral detection by syntax checking and semantic evaluation. The *AMBL* semantic attributes, specified for binding and typing, increase the linking between the operations making up the behaviors. In reference to intrusion scenarios [38, 79], these attributes eventually constitute two sets referred to as prerequisites and consequences, evaluated at every step of the automata. Through prerequisites and consequences, operations unrelated to the

behavior are precisely identified. Unlike traditional parsing, unrelated symbols must be dropped to keep on with the detection process, similarly to the event filters formalized in [87]. An other difference with parsing is that detection searches, in a single pass, for multiple instances of a same behavior, some possibly incomplete. Just like in [87], derivation duplication is used to handle these multiple instances without risk of missing one.

In input to parsing, collection mechanisms supply raw data and abstraction is needed to translate the observed traces into the behavioral language. [70] addresses by a layered architecture the semantic gap existing between the system call traces, understandable by OS specialists, and high-level behaviors. An abstraction layer for translation into the *AMBL* was introduced in [98, Chpt.3.2.3].

This chapter covers the successive layers of the process as published in [58]. Section 6.1 defines the detection layer in terms of parsing automata, allowing their formal assessment. In particular, we have been able to identify the classes of attribute-grammars acceptable for signature detection in a single pass, but also to assess the detection complexity in various cases. Since detection only provides information about independent behaviors, Section 6.2 addresses behavior correlation over the parsing results, in order to merge this information and profile malware into families. Implementation of the different layers is covered in Section 6.3 in order to provide in Section 6.4 a second operational assessment in terms of coverage and performance.

6.1 Detection by parsing automata

In a grammatical model, detecting malicious behaviors is reduced to parsing their descriptions. According to [98], the *AMBL* is well-formed, thus guaranteeing a possible order for semantic attribute valuation. However, in a detection context, this property is insufficient. Deployed in real-time, the detector is confronted to a continuous flow of data forbidding the decoupling of syntactic parsing from semantic evaluation into consecutive processes. Syntactic parsing and semantic evaluation must thus be achieved in a single-pass. To satisfy this constraint, attribute grammars must either be LL and L-attributed grammars, or LR and S-attributed grammars [97, Chpt.10]. By definition, LL-grammars are parsed from Left to right in order to construct Leftmost derivations whereas LR-grammars construct Rightmost derivations. As specified in Definition 1, L-attribute grammars only allow attribute dependency from left to right in the production rules. S-attributed grammars specified in Definition 2 are included within L-attributed grammars and only authorize synthesized attributes. With respect to syntax, LR-parsers can handle a larger class of grammars than LL-parser. However, the *AMBL* has very

simple syntactic rules. Sematic evaluation will thus constitute our main choice criteria. By definition of the language, typing attributes are inherited. LR-parsers using a bottom-up approach will thus be missing the typing information inherited from parent nodes. LL-parsers have thus been chosen because of their capacity to handle larger classes of semantic attributes. We therefore constrain the description generation within the *AMBL* to LL and L-attributed subgrammars.

Definition 1 *In an L-attributed grammar, attribute dependencies on the right-hand sides of productions are only allowed from left to right positions. An attributed grammar G is L-attributed if for every* $\pi \in P$ *and* $Y_i.\alpha = f(..., Y_j.\beta, ...)$ *with* $\alpha \in Inh$ *and* $\beta \in Syn$*, we have* $i < j$*.*

Definition 2 *An attributed grammar G is S-attributed if every of its attribute is synthesized.*

Definition 3 *A LL-parser A is an extended pushdown automaton that can be built as a ten-tuple* $< Q, \Sigma, D, \Gamma_p, \Gamma_s, \delta, q_0, Z_{p,0}, Z_{s,0}, F >$ *where:*
- Q *is the finite set of states, and* $F \subset Q$ *is the subset of accepting states,*
- Σ *is the alphabet of input symbols and* D *is the set of values for attributes,*
- Γ_p */* Γ_s *are the parsing / semantic stack alphabets,*
- $q_0 \in Q$ *is the initial state and* $Z_{p,0}$ */* $Z_{s,0}$ *are the stacks start symbols,*
- δ *is the transition function defining the production rules and semantic routines, of the form:* $Q \times (\{\Sigma \cup \epsilon\}, D^*) \times (\Gamma_p, \Gamma_s) \rightarrow Q \times (\{\Gamma_p \cup \epsilon\}, \Gamma_s)$*.*

From the behavioral descriptions, the LL-parsers for detection are constructed as pushdown automata, enhanced with attribute evaluation in order to recognize their synatx and semantic [97, Chpt.10]. To build the detector, several behaviors are monitored in parallel, each one parsed by a dedicated automaton as represented in Figure 6.1. According to Definition 3, these automata are capable of building, from top to down, the annotated leftmost-derivation trees by using two different stacks for syntactic symbols and semantic attributes. However, the construction of these automata differs from traditional parsing, thus explaining that we did not use parser generators such as *ANTLR* [82]. In fact, each automaton A_k, associated to the k^{th} behavior, parses at the same time several instances of the behavior, storing its progress in independent derivations. These derivations correspond to triples made up of the current state q_k and the content of the parsing and semantic stacks, Γ_{pk} and Γ_{sk}. Through the abstraction layer, sequences of events e_i are collected and translated into input symbols and semantic values of the recognized language. The parsing automata, deployed in parallel, are fed with all these events and progress along their derivations. These events may appertain to

any behavioral instance, so all the derivations handled by a given automaton are independently updated. When an irrelevant input is read (an interleaved operation inside the behavior for example), this input is ignored instead of causing an error state in a derivation. When an ambiguous input is read (a seemingly relevant operation that does not eventually help to the behavior completion), the derivation is duplicated to handle new instances. Individual parsers and the global procedure are respectively defined in Algorithms 1 and 2. The handling of irrelevant events and ambiguous events are respectively described in greater details in Sections 6.1.1 and 6.1.2. The resulting parsing complexity is finally addressed in Section 6.1.3.

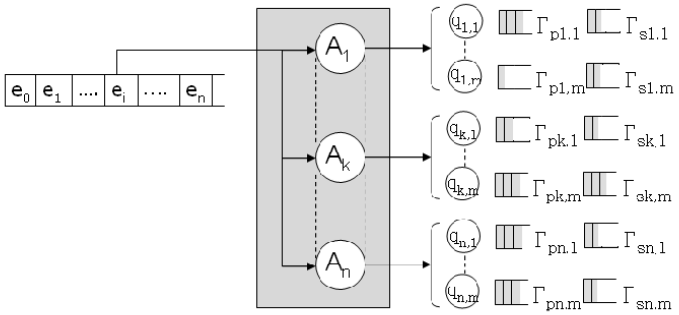

Figure 6.1: Detection by parallel automata.
The n automata correspond to the different monitored behaviors. Each automaton handles several parallel derivations with independent states and stacks in order to handle ambiguities.

6.1.1 Semantic prerequisites and consequences

The present detection method can be related to scenario recognition in intrusion detection. An intrusion scenario is defined as a sequence of dependent attacks [38, 79]. For each attack to occur, a set of prerequisites or preconditions must be satisfied. Once the attack completed, new consequences are introduced, also called postconditions. In [16], isolated alerts are correlated into scenarii by parsing attribute-grammars annotated with semantic rules to guarantee the flow between related alerts. Similarly, a malicious behavior is a sequence where each operation prepares for the next one. In a formalization by attribute grammars, the sequence order is ensured by the syntax whereas prerequisites

Algorithm 1 A.ll-parse(e,Q,Γ_p,Γ_s).

if e, Q, Γ_p, Γ_s match a transition $T \in \delta_A$ **then**
  if e introduces a possible ambiguity **then**
    duplicate state and stack triple (Q, Γ_p, Γ_s). {*Start new parallel derivation*}
  end if
  compute transition T to update (Q, Γ_p, Γ_s).
  if Q is an accepting state $Q \in F_A$ **then**
    alert "malicious behavior detected".
  else
    ignore e.
  end if
end if

Algorithm 2 BehaviorDetection(e_1,...,e_t).

Require: events e_i are couples of symbol and semantic values: $(\{\Sigma \cup \epsilon\}, D^*)$.
for all collected events e_i **do**
  for all the automata A_k such as $1 \leq k \leq n$ **do** {*Detection of n behaviors*}
    m = number of derivations.
    for all state and stack triples $(Q_{k,j}, \Gamma_{pk,j}, \Gamma_{sk,j})$ such as $1 \leq j \leq m$ **do**
      A_k.**ll-parse**(e_i,$Q_{k,j}$,$\Gamma_{pk,j}$,$\Gamma_{sk,j}$).
    end for
  end for
end for

and consequences are ensured by semantic rules of the form $Y_i.\alpha = f(Y_1.\alpha_1...Y_n.\alpha_n)$ according to Definition 4.

- **Checking prerequisites:** Prerequisites are defined by specific semantic rules where the left-side attributes of the equations are attached to terminal symbols ($Y_i \in \Sigma$). During parsing, semantic values are collected along input symbols. These values are compared to values computed using inherited and already synthesized attributes. This comparison corresponds to the matching performed on the semantic stack Γ_s during transitions from δ. As in [87], symbols failing to satisfy the prerequisites are simply ignored instead of raising errors.
- **Evaluating consequences:** When the left-side attribute is attached to a non-terminal ($Y_i \in V$) and all right-side attributes are valued, the attribute is evaluated. During the transitions from δ, the evaluation corresponds to the reduction step where the computed value is pushed on the semantic stack Γ_s. Once computed, the consequences can impact next transitions by being integrated to their prerequisites.

Definition 4 *An attribute-grammar G_A is a triplet $<G, D, E>$ where:*
- G is originally a context-free grammar $<V, \Sigma, S, P>$,
- $att : X \in \{V \cup \Sigma\} \rightarrow att(X) \in Att^*$ is an assignment function for attributes and $D = \cup_{\alpha \in Att} D_\alpha$ their set of values,
- E is a set of semantic rules such as for any production of P, there is at most one rule per variable of the form $Y.\alpha = f(Y_1.\alpha_1...Y_n.\alpha_n)$ with $f : D_{\alpha_1} \times ... \times D_{\alpha_n} \rightarrow D_\alpha$.

6.1.2 Ambiguity support

All events are fed to the behavior automata. However, some of them may be unrelated to the behavior or unuseful to its completion. Unrelated events do not match any transition and are simply dropped as explained in Section 6.1.1. This is insufficient for unuseful events raising ambiguities: they may be related to the behavior but parsing them makes the derivation fail unpredictably. Let us take an explicit example for duplication in Figure 6.2. After opening the self-reference, two files are consecutively created. If duplication is achieved between the self-reference and the first file, parsing succeeds. If duplication is achieved with the second one, parsing fails because the automaton has progressed beyond the state of accepting a second creation. Similar ambiguities may be observed along the variable affectations which alter the data-flow.

The algorithm should thus be able to manage the different objects and variables combinations. As represented in Figure 6.2, ambiguities are handled by the detection algorithm using derivation duplicates. This solution guarantees that no behavior instance can be missed as proven by the completeness proof in [87]. Before transition reduction,

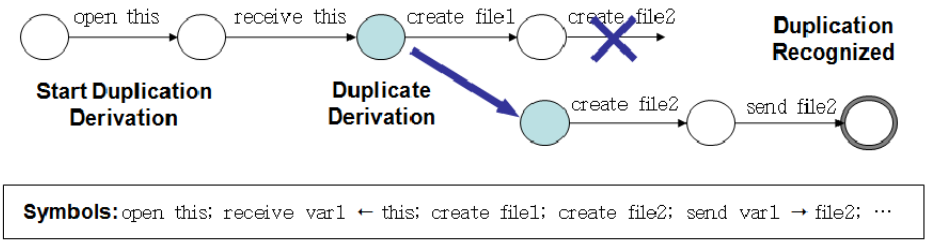

Figure 6.2: Handling ambiguous symbols.
To resist interleaved interactions, the derivation is duplicated before transition to handle alternate object combinations.

if the operation is potentially ambiguous, the current derivation is copied in a new triple containing the current state and the parsing and semantic stacks. This solution handles the combinations of events without backtracking. To come back and forth in the derivation trees would have proved too cumbersome for real-time detection. To avoid an explosion in the number of derivations, derivations, as soon as they become useless, may be destroyed as it will be presented in Section 6.3.3 on implementation.

6.1.3 Time and space complexity

LL-parsing is linear in function of the number of symbols [52]. However, parallelism and ambiguities increase the complexity of the detection algorithm. Let us consider calls to the parsing procedure as the reference operation. This procedure is decomposed in three steps: matching, reduction and accept (two comparisons and a computation). In the worst case scenario, all events are related to the behavior automata and all these events introduce ambiguities. In the best case scenario, no ambiguity is raised. Resulting complexities are given in Proposition 1.

Proposition 1 *In the worst case, behavioral detection using attributed automata has a time complexity in* $\vartheta(k(2^n - 1))$ *and a space complexity in* $\vartheta(k2^n(2s))$ *where* k *is the number of automata,* n *is the number of input symbol and* s *is the maximum stack size. In the best case, time complexity drops to linear time* $\vartheta(kn)$ *and space complexity becomes independent of the inputs* $\vartheta(k2s)$.

The worst case complexity is important but it quickly drops as the number of ambiguous events decreases. The experimentations in Section 6.4.5 show that the ratio of

ambiguous events is limited and the algorithm offers satisfactory performances. Based on this ratio, a new assessment of the average practical complexity is provided. Besides, these experimentations also show that an important ratio of ambiguous events are already a sign of malicious activity.

Proof 1 *In a best case scenario, the number of derivation for each automaton remains constant. Considering the worst case scenario, all events are potentially ambiguous for all the current derivations. Technically, ambiguities multiply by two the number of derivations at each iteration of the main loop. Consequently, each automaton handles* 2^{i-1} *different derivations at the* i^{th} *iteration. The time complexity is then equivalent to the number of calls to the parsing procedure:*

(1) $k + 2k + ... + 2^{n-1}k = k(1 + 2 + ... + 2^{n-1}) = k(2^n - 1)$

The maximum number of derivations is reached after the last iteration. In the worst case, all automata manage 2^n *parallel derivations. Each derivation is stored in two stacks of size* s*. This moment thus coincides with the maximum memory occupation:*

(2) $k2^n(2s)$.

6.2 Profiling the main classes of malware

In the previous sections, a behavioral approach for detection has been provided. Strictly speaking, this approach does not detect malware, but offers a finer-grained approach by detecting the independent malicious behaviors encountered inside these malware. A complete detection scheme, as presented in Definition 5, requires a third layer, above translation and individual detection, for behavior correlation. The interest of correlation is twofold. It first reduces the risks of false positives. The experimentations coming in Section 6.4.3 show that some behaviors are more discriminating than others. Correlation is a way to give these significant behaviors a greater weight in the detection process. In addition, correlation can also be used to associate individual behaviors with a family the malware instance belongs to.

Definition 5 *A behavioral detection scheme is the pair* $\{\mathcal{B}, \phi_c\}$ *where* $\mathcal{B}$ *is a set of behavior signatures defined as Boolean variables and* $\phi_c : \mathbb{F}_2^{|\mathcal{B}|} \rightarrow \mathbb{F}_n$ *is a Boolean correlation function for detection,* $\mathbb{F}_n$ *being the n-ary field indexing legitimate programs and malware families [42, 43].*

Resulting of detection by automata, the Boolean variables corresponding to the monitored behaviors $\mathcal{B}$ are resolved. These variables may express the simple behavior presence

(example (1)). However, since the detection automata provide richer information than behaviors alone, these variables can also convey more meaningful expressions. Additional information can be recovered from the derivation trees built by the automata during parsing. For example, a duplication derivation tree distinguishes the possible data flows, between direct transfer, single read/write or interleaved reads/writes (example (2)). Through the semantic annotations of the tree, information about the duplication target can also be recovered such as its name or its status: existing or created by the malware (example (3)). All this information constitutes additional Boolean variables that can be fed into the correlation process, to increase its deduction capability.

$$(1)\ X_{\beta} = \begin{cases} 1 & if\ \beta\ has\ been\ identified \\ 0 & otherwise \end{cases}$$

$$(2)\ X_{\beta,m} = \begin{cases} 1 & if\ \beta\ has\ been\ identified\ using\ method\ m \\ 0 & otherwise \end{cases}$$

$$(3)\ X_{\beta,o,s} = \begin{cases} 1 & if\ \beta\ manipulates\ object\ o\ with\ status\ s \\ 0 & otherwise \end{cases}$$

A solution to finally build the correlation function ϕ_c is to establish, according to the common properties of their behaviors, profiles for the generic classes of malware. These profiles can be specified by belonging conditions, using all the behavioral information at our disposal. In Figure 6.3, we have put forward profiles for different kinds of Viruses, Trojans and Worms, their belonging conditions expressed as Boolean statements.

6.3 Prototype implementation

As a proof of concept, a prototype of behavioral detector has been designed, satisfying the formalization of the previous sections. We have developed a first version of the prototype, which includes the two aforementioned layers: a specific data collection and abstraction layer and a generic detection layer. The second version has been enhanced with an additional layer for behavior correlation by profiles. The overall architecture is described in Figure 6.4. For the abstraction layer, dedicated components capture the features of different languages whereas a common object classifier apprehends the platform-specific elements of the environement. In order to cover different use cases, abstraction components have been designed for two different languages, and both collection methods: a native language through the interpretation of dynamic traces of *PE Executables* in Section 6.3.1 and, to show the independence of detection from the collection method, an interpreted language through the static analysis of *Visual Basic Script* in Section 6.3.2. Above abstraction, the detection layer described in Section 6.3.3 de-

Profile for the Virus class:

$duplication.number \geq 1$
$duplication.target.status \in \{existing\}$
File overwriter subclass:
$duplication.flow \in \{transfer\}$
File infector subclass:
$duplication.flow \in \{single\ read/write, interleaved\ read/write\}$

Profile for the Trojan class:

$duplication.number \geq 1$
$executionproxy.number \geq 1$

Profile for the Net Worm class:

$propagation.number \geq 1$
$propagation.interface \in \{network\}$

Profile for the Mail Worm class:

$duplication.number \geq 1$
$propagation.number \geq 1$
$propagation.interface \in \{mail\}$

Profile for the P2P Worm class:

$duplication.number \geq 1$
$propagation.number \geq 1$
$propagation.interface \in \{file, folder\}$

Profile for the Drive Worm class:

$duplication.number \geq 1$
$propagation.number \geq 1$
$propagation.interface \in \{drive\}$
Amovible drive subclass:
$residency.target.name \in \{autorun.inf\}$
Generic drive subclass:
$residency.target.name \notin \{autorun.inf\}$

Profile for the IRC Worm class:

$duplication.number \geq 1$
$\vee\ propagation.number \geq 1$
$residency.number \geq 1$
$residency.target.name \in \{mirc.ini, script.ini\}$

Figure 6.3: Generic Malware Profiles.
The profiles are mainly built on the presence of specific behaviors inside malware, but additional parameters, corresponding to derivation-related and semantic information, refine the belonging conditions.

ploys parallel automata parsing the interpreted traces independently from their original source. The behavioral information extracted by the automata is finally correlated by the profiling layer described in Section 6.3.4, in order to classify malware.

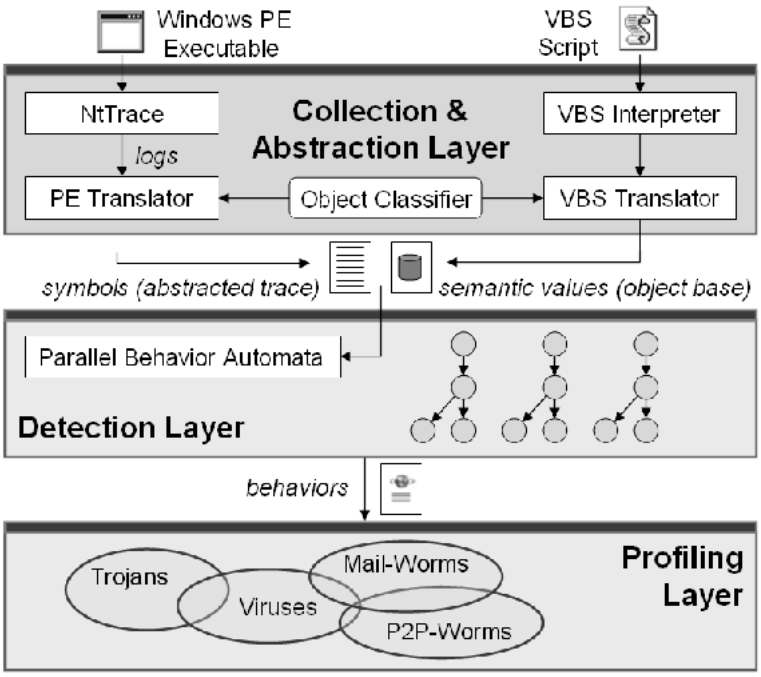

Figure 6.4: Multi-layered architecture of the detector.
The detector prototype is constituted of three stacked layers, making-up the global detection process. Each layer handles more generic and synthetic data, starting from the collected raw traces, passing by detected behaviors, to the above malware classification.

6.3.1 Analyzer of process traces

Process traces provide useful information about the system activity of an executable. Whatever the considered operating system, different dynamic tools exist to capture these traces of system calls. The prototype deploys a free tool called *NtTrace* which has been chosen for its capacity to collect *Windows Native Calls*, their arguments as well as their returned values [6].

1) Collection environment: Contrary to static analysis, the main point with dynamic collection mechanisms, either real-time or emulation based, is that most behaviors are conditioned by external objects and events, such as available target for infection or listening servers for network propagation. The configuration of the collection environment is thus critical. For trace collection, the virtual environment from Figure 6.5 has been installed over *Qemu* [8] using a drive image under *Windows XP*. In order to increase the mechanism coverage and collect conditioned behaviors, useful services and resources were configured or installed: system time, Internet Service Provider accounts, mail and peer-to-peer clients, potential targets (executables, pictures, music, web pages). To create a more realistic network configuration, emulations of DNS and SMTP servers have been deployed outside the virtual machine. These servers are not used to directly collect data but their presence is mandatory to establishing network connections and exchanges. They constitute the only way to capture the associated trace, containing the network activity at the system call level. Additional servers for IRC (*Unreal*) and FTP (*FileZilla*) have been deployed in a second step to observe any botnet activity for the related samples. *NtTrace* is finally run inside the virtual operating system, outputting system call traces as text files.

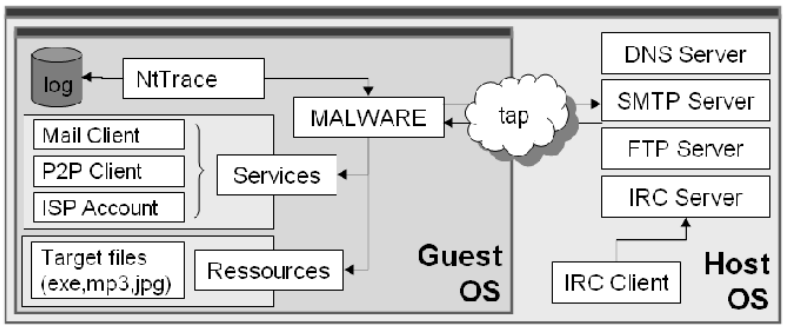

Figure 6.5: Collection environment for Windows API calls.
For an optimal coverage, the virtual environment is configured for the maximum similitude with the configuration of a personal computer, considering an average user.

2) Trace analysis: On top of the collection tool, we have developed an analyzer for line by line translation of the collected traces. Referenced APIs are directly classified over the different interaction categories according to Table 6.1, whereas unreferenced APIs are simply ignored until their integration in a future version. Sequences of identical calls as well as sequences of two combined calls are detected during the analysis and formatted

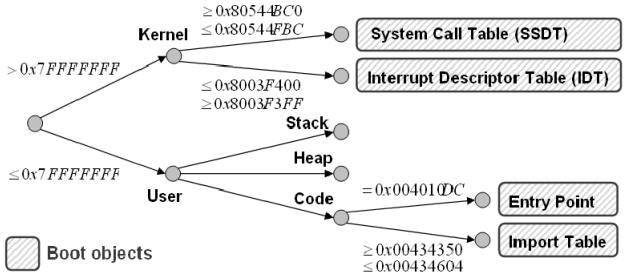

Figure 6.6: Addresses interpretation by space partitioning.

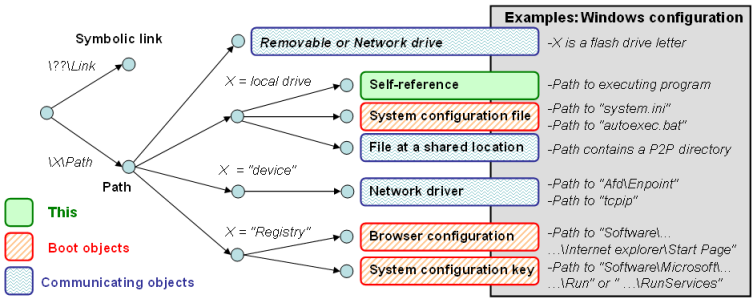

Figure 6.7: Character strings interpretation by structural analysis.

into loops in order to compress the resulting abstract trace.

In addition to interactions, the analyzer must be able to manage objects through identification and typing. In order to enforce typing on the call parameters, an object classifier, embedding decision trees such as the ones described in Figures 6.6 and 6.7, has been specifically designed for a *Windows* configuration. The identification of objects is more complex. Looking specifically at creation and opening interactions, their resolution establishes a correspondence between the names of the involved objects and their references, either addresses or handles. The correspondence is stored in a dedicated object base which is looked up during the analysis of the following calls. The code sample from Figure 6.8 illustrates the management of object correspondences inside the prototype. Conversely, deleting and closing interactions destroy correspondences for the remainder of the analysis. Names and identifiers must be unlinked since references could be reused for a different object. The identification of variables in reading interactions is a last point worth mentioning. The manipulated variables do not simply replaced each other

like handles; they may overlap. Let us consider a first variable defined by an address a_1 and a size s_1. Any reading interaction storing its result at the address a_2 such as $a_1 < a_2 < a_1 + s_1$ creates a second variable and reduces the size of the first variable to $a_2 - a_1$ like in the code sample from Figure 6.9.

Table 6.1: Mapping Windows Native and VBScript APIs to interaction classes.

Interaction Class	Object Nature	Windows Native API	VBScript API
Open	File	**NtOpenFile**(ptr FileHandle, ..., str FilePath, ...) **NtCreateSection**(ptr SectionHandle, ..., ptr FileHandle)	FileSystemObject.**GetFile**(str FilePath) FileSystemObject.**GetFolder**(str FilePath) FileSystemObject.**OpenTextFile**(str FilePath) FileSystemObject.**FileExists**(str FilePath) FileSystemObject.**GetDrive**(str DivePath) FileSystemObject.**Drives.Item**(int DriveNumber)
	Registry	**NtOpenKey**(ptr KeyHandle, ..., str KeyName, ...) **NtEnumerateKey**(ptr KeyHandle, ...)	
	Network	**NtOpenFile**(ptr DeviceHandle, ..., **str NetworkDevicePath**, ...)	
Create	File	**NtCreateFile**(ptr FileHandle, .. , str FilePath, ...)	FileSystemObject.**CreateFolder**(str FilePath) FileSystemObject.**CreateTextFile**(str FilePath)
	Registry	**NtCreateKey**(ptr KeyHandle, ..., str KeyName, ...)	
	Network	**NtCreateFile**(ptr FileHandle, .. , str NetworkDevicePath, ...)	
	Mail		**CreateObject**("CDO.Message") **CreateObject**("Cdonts.NewMail") OutlookApplication.**CreateItem**(int ItemNumber)
Close	File	**NtClose**(ptr FileHandle)	FileObject.**Close**()
	Registry	**NtClose**(ptr KeyHandle)	
	Network	**NtClose**(ptr DeviceHandle)	
Delete	File	**NtDeleteFile**(str FilePath)	FileSystemObject.**DeleteFile**(str FilePath) FileSystemObject.**DeleteFolder**(str FilePath)
	Registry	**NtDeleteKey**(ptr KeyHandle)	ShellObject.**RegDelete**(str KeyName)
Read	File	**NtReadFile**(ptr FileHandle, ..., ptr Buffer, ...) **NtReadFileScatter**(ptr FileHandle, ..., ptr SegmentArray, ...) **NtMapViewOfSection**(ptr SectionHandle, ..., ptr BaseAddress, ...)	FileObject.**Read**() FileObject.**ReadLine**() FileObject.**ReadAll**()
	Registry	**NtQueryValueKey**(ptr KeyHandle, str Value, ..., ptr Buffer, ...)	ShellObject.**RegRead**(str KeyName)
	Network	**NtDeviceIoControlFile**(ptr DeviceHandle, ..., **ReadControl**, ptr Buffer, ...)	
Write	File	**NtWriteFile**(ptr FileHandle, ..., ptr Buffer, ...) **NtWriteFileGather**(ptr FileHandle, ..., ptr SegmentArray, ...)	FileObject.**Write**(str Value) FileObject.**WriteLine**(str Value) FileObject.**Copy**(str FilePath) FileObject.**Move**(str FilePath) FileSystemObject.**CopyFile**(str FilePath,str FilePath) FileSystemObject.**MoveFile**(str FilePath str FilePath)
	Registry	**NtSetValueKey**(ptr KeyHandle str Value, ..., str Buffer, ...)	ShellObject.**RegWrite**(str KeyName, str Value)
	Network	**NtDeviceIoControlFile**(ptr DeviceHandle, ..., **SendControl**, ptr Buffer, ...)	
	Mail		MailObject.**TextBody**(str Content) MailObject.**Body**(str Content) MailObject.**AddAttachment**(str FilePath) MailObject.**AttachFile.Add**(str FilePath) MailObject.**Attachments.Add**(str FilePath)
Execute	Process	**NtCreateProcess**(..., ptr SectionHandle, ...)	Wscript.**Run**(str Commmand) ShellObject.**Run**(str Command)

6.3.2 Analyzer of Visual Basic Scripts

No collection tool similar to *NtTrace* is available for *VBScript*. A dedicated collection tool has thus been developed, embedding the abstraction layer directly. *VBScript* being

```
if(!strncasecmp(OPENF1,line,10)){  //NtOpenFile(@[handle], ..., filename, ...)
    //Parsing arguments
    args = strchr(line,'('); args++;
    objtoken1 = strtok(args,",");
    token = strtok(NULL,",");
    filename = strtok(NULL,",[]");
    token = strtok(objtoken1," []");
    token = strtok(NULL," []"); sscanf(token,"%X",&handle1);
    //Updating object base
    objind = isKnownObject(types,filename,0);
    if(objind==UNKNOWN) objind = addNewObject(types,filename,OBJ_FILE);
    if(handle1) addObjectHandle(types,objind,handle1);
    *obj1 = objind;                //Object parameter
    return OP_OPEN;                //Recognized command
}
```

Figure 6.8: Recognition of opening interactions.
In input, `line` is read from the process trace. If `NtOpenFile` is recognized, its arguments are parsed to manage objects. A look up determines if the object is existing in the base or must be created. A correspondence is then established with the returned handle value.

an interpreted language, its static analysis was easier to consider than for native code, because of the visibility of the source code and its integrated safety properties: no direct code rewriting during execution and no arbitrary transfer of the control flow [69]. Relying on these advantages, we have conceived the VBScript Analyzer as a partial interpreter using static analysis for path exploration. The analyzer is divided into three parts: a static part recovering the script structure and normalizing its code, a second dynamic part exploring the different execution paths and collecting significant events, and the object classifier.

1) Static analyzer: The static analysis heavily relies on the syntactic specifications of the *VBScript* language [5]. The script is first parsed to localize the main, the local functions and procedures, as well as to retrieve their signature. Its structure is then parsed by blocks to recover information about the declared variables and the instantiated managers (file system, shell, network, mail). In addition to information collection, the static analyzer also deploys code normalization. Code normalization removes several syntactic shortcuts provided by *VBScript* but most critically thwarts obfuscation and encryption. By normalization, the current version of the analyzer can handle certain categories of obfuscation such as integer encoding, string splitting or string encryption.

2) Dynamic interpreter: A partial script interpreter has been defined to explore the

```
if(!strncasecmp(READF1,line,10)){      //NtReadFile(@[handle], ..., buffer, size, offset)
    //Parsing arguments
    args = strchr(line,'('); args++;
    token = strtok(args,","); sscanf(token,"%X",&handle2);
    token = strtok(NULL,","); ...      //Skip the four next parameters
    token = strtok(NULL,", "); sscanf(token,"%X",&ptr1);
    token = strtok(NULL,", "); sscanf(token,"%X",&size);
    objtoken1 = strtok(NULL,",)");
    token = strtok(objtoken1,"[]");
    token = strtok(NULL,"[]"); sscanf(token,"%X",&offset);
    //Updating object base
    objind2 = isKnownObject(types,NULL,handle2);
    if(objind2==UNKNOWN) return 0;
    objind1 = UNKNOWN;
    for(i=0; i<types->nbobj; i++){
        address = getObjectAddress(types,i);
        space = getObjectSize(types,i);
        if(address==ptr1){
            if(!objind1) objind1 = i;  //Reuse known variable
        }else if(ptr1>add && ptr1<(add+addsize)){
            diff = ptr1-address-1;     //Restraining variable size
            setObjectSize(types,i,diff);
        }
    }
    if(!objind1){  //Creating second variable
        objind1 = addNewObject(types,NULL,VAR);
        setObjectAddress(types,objind1,ptr1);
    }
    setObjectSize(types,objind1,size);
    *obj1 = objind1; *obj2 = objind2;  //Object parameters
    return OP_READ;                    //Recognized command
}
```

Figure 6.9: Recognition of reading interactions.
The basic functioning is identical than for opening interactions except for variable management. If the manipulated variable is unknown, a new one is simply created using the given address and size. In case of overlapping, an overwriting variable is created; original variables are maintained but their size is reduced to respect the boundary of the created variable.

different execution paths. This interpreter has a partial capability, only in the sense that the script code is not really executed but only significant operations and dependencies are collected. To support path exploration, the analyzer handles conditional structures,

loop structures, and calls to local functions and procedures. Inside these different code blocks, each line is processed to retrieve the monitored API calls manipulating files, registry keys, network connections or mails. Monitored calls are interpreted by mapping according to the Table 6.1. Variable affectations, greatly impacting the data-flow, are thereby also monitored. With respect to the call arguments and the affected values, a second level of analysis is deployed to process these expressions. In order to control the data-flow, object references and aliases must be followed up through the processing of expressions, and in particular at some key operations:

- Local function and procedure calls - linking signature names with the passed arguments,
- Monitored API calls - creating new objects or updating their type and references,
- Affectations - linking variables with affected values,
- Calls to execute - evaluating expressions as code.

3) Object classifier: The previous object classifier has been reused as shown in the architecture of Figure 6.4. However, scripts being mainly based on character strings, the address classifier is unused. In addition, extensions to the string classifier have been implemented to best fit the script particularities, with new constants for the self-reference for example (`"Wscript.ScriptName"`, `"Wscript.ScriptFullName"` containing the path to the running script).

6.3.3 Detection automata

The real implementation of the detection automata complies with the algorithm presented in Section 6.1. The current version we have developed supports five different automata detecting respectively duplication, propagation, residency, overinfection and execution proxy behaviors [58]. As shown in the code sample from Figure 6.10, the production rules from the grammatical behavior descriptions have been directly coded as state transitions inside the automata. Semantic prerequisites have been integrated as tests conditioning these transitions whereas consequences are computed when resolving them. In input, the automata are fed with the traces of abstracted events, obtained by the analyzers. Notice that both analyzers format their traces in a same binary format for interoperability. For each behavior detected along parsing, a new entry is written down in a behavior report. In order to enrich the behavioral reports, the object databases containing all semantic values related to traces are also loaded. In output, the global report is finally formatted in an *XML* format satisfying the *Data Type Definition* presented in Figure 6.11.

With respect to the original algorithm, two enhancements have been brought to in-

```
/** updateDuplicationAutomata()
 * According to the operation symbol in input, the function dispatches the control to the
 * production rule whose first set contains the operation (Construction of LL-parsers [57]).
 */
void updateDuplicationAutomata(unsigned long ul_Operation,
    long l_Arg1id, int i_Arg1type, long l_Arg2id, int i_Arg2type){
    switch(ul_Operation){
    case OP_OPEN:
        parseDupOpen(l_Arg1id, i_Arg1type);
        break;
    case OP_CREATE:
        parseDupCreate(l_Arg1id, i_Arg1type);
        ...
    }
}

void parseDupOpen(long l_Argid, int i_Arg1type){
    for(i=0; i<duplication.nbderivation; i++){
        struct PARSED_AUTOMATON * aut;
        aut = &duplication.derivations[i];            //Selects ith derivation
        curstate = getCurrentState(aut);              //Recovers derivation state
        getCurrentAttributes(aut,t_curids,t_curtypes); //Recovers derivation semantic stack
        switch(curstate){
        case q1:
            if(i_Argtype==TYPE_THIS){                 //Checks semantic rules
                startDerivation(&duplication, q1,
                                t_curids,t_curtypes); //Duplicate derivation (ambiguity)
                t_curids[1] = l_Argid;                //Computes semantic values
                t_curtypes[1] = i_Argtype;
                addNode(aut,q2,t_curids,t_curtypes);  //Progression towards next node
            }
            break;
        case q2:
            ...
        }
    }
}
```

Figure 6.10: Transitions of the duplication automaton.
As input, the automaton receives the abstracted events decomposed as operations and arguments. All parallel derivations are confronted to these operations and progress according to their current state `q` and their semantic stack stored in `t_curids` and `t_curtypes`.

```
<?xml version="1.0"?>
<!DOCTYPE Behaviors [
   <!ELEMENT Behaviors (Duplication|Propagation|Residency|Overinfection|ExecutionProxy)*>
   <!ELEMENT Duplication (sequence,flow,source,target,transit?)>
   <!ELEMENT Propagation (sequence,flow,source,interface,transit?)>
   <!ELEMENT Residency (sequence,value,target)>
   <!ELEMENT Overinfection (sequence,conditional,marker)>
   <!ELEMENT ExecutionProxy (sequence,flow,source,target,transit?)>
   <!ELEMENT sequence EMPTY>
   <!ATTLIST sequence number ID #REQUIRED>
   <!ELEMENT flow EMPTY>
   <!ATTLIST flow method (transfer|single-block|interleaved) #REQUIRED>
   <!ELEMENT conditionnal EMPTY>
   <!ATTLIST conditionnal method (straight|inverse) #REQUIRED>
   <!ELEMENT source EMPTY>
   <!ATTLIST source id CDATA #REQUIRED>
   <!ATTLIST source name CDATA #REQUIRED>
   <!ATTLIST source nature (none|file|folder|drive|registry|network|mail) #REQUIRED>
   <!ELEMENT target EMPTY>
   <!ATTLIST target id CDATA #REQUIRED>
   <!ATTLIST target nature (none|file|folder|drive|registry|network|mail) #REQUIRED>
   <!ATTLIST target status (created|existing) #REQUIRED>
   <!ELEMENT interface EMPTY>
   <!ATTLIST interface id CDATA #REQUIRED>
   <!ATTLIST interface name CDATA #REQUIRED>
   <!ATTLIST interface nature (none|file|folder|drive|network|mail) #REQUIRED>
   <!ELEMENT transit EMPTY>
   <!ATTLIST transit id CDATA #REQUIRED>
   <!ATTLIST transit nature (none|variable) #REQUIRED>
   <!ELEMENT value EMPTY>
   <!ATTLIST value id CDATA #REQUIRED>
   <!ATTLIST value nature (none|file|folder|drive|registry|network|mail|variable) #REQUIRED>
   <!ELEMENT marker EMPTY>
   <!ATTLIST marker id CDATA #REQUIRED>
   <!ATTLIST marker name CDATA #REQUIRED>
   <!ATTLIST marker nature (none|file|folder|drive|registry) #REQUIRED>
]>
```

Figure 6.11: DTD of the Behavioral Report.
In addition to behaviors, the report stores information about the deployed method or the involved objects, these information being recovered respectively from the derivation and the object database.

crease its performance. A first mechanism avoids duplicate derivations. Coexisting iden-

tical derivations artificially increase the number of algorithm iterations without identifying other behaviors than the ones already detected. The second enhancement is related to the close and delete interactions. In order to decrease the number of iterations, useless derivations where no interaction occurs between the opening/creation and the closing/deletion of a same object are destroyed. These mechanisms have proved helpful in maintaining the number of parallel derivations at a manageable level.

6.3.4 Malware profiler

Above the detection automata, a malware profiler has been implemented in order to assess the profiles defined in Section 6.2. The behavioral reports generated by the automata contain the required information and are parsed using an open-source library for *XML parsing* called *Expat* [3]. According to the recovered information, the profiler associates the related malware to one or several generic classes. The profile report generated in output is also provided in an *XML* format satisfying the *Data Type Definition* presented in Figure 6.12.

```
<?xml version="1.0"?>
<!DOCTYPE Profile [
   <!ELEMENT Profile (Category)*>
   <!ELEMENT Category EMPTY>
   <!ATTLIST Category class CDATA #REQUIRED>
   <!ATTLIST Category subclass CDATA #IMPLIED>
]>
```

Figure 6.12: DTD of Profile Report.
The report can contain several entries since malware can satisfy the belonging conditions of different classes and subclasses.

6.4 Experimentation and discussions

Experimentations have been led to assess the prototype in operational conditions. For this, a pool of samples has been gathered, divided into two categories: *Portable Executables* and *Visual Basic Scripts*. Each category contains about 200 malware and 50 legitimate samples, split up in families according to the repartition from Figure 6.13. Malware have been mainly downloaded from repositories [7, 10], whereas legitimate samples have been selected from an healthy system installation, with a priority to samples whose behavior presents some similarities with malware. The different samples have been transmitted to their respective analyzers, before submitting the resulting abstracted logs to the detection automata. The resulting coverage is interpreted in Section 6.4.1 and observed phenomenons such as the collection impact, the behavior relevance or the profile

adequacy are respectively explained in further details in Sections 6.4.2, 6.4.3 and 6.4.4. Operational performance is finally addressed in Section 6.4.5.

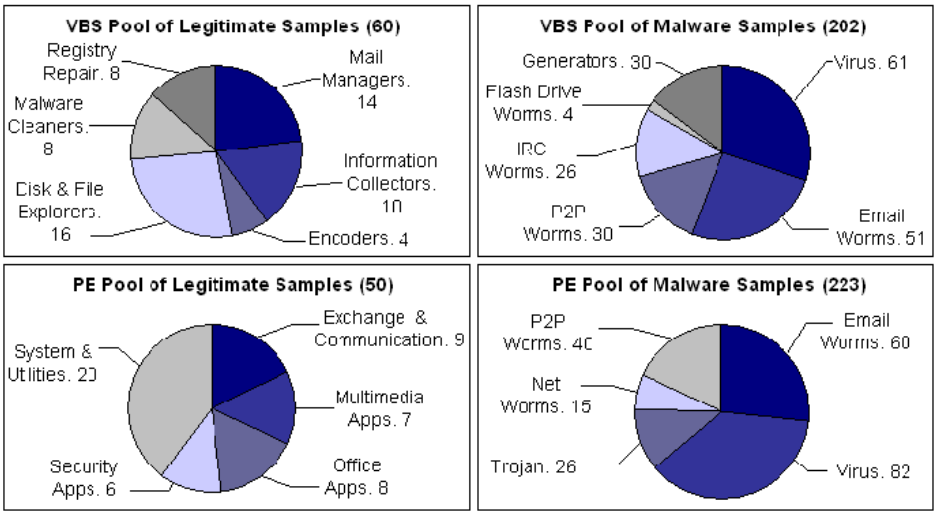

Figure 6.13: Repartition of the Test Pool.
The pool contains various types of malware among which are some of the most known: *Agobot*, *MySoom*, *Sober*, *Sobig*, etc. The pool also contains various samples whose behaviors show similarities with malware: *Outlook* for the mail activity, *Azureus* for file transmission, etc.

6.4.1 Coverage

The experimentation has provided significant results with a detection rate of 52% for *PE Executables* and up to 90% for *VB Scripts*. The detection rates by behaviors are described in Tables 6.2 and 6.3. Duplication is indeed the most significant malicious behavior. However the additional behaviors, and in particular residency, helps to detect additional malware where duplication is missed. False positives are almost inexistent, as shown in Tables 6.4 and 6.5. The only false positive, observed for residency, can be easily explained: the given script is a malware cleaner which reinitializes the *Internet Explorer* start page after infection.

Some false negative spikes, superior to 80%, can be localized in the PE results from Table 6.2: the low duplication detection rate for PE Viruses and the propagation detection rates for Net and Mail Worms are explained by limitations in the collection mechanisms. The impact of the collection mechanism on detection is assessed in Section

Behaviors	EmW	P2PW	V	NtW	Trj	Global
Duplication	41(68,33%)	31(77,5%)	15(18,29%)	8(53,33%)	10(38,46%)	47,09%
direct copy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
single read/write	41(68,33%)	30(75%)	14(17,07%)	8(53,33%)	10(38,46%)	46,19%
interleaved r/w	9(15%)	3(7,5%)	3(3,66%)	3(0,2%)	0(0%)	8,07%
Propagation	4(6,67%)	19(47,5%)	3(3,66%)	1(6,67%)	0(0%)	12,11%
direct copy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
single read/write	4(6,67%)	19(47,5%)	3(3,66%)	1(6,67%)	0(0%)	12,11%
interleaved r/w	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	36(60%)	22(55%)	5(60,98%)	6(40%)	12(46,15%)	36,32%
Overinfection test	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
conditional	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
inverse conditional	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Execution Proxy	0(0%)	0(0%)	0(0%)	0(0%)	4(15,38%)	1,79%
Global detection	43(71,67%)	33(82,50%)	16(19,51%)	8(53,33%)	16(61,54%)	52,02%

Table 6.2: PE Malware detection.

EmW = Email Worms, P2PW = Peer-to-Peer Worms, V = Viruses, NtW = Net Worms, Trj = Trojans.

6.4.2. Comparing *VB Scripts* and *PE Traces*, the false negative rates are lower for the scripts. The VBScript Analyzer works statically with path exploration; its coverage is thus more complete. The explanation of the remaining false negatives is twofold: the encryption of the whole malware body which is not supported yet and the cohabitation in a same web page of *JavaScript* and *VBScript* code which makes the syntactic analysis fail. Reversing code encryption can be handled similarly to string encryption, by localization of the decryption routine and calling it on-demand. Cohabitation of scripting languages can be addressed by a localization mechanism, parsing the tags of web pages to extract those containing *VBScript* code.

Globally, the observed detection rates for duplication are consistent with the results previously obtained in existing works [72]. The real enhancements from this work are twofolds: the parallel detection of additional behaviors described in the same language (propagation, residency and overinfection), and the possibility to feed detection with traces from other sources such as those coming from the script analyzer. With regards to [26], the execution proxy behavior has been transposed for testing the compliance with their model. The samples tested in common were mostly detected likewise; the exceptions are also explained by limitations in the collection mechanism.

Behaviors	EmW	FdW	IrcW	P2PW	V	Gen	Global
Encrypted strings	1/51	0/4	1/26	0/30	3/61	10/30	15/202
Encrypted body	4/51	0/4	0/26	1/30	2/61	0/30	7/202
String encryption	1(100%)	0	0	0(0%)	2(66,67%)	10(100%)	86,67%
Duplication	43(84,31%)	4(100%)	20(76,96%)	22(73,33%)	44(72,13%)	30(100%)	80,70%
direct copy	41(80,39%)	4(100%)	20(76,96%)	22(73,33%)	25(40,98%)	30(100%)	70,30%
single read/write	8(15,69%)	0(0%)	4(15,38%)	3(10%)	21(34,43%)	0(0%)	17,82%
interleaved r/w	1(1,96%)	0(0%)	0(0%)	0(0%)	8(13,11%)	0(0%)	4,46%
Propagation	33(64,71%)	3(75%)	5(19,23%)	25(83,33%)	5(8,20%)	30(100%)	49,99%
direct copy	33(64,71%)	3(75%)	4(15,38%)	25(83,33%)	3(4,92%)	30(100%)	48,52%
single read/write	3(5,88%)	0(0%)	2(7,69%)	1(3,33%)	2(3,28%)	0(0%)	3,96%
interleaved r/w	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	32(62,75%)	4(100%)	20(76,92%)	18(60,00%)	20(32,79%)	30(100%)	61,39%
Overinfection test	4(7,84%)	1(25%)	1(3,85%)	0(0%)	0(0%)	0(0%)	2,97%
conditional	4(7,84%)	1(25%)	1(3,85%)	0(0%)	0(0%)	0(0%)	2,97%
inverse conditional	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Execution proxy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Global detection	46(90,20%)	4(100%)	25(96,15%)	27(90,00%)	50(81,97%)	30(100%)	90,09%

Table 6.3: VBS Malware detection.

EmW = Email Worms, FdW = Flash Drive Worms, IrcW = IRC Worms,
P2PW = Peer-to-Peer Worms, V = Viruses, Gen = Generators variants.

Behaviors	PE ComE	PE MM	PE Off	PE Sec	PE SysU	PE Global
Duplication	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Propagation	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Overinfection test	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Execution proxy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Global detection	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%

Table 6.4: PE Legitimate Samples Detection.

Com = Communication and Exchange Applications, MM = Multimedia Applications,
Off = Office Applications, Sec = Security Tools, SysU = System and Utilities.

Behaviors	VBS EmM	VBS InfC	VBS Enc	VBS DfE	VBS MwC	VBS RegR	VBS Global
Duplication	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Propagation	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	0(0%)	0(0%)	0(0%)	0(0%)	1(12,50%)	0(0%)	1,67%
Overinfection test	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Execution proxy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Global detection	0(0%)	0(0%)	0(0%)	0(0%)	1(12,5%)	0(0%)	1,67%

Table 6.5: VBS Legitimate Samples Detection.

EmM = Email Managers, InfC = Information Collectors, Enc = Encoders, DfE = Disk and File Explorers, MwC = Malware Cleaners, RegR = Registry Repairs.

6.4.2 Limitations in trace collection

A significant part of the missed behaviors, or false negatives, are due to limitations existing in the collection coverage. However, thanks to the layer-based approach, collection and abstraction can be improved for a given platform or language without modifying the upper detection layer.

1) Dynamic analysis (PE Traces): Due to the dynamic nature of the collection, the first reason for detection failure is a problem related to the configuration of the simulated environment. The simulation must appear as real as possible in order to satisfy the execution conditions of the malware, in particular for triggered actions. The software configuration of the simulated environment constitutes a first difficulty. 64,6% of the tested PE Viruses (53/82) did not execute properly in the simulated environment: invalid PE files, access violations or unhandled exceptions. These failures may be explained by the detection of virtualization or anti-debug techniques crafted to thwart dynamic analysis.

The configuration of the simulated network constitutes a second problem. For example, the propagation of Mail Worms is conditioned by the network configuration. 75% of the PE Mail Worms (45/60) did not show any SMTP activity because they could not reach any server. In certain worms, the address of the server was hard coded making the redirection by the DNS server useless. Certain worms were also unable to retrieve from the environment the address of a registered mail server. Likewise, Net Worms propagate through vulnerabilities only if a vulnerable target is reachable. The absence of potential targets explains that 93,33% of them did not propagate (14/15). They contented themselves with scanning different ranges of IPs. The problem is even worse with the bot samples from the Trojan pool. In order to observe the different behaviors, the bots must receive the right commands through an IRC channel which is often protected by a password. In order to configure this password as well as the URL of the reachable IRC

server, six bots were produced from customized code sources, recompiled specifically for the collect platform [9]. 66% of the behaviors of execution proxy were detected in these bots (4/6). On the opposite, for the other bots whose binary only was available (3/3), only duplication was observed because no command was sent. Generally speaking, all actions conditioned by the configuration of the simulated environment are difficult to observe: a potential solution could be forced branching.

Beyond the configuration problem, the level of the trace collection can also explain the detection failure. With a high-level collection mechanism, like *NtTrace* running in user space, visibility over the performed actions and the data flow is reduced. All flow-sensitive behaviors such as duplication can be missed because of breakdowns in this data flow. Such breakdowns can find their origin sometimes in non monitored system calls and for the most part in the intervention of intermediate buffers where all operations are executed in memory. These buffers are often used in code mutation (polymorphism, metamorphism). 12,20% of the viruses duplications (10/82) were missed because of a data flow breakdown. The problem is identical with mail propagation: 8,33% of the propagations (5/60) were missed for Mail Worms because of an intermediate buffer used for *Base64 encoding*. These problems do not come from the behavioral descriptions but from *NtTrace* which does not capture any information about operations in memory. More complete collection tools, either collecting instructions [29] or deploying tainting techniques [1, 85], could avoid these breakdowns in the data flow. Tainting, in particular, uses a shadow memory to store taint information about the sensitive data manipulated. Taints are then propagated at the instruction level whenever the result of the computation depends on data already tainted.

2) Static analysis (VB Scripts): In the VBScript Analyzer, the static analysis of the source code enables branching exploration and observation of the data flow. Their implementation compensates for the drawbacks that were encountered with *NtTrace*. The greater coverage of the Analyzer eventually results in better detection rates.

However, contrary to the stable set of system calls, the *VBS* language offers numerous services to monitor. The same operation can be achieved using different managers or interfacing with different *Microsoft* applications. The actual version of the analyzer should monitor additional features to increase its coverage: accesses to *Messenger* services or the support of the *Windows Management Instrumentation (WMI)*. For example, listing connected drives for propagation is currently supported by the analyzer but this same list could be recovered using *WMI* by querying the `LogicalDisk` entries from the `Win32_ComputerSystem` object. The support of the WMI is required to detect Drive Worms using this technique.

Moreover, like any other static analysis, script analysis is hindered by encryption and obfuscation techniques. Generally speaking, static analysis of scripts is easier because no

prior disassembly is required and some security locks ease the analysis: no dynamic code rewriting, no dynamically resolved jumps. However, inserting an intermediate interpretation layer can reintroduce all obfuscation techniques possible in low level languages (C language, Assembly) [69].

6.4.3 Behavior relevance

The previous section deals with problems related to data collection, but the behavioral model itself must be assessed. The relevance of each behavior must be individually assessed by checking the coverage of its grammatical model. It then becomes possible to extrapolate possible correlations between the different behaviors, by attaching a greater weight to the most relevant behaviors.

Duplication, propagation and residency are obviously characteristic to malware. However, only duplication and propagation are discriminating enough for detection. On the contrary, residency has exhibited false positives during the experimentations. Its behavioral model could be refined by introducing a constraint on the value written to the booting object: the value should refer to the program itself or to one of its duplicated versions. This modification could help avoiding the observed false positives. Anyhow, residency is still likely to occur in legitimate cases, during installation of programs or drivers. For example, antivirus products use the same hooking techniques to monitor system calls than malware use for stealth. False positives can also be found for the behavior of execution proxy, even if it is not observed in the tested legitimate samples. Obviously, remote installers deploy the exact same technique; and this is confirmed in [26]. Consequently, bivalent behaviors, used both by legitimate and malicious programs, can not really be considered as false positives. Their behavioral model can be maintained; the distinction of malicious intents must eventually be addressed by correlation with other behaviors, purely malicious. For example, the profiler correlates the behavior of execution proxy with duplication to detect Trojans.

On the other hand, the behavioral model for overinfection tests is not completely relevant. The weak detection rates are explained by a description that is overly specific. The conditional structure on which the behavioral model is built constitute a first restriction because it is not captured by dynamic monitoring. The collected traces of system call do not contain information about conditional jumps and their alternative paths. In addition, stopping is always triggered in case of overinfection, which is not always true. A benign behavior could be deployed instead. A potential solution to these restrictions could be a generalization of the model. For example, the conditional could be removed and replaced by consecutive open and create commands. However, it would increase the risk of confusion with error handling in legitimate programs. Maintaining this behavior

may finally be arguable.

6.4.4 Profiles adequacy

To study the adequacy of our profiles, the experimentations have been pursued by submitting the output of the detection automata to the profiler. In addition, this study is also a mean to measure the impact of the individual behaviors on classification. Obviously, classifying legitimate programs into malware families shows little interest. Legitimate results are thus put aside. Similarly, correlation when no behavior is detected makes little sense. These results are also removed from the study. The profiler results are finally presented in confusion matrices where they are compared with their original malware family. Since they may be errors in the repositories from which samples were downloaded, a reclassification has been manually realized before the comparison.

	Drive Worm	Email Worm	Irc Worm	P2P Worm	Virus
DWG	1/4(25,00%)	2/77(02,60%)	1/29(03,45%)	3/27(11,11%)	4/44(09,09%)
DWA	3/4(75,00%)	(00,00%)	(00,00%)	(00,00%)	(00,00%)
EMW	(00,00%)	42/77(54,54%)	1/29(03,45%)	(00,00%)	(00,00%)
EMW+DWG	(00,00%)	13/77(16,88%)	(00,00%)	1/27(03,70%)	(00,00%)
EMW+IRW	(00,00%)	1/77(01,30%)	(00,00%)	(00,00%)	(00,00%)
EMW+DWG+VFI	(00,00%)	1/77(01,30%)	(00,00%)	(00,00%)	(00,00%)
EMW+IRW+PPW	(00,00%)	1/77(01,30%)	(00,00%)	(00,00%)	(00,00%)
EMW+IRW+VFI	(00,00%)	1/77(01,30%)	(00,00%)	(00,00%)	(00,00%)
EMW+IRW+PPW+VFI	(00,00%)	1/77(01,30%)	(00,00%)	(00,00%)	(00,00%)
IRW	(00,00%)	(00,00%)	12/29(41,38%)	(00,00%)	(00,00%)
IRW+DWG	(00,00%)	(00,00%)	2/29(06,89%)	(00,00%)	(00,00%)
IRW+PPW	(00,00%)	1/77(01,30%)	1/29(03,45%)	(00,00%)	(00,00%)
IRW+DWG+PPW	(00,00%)	1/77(01,30%)	(00,00%)	(00,00%)	(00,00%)
PPW	(00,00%)	(00,00%)	(00,00%)	15/27(55,56%)	(00,00%)
PPW+DWG	(00,00%)	(00,00%)	(00,00%)	1/27(03,70%)	(00,00%)
PPW+IRW	(00,00%)	(00,00%)	(00,00%)	1/27(03,70%)	(00,00%)
VFI	(00,00%)	(00,00%)	(00,00%)	(00,00%)	8/44(18,18%)
VFO	(00,00%)	(00,00%)	(00,00%)	1/27(03,70%)	(00,00%)
GM	(00,00%)	13/77(16,88%)	12/29(41,38%)	5/27(18,53%)	32/44(72,73%)

Table 6.6: VBS Malware classification.

This confusion matrix is built with the columns indexed with the real malware classes and the lines indexed by the output of the profiler. The generic malware correspond to samples with no attributed class. Labels: DWG = DriveWorm (generic), DWA = DriveWorm (amovible), EMW = Email Worm, IRW = Irc Worm, PPW = Peer-to-Peer Worm, VFI = Virus (file infector), VFO = Virus (file overwriter), GM = Generic Malware.

The best results of coverage were obtained with the *VBS Scripts* samples, consequently

the classification of the malware is likely to be more precise. The results obtained with the profiler are synthesized in the confusion matrix from Table 6.6. The matrix takes into consideration the fact that a given malware instance can simultaneously satisfies several profiles. Globally, the results are quite satisfying with an accuracy of 70% on average, except for viruses where it drops to 18%. The problem is that viruses in *VBS* are not really viruses in the sense of programs infecting a host application, but simply duplicating programs.

Part of the remaining confusions are mainly due to the fact that some duplications were missed. For example, some Mail Worms and Peer-to-Peer Worms were classified as generic malware in spite of their propagation; only because they did not duplicate as required by their profiles. Similarly, residency was found in almost all Irc Worms; but only 51% were correctly classified because no duplication nor propagation was detected. However, since residency is the behavior the most prone to false positives; residency alone can not be sufficient to define a profile for Irc Worms.

	Email Worm	Net Worm	P2P Worm	Trojan	Virus
EMW	(00,00%)	(00,00%)	(00,00%)	(00,00%)	(00,00%)
NW	(00,00%)	1/8(12,50%)	(00,00%)	(00,00%)	(00,00%)
NW+VFI	2/43(04,65%)	(00,00%)	(00,00%)	(00,00%)	(00,00%)
NW+PPW+VFI	1/43(02,33%)	(00,00%)	(00,00%)	(00,00%)	(00,00%)
PPW	2/43(04,65%)	(00,00%)	18/34(52,94%)	(00,00%)	(00,00%)
T	(00,00%)	(00,00%)	(00,00%)	4/16(25,00%)	(00,00%)
VFI	7/43(16,28%)	2/8(25,00%)	(00,00%)	(00,00%)	2/15(13,33%)
VFI+PPW	(00,00%)	(00,00%)	(00,00%)	(00,00%)	1/15(06,67%)
VFO	(00,00%)	(00,00%)	(00,00%)	(00,00%)	(00,00%)
GM	31/43(72,09%)	5/8(62,50%)	16/34(47,06%)	8/16(75,00%)	12/15(80,00%)

Table 6.7: PE Malware classification.

This confusion matrix is built with the columns indexed with the real malware classes and the lines indexed by the output of the profiler. The generic malware correspond to samples with no attributed class. Labels: EMW = Email Worm, NW = Network Worm, PPW = Peer-to-Peer Worm, T = Trojan, VFI = Virus (infector), VFO = Virus (overwriter), GM = Generic Malware.

The results are less precise for *PE Executables*, as shown by the confusion matrix in Table 6.7. This loss of precision is mainly explained by the missed behaviors. In particular, an important number of propagations were missed, explaining significant confusions in the classification of the different Worms. An other important remark on propagation is that no precise information about the network communications, such as the port or the protocol, was available inside the traces of system calls. Consequently, no distinction could be done between Net Worms and Mail Worms. The accuracy of the Trojan classification is also low. We have only considered for detection the behavior of

execution proxy, whereas the Trojans can also offer other services such as Spam relay or stealth techniques. The Trojan profile is thus incomplete and would require additional behavioral signatures for these services.

6.4.5 Performance

Table 6.8 provides the measured performance for the different components of the prototype. Starting with the abstraction layer, the analysis of *PE Traces* is the most time consuming task. This is not surprising since the analyzer uses numerous string comparisons which could be partially avoided by replacing the off-line analysis by real-time collection and translation. By hooking the system calls, the translation becomes immediate. As for the VBScript Analyzer, it offers satisfying performances. Optimized, it could be deployed on mail servers to analyze joint pieces for example.

NtTrace Analyzer	Data reduction from PE traces to logs	
	Total size: 351,32Mo Reduced logs: 11,85Mo	Average: 1,32Mo/Trace Reduction ratio: 29
	Execution speed	
	Single core M 1,4GHz	Dual core 2,6GHz
	1,48 s/trace	0,34 s/trace
VB Script Analyzer	Data reduction from VB scripts to logs	
	Total size: 1842Ko Reduced logs: 298Ko	Average: 7Ko/Script Reduction ratio: 6
	Execution speed	
	Single core M 1,4GHz	Dual core 2,6GHz
	0,042 s/script +0,50 s/encrypted line	0,016 s/script +0,21 s/encrypted line
Detection Automata	Execution speed	
	Single core M 1,4GHz	Dual core 2,6GHz
	NT: 0,44 s/log VBS: 0,002 s/log	NT: 0,14 s/log VBS: <0,001 s/log

Table 6.8: Prototype performances.

The time and space performances are described components by components for mono-core and multi-core configurations.

The performance of the detection automata are also satisfying compared with the worst case complexity found in Proposition 1. The detection speed remains far below the order of a half second in more than 90% of the cases; the remaining 10% were all malware. In real-time conditions, it would correspond to a charge of 50.000 system call-s/second. The prototype implementation has also revealed that the maximum required space for the derivation stacks was very low: 7 and 3 elements are the respective max-

imal sizes reached by the syntactic and semantic stacks ($2s < 10$ in Proposition 1). In addition to speed, the number of raised ambiguities has also been measured leading to the establishment of an operational complexity stated in Proposition 2.

Proposition 2 *In the average case, behavioral detection using attributed automata has an operational time complexity in* $\vartheta(k\alpha(\frac{n^2+n}{2}))$ *and space complexity in* $\vartheta(k\alpha n(2s))$, *where* k *is the number of automata,* n *is the number of input symbol and* α *the ambiguity ratio.*

Proof 2 *If* n_e *denotes the number of events and* n_a *the number of ambiguity, in the worst case, we would have* $n_a = 2^{n_e}$. *By experience, we obtain:*

$n_a << 2^{n_e}$ *and* $n_a << n_e^2$ *and* $n_a \approx \alpha n_e$

Let us consider a regular distribution of these ambiguities, meaning that α *derivations are started at each iteration.*

(1) $k\alpha + 2k\alpha + ... + nk\alpha = k\alpha(1 + 2 + ... + n) = k\alpha(\frac{n^2+n}{2})$

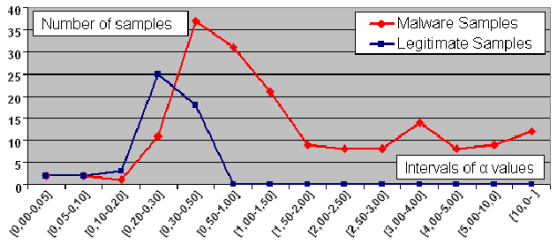

Figure 6.14: Ambiguity ratios (α) for the PE samples.

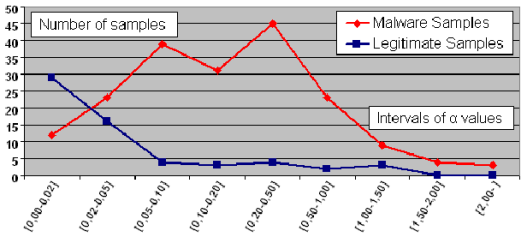

Figure 6.15: Ambiguity ratios (α) for the VB scripts.

The approximation of Proposition 2 provides an operational complexity more worth considering. Moreover, this algorithm can easily be parallelized for optimization in a multi-core architecture. Figures 6.14 and 6.15 provide graphs of the collected α ratios. **From these graphs, it can be observed that above a certain threshold, an important ambiguity ratio** α **is already a sign of malicious activity**.

7 Exploit Behaviour and Shellcode Analysis

The Argos emulator [85] is used by various projects to detect zero-day attacks. Some advanced versions of Argos, like Prospector [92], go beyond basic detection by identifying the bytes that are responsible for buffer overflows and matching these bytes with protocol fields of network traffic to serve as signatures. However, even the most advanced version of Argos to date stops at the shellcode. In fact, we have carefully tried to avoid executing even a single instruction of the attacker's code.

On the one hand, this design decision has helped deployment. Administrators are less reluctant to host a complex piece of intrusion detection software if they know that malicious code will not be executed. As such, it has been one of the key selling points of Argos.

On the other hand, stopping at the very first instruction limits our options for further analysis. It is hard to determine what the shellcode does, which server it tries to contact, and which binary it downloads, unless we execute part of the shellcode.

We recently extended the Argos emulator to make it execute and analyze the shellcode. Much of this analysis concerns the *structural* features of the attacker's code. For instance, the new version determines how many and what sort of unpackers are used, where the real shell code starts, etc.

However, the execution and analysis of shellcode also enables behavioral analysis. For instance, we can see what Windows API calls the shell code makes, and in some cases even which malware it downloads. We will report the structural analysis in Deliverable 4.4 (Final Analysis report of structural features), while in this deliverable we will limit ourselves to the behavioural aspects.

7.1 Analysis of shellcode behavior

The Argos emulator currently monitors the shellcode while it executes. Unpackers are tracked and the final shellcode is carefully instrumented. For later analysis, we record and disassemble every instruction that is executed. Doing so gives us a reliable trace of the attacker's code in its early stages.

Moreover, we explicitly track all calls to imported Windows libraries. First, we log the calls and map the target addresses back to high-level function descriptions (by means

of lookups in symbol tables). As a result, we know what functions are called by the attacker's code (e.g., `connect`, or `createFile`) with the arguments.

The shellcode keeps executing until it reaches a stop condition. A stop condition is specified by the Argos administrator in terms of API calls. A trivial stop condition is to stop executing when the first system call is made. However, more interesting stop conditions can also be specified. One fairly safe one would be to limit the number of outgoing connections and stop at the first write to the network. Alternatively, we can relax the stop condition even more and stop, for instance, at the second outgoing network connection. This would allow the shellcode to download the malware from a server, while limiting possible harm that may be caused by the shellcode to other machines on the Internet.

7.2 Advanced use of shellcode analysis in WOMBAT

As new attacks are more interesting than old ones, we would like to handle known attacks automatically. Suppose an attack exchanges a sequence of messages with a host which eventually compromises it and makes it execute shellcode that will start downloading the real malware (e.g., the bot software). If the initial exchange of messages is known, we do not want to spend time analysing it. Nor do we want to waste the scarce cycles of high-interaction honeypots on it. Rather, we want to handle the attack handshake automatically in a light-weight front-end that has learned how to respond to the attacker's overtures, and rather than execute the shellcode, simply executes the appropriate download command to obtain the malware.

7.2.1 Extending SGNet

In WOMBAT, we use SGNet sensors to handle the interaction of known attacks automatically [65]. Attacks that cannot be handled by the sensors are forwarded to an Argos node [85] which serves as an oracle for separating attacks from benign interaction. So far, however, there has not been an automatic way to analyse the shellcode to find out what it tries to download and then to download it in a safe manner. Instead, SGNet has relied on static handlers (pre-)defined by the Nepenthes project. A handful of static handlers does not scale if WOMBAT aims to cover many nodes with many applications with many different vulnerabilities.The implication is that we should perform automatic shellcode analys is to find out what the code wants to download.

7.2.2 Joining forces: automatic analysis using Argos, SGNet, Nemu and Anubis

A brief analysis of the above goals reveals that within the WOMBAT project, various pieces of the overall puzzle are available. For instance:

1. SGNet [65] is capable of inferring the right handshake with attackers given a set of samples.
2. Argos [85] is good at detecting exploits and linking bytes in memory to bytes in network traffic. With the extension we are also able to analyse the shellcode.
3. Nemu [84] is a network emulator that is explicitly designed to detect non-selfcontained polymorphic shellcode.
4. Anubis [20] uses dynamic analysis to cluster malware in different families, helping us focus on the most interesting ones.

As a logical next step, we have decided to combine these subprojects into a single system as follows. The majority of known attacks will be handled by SGNet sensors. A sensor handshakes with the attacker. Let us assume for now that the attack is a new one that cannot be handled by the sensor. In that case, the sensor forwards the attack to the Argos oracle at the SGNet core by replaying the interaction. So far, this behaviour is similar to that of existing SGNet sensors. We will see shortly that the new SGNet sensors behave differently from their predecessors in case of known attacks.

Argos not only serves as the oracle that decides whether or not a specific interaction is malicious. It also executes the shellcode and indicates where we can find these bytes in the network stream. This information is then given to Nemu which uses the starting location in the network trace to start detecting the shellcode. Nemu also identifies the malware that should be downloaded. Next, the file is downloaded.

To help integrating the new version of Argos with the existing infrastructure, we have added a new data channel to the emulator by means of a Unix domain socket. The socket operates as a tap interface to Argos. In other words, SGNet can simply inject Ethernet frames into the socket, which will then be read by Argos as if these packets arrived on the network. The socket should make it very simple for SGNet sensors to replay new/unknown interactions against Argos.

The role of Nemu may not be immediately clear. After, all the identification of the shellcode can also be handled by Argos. This is true in principle, but Nemu is much more lightweight. In the absence of static handlers that determine how we should download the malware for a specific attack, we need to dig out the shellcode dynamically to find out what is downloaded from where at the SGNet sensors. Installing Argos at every sensor scales poorly, but we should be able to run Nemu to analyse the shellcode, given that Argos has already told it what to look for.

In short, an SGNet that is able to handle the interaction with the attacker, will invoke Nemu-based analysis on the shellcode. The result will be a clear procedure to download the malware.

Finally, we use Anubis to analyse the malware and place it in a cluster. Doing so allows us to zoom in on new or otherwise interesting families of malicious code.

While the design is in place, the implementation of this more advanced use of shellcode analysis is still ongoing.

8 Conclusion

It is clear from the preceding chapters that the behavioral analysis of malicious code has been an extremely active research area within the Wombat consortium. Many consortium members were involved in this work package (and indeed this deliverable) and they have used a wide variety of techniques to perform the analysis. Most importantly, the consortium has clearly pushed the state of the art in analysis of malware behaviour in many aspects.

Several of the results presented in this deliverable have been presented at peer-reviewed venues. Others - like the shellcode analysis - have led to closer collaboration between the Wombat partners.

In summary, we conclude that the work on behavioral features within the Wombat project has been both rich in variety and successful in their results. Moreover, the techniques described here will provide invaluable support to our work on threat analysis.

Bibliography

[1] Anubis: Analyzing unknown malware.

[2] Darpa intrusion detection evaluation datasets. Available online at `http://www.ll.mit.edu/IST/ideval/data/dataindex.html`.

[3] ¡eXpat/¿ - The Expat XML Parser.

[4] MAFIA: Metasploit anti forensics investigation arsenal. Available online at `http://metasploit.com/projects/antiforensics/`.

[5] Msdn - vbscript language reference.

[6] Nttrace - native api tracing for windows.

[7] Offensive computing repository.

[8] Qemu - open source processor emulator.

[9] Securitydot : exploits, vulnerabilities, articles.

[10] Vx heaven repository.

[11] Advanced antiforensics – SELF. Available online at `http://www.phrack.org/issues.html?issue=63&id=11`, 2005.

[12] Anubis. `http://anubis.seclab.tuwien.ac.at`, 2008.

[13] AVG Virus Database - Mabezat. `http://www.avg.com/virbase?nam=win32/mabezat`, 2008.

[14] F-Secure Malware Information Pages - Allaple.A. `http://www.f-secure.com/v-descs/allaple_a.shtml`, 2008.

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

[16] S. O. Al-Mamory and H. Zhang. IDS Alerts Correlation using Grammar-based Approach. *Journal in Computer Virology*, Published online, 2008.

[17] M. Bailey. Malware clustering results. `http://www.eecs.umich.edu/~mibailey/malware/`, 2008.

[18] M. Bailey, J. Oberheide, J. Andersen, Z. Mao, F. Jahanian, and J. Nazario. Automated Classification and Analysis of Internet Malware. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[19] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, September 2007.

[20] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda. Scalable, behavior-based malware clustering. 2009.

[21] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, April 2006.

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

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

[24] H. Berghel. Hiding data, forensics, and anti-forensics. *Commun. ACM*, 50(4):15–20, 2007.

[25] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *Proc. IEEE Symposium Security and Privacy (SSP)*, pages 48–62. IEEE Computer Society, 2006.

[26] D. Bruschi, L. Martignoni, and M. Monga. Detecting Self-Mutating Malware using Control-Flow Graph Matching. In *Proceedings of the Conference on the Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Lecture Notes in Computer Science, pages 129–143, 2006.

[27] M. Burdach. In-memory forensics tools. Available online at `http://forensic.seccure.net/`.

[28] J. B. D. Cabrera, L. Lewis, and R. Mehara. Detection and classification of intrusion and faults using sequences of system calls. *ACM SIGMOD Record*, 30(4), 2001.

[29] E. Carrera. Malware - behavior, tools, scripting and advanced analysis. In *HITB-SecConf*, 2008.

[30] G. Casas-Garriga, P. Díaz, and J. Balcázar. ISSA: An integrated system for sequence analysis. Technical Report DELIS-TR-0103, Universitat Paderborn, 2005.

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

[32] B. L. Charlier, A. Mounji, and M. Swimmer. Dynamic detection and classification of computer viruses using general behaviour patterns. In *Proc. Conf. Virus Bulletin*, 1995.

[33] M. Christodorescu, S.Jha, and C. Kruegel. Mining specifications of malicious behaviour. In *Proc. joint meeting Conf. European Software Engineering and ACM SIGSOFT Symp. Foundations of Software Engineeering*, pages 5–14, 2007.

[34] R. Cilibrasi and P. Vitányi. Complearn version 1.15. `http://www.complearn.org/`, 2008.

[35] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. Version Control with Subversion. `http://svnbook.red-bean.com/en/1.5/svn-book.html`, 2008.

[36] Core Security Technologies. CORE Impact. `http://www.coresecurity.com/?module=ContentMod&action=item&id=32`.

[37] A. Cozzie, F. Stratton, H. Xue, and S. King. Digging For Data Structures . In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[38] F. Cuppens and A. Miège. Alert correlation in a cooperative intrusion detection framework. In *Proc. IEEE Symp. Security and Privacy (SSP)*, page 202. IEEE Computer Society, 2002.

[39] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection . In *IEEE Symposium on Security and Privacy*, 2004.

[40] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information . In *IEEE Symposium on Security and Privacy*, 2003.

[41] P. Ferrie. Attacks on more virtual machine emulators.

[42] E. Filiol. Malware Pattern Scanning Schemes Secure against Black-Box Analysis. *Journal in Computer Virology*, 2(1, EICAR 2006 Special Issue, V. Broucek and P. Turner Eds.):35–50, 2006.

[43] E. Filiol, G. Jacob, and M. L. Liard. Evaluation Methodology and Theoretical Model for Antiviral Behavioural Detection Strategies. *Journal in Computer Virology*, 3(1, WTCV'06 Special Issue, G. Bonfante and J-Y. Marion Eds.):23–37, 2007.

[44] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 120, Washington, DC, USA, 1996. IEEE Computer Society.

[45] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 1996. IEEE Computer Society.

[46] J. Foster and V. Liu. Catch me if you can.... In *Blackhat Briefings 2005*, Las Vegas, NV, August 2005.

[47] S. Garfinkel. Anti-Forensics: Techniques, Detection and Countermeasures. In *Proceedings of the 2nd International Conference on i-Warfare and Security (ICIW)*, pages 8–9, 2007.

[48] S. Garfinkel and A. Shelat. Remembrance of data passed: a study of disk sanitization practices. *Security & Privacy Magazine, IEEE*, 1(1):17–27, 2003.

[49] M. Geiger. Evaluating Commercial Counter-Forensic Tools. In *Proceedings of the 5th Annual Digital Forensic Research Workshop*.

[50] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *11th Annual Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, February 2004.

[51] Grugq. The art of defiling: defeating forensic analysis. In *Blackhat briefings 2005*, Las Vegas, NV, August 2005.

[52] D. Grune and C. Jacobs. *Parsing Techniques - A Practical Guide*. Springer, 2008.

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

[54] J. Han and M. Kamber. *Data Mining: concepts and techniques*. Morgan-Kauffman, 2000.

[55] R. Harris. Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. In *Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06)*, volume 3 of *Digital Investigation*, pages 44–49, September 2006.

[56] S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[57] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages and Computation, 2nd ed.* Addison Wesley, 1995.

[58] G. Jacob, H. Debar, and E. Filiol. Malware Behavioral Detection by Attribute-Automata using Abstraction from Platform and Language. In *Proc. Int. Symp. Recent Advances in Intrusion Detection (RAID)*, LNCS, pages 81–100. Springer, 2009.

[59] S. Jha, K. Tan, and R. A. Maxion. Markov chains, classifiers, and intrusion detection. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations (CSFW'01)*, pages 206–219, Washington, DC, USA, June 2001. IEEE Computer Society.

[60] N. Johnson and S. Jajodia. Exploring steganography: Seeing the unseen. *COMPUTER*, 31(2):26–34, 1998.

[61] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. *IEEE Softw.*, 14(5):35–42, 1997.

[62] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating Mimicry Attacks Using Static Binary Analysis. In *14th Usenix Security Symposium*, 2005.

[63] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In *Proceedings of the 2003 European Symposium on Research in Computer Security*, Gjvik, Norway, October 2003.

[64] T. Lee and J. J. Mody. Behavioral Classification. In *EICAR Conference*, 2006.

[65] C. Leita, M. Dacier, and F. Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with scriptgen based honeypots. In *Proceedings of RAID'06*, pages 185–205, Hamburg, Germany, September 2006.

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

[67] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. Submitted for publication, 2007.

[68] F. Maggi, S. Zanero, and V. Iozzo. Seeing the invisible: forensic uses of anomaly detection and machine learning. *SIGOPS Oper. Syst. Rev.*, 42(3):51–58, 2008.

[69] J.-Y. Marion and D. Reynaud-Plantey. Practical obfuscation by interpretation. In *Presented at 3rd Workshop on the Theory of Computer Viruses (WTCV)*, 2008.

[70] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *Proc. Int. Symp. Recent Advances in Intrusion Detection (RAID)*, LNCS, pages 78–97. Springer, 2008.

[71] Microsoft PECOFF. Microsoft Portable Executable and Common Object File Format Specification. `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`, 2000.

[72] J. A. Morales, P. J. Clarke, and Y. Deng. Identification of file infecting viruses through detection of self-reference replication. *Journal in Computer Virology*, Online, 2008.

[73] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, 2006.

[74] National Vulnerability Database. CVE-2007-1719. Available online at `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-1719`.

[75] National Vulnerability Database. CVE-2007-3641. Available online at `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-3641`.

[76] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *IEEE Symposium on Security and Privacy*, 2005.

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

[78] J. Nick L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197–210, december 2006.

[79] P. Ning, Y. Cui, D. S. Reeves, and D. Xu. Techniques and tools for analyzing intrusion alerts. *ACM Trans. Information and System Security*, 7(2):274–318, 2004.

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

[81] D. Ourston, S. Matzner, W. Stump, and B. Hopkins. Applications of Hidden Markov Models to detecting multi-stage network attacks. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, page 334, 2003.

[82] T. Parr. ANTLR Parser Generator.

[83] S. Piper, M. Davis, G. Manes, and S. Shenoi. *Detecting Hidden Data in Ext2/Ext3 File Systems*, volume 194 of *IFIP International Federation for Information Processing*, chapter 20, pages 245–256. Springer, Boston, 2006.

[84] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Real-world polymorphic attack detection using network-level emulation. In *CSIIRW '08: Proceedings of the 4th annual workshop on Cyber security and information intelligence research*, pages 1–3, New York, NY, USA, 2008. ACM.

[85] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[86] J.-P. Pouzol and M. Ducassé. From Declarative Signatures to Misuse IDS. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2001.

[87] J.-P. Pouzol and M. Ducassé. Formal Specification of Intrusion Signatures and Detection Rules. In *Proceedings of the IEEE Computer Security Foudations Workshop (CSF)*, 2002.

[88] S. Ring and E. Cole. Volatile Memory Computer Forensics to Detect Kernel Level Compromise. In *Proceedings of the 6th International Conference on Information And Communications Security (ICICS 2004)*, Malaga, Spain, October 2004. Springer.

[89] B. Schatz. Bodysnatcher: Towards reliable volatile memory acquisition by software. In *Proceedings of the 7th Annual Digital Forensic Research Workshop (DFRWS '07)*, volume 4 of *Digital Investigation*, pages 126–134, September 2007.

[90] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2001. IEEE Computer Society.

[91] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[92] A. Slowinska and H. Bos. The age of data: pinpointing guilty bytes in polymorphic buffer overflows on heap or stack. In *23rd Annual Computer Security Applications Conference (ACSAC'07)*, Miami, FLA, December 2007.

[93] W. Sun, Z. Liang, V. Venkatakrishnan, and R. Sekar. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Network and Distributed Systems Symposium (NDSS)*, 2005.

[94] D. Wagner and D. Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.

[95] R. Watson. OpenBSM. `http://www.openbsm.org`, 2006.

[96] M. Weiser. Program Slicing. In *International Conference on Software Engineering (ICSE)*, 1981.

[97] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.

[98] Wombat Partners. D08 (d4.1) specification language for code behavior. Technical report, Wombat European Project from the 7th FP, 2008.

[99] S. Zanero. *Unsupervised Learning Algorithms for Intrusion Detection*. PhD thesis, Politecnico di Milano T.U., Milano, Italy, May 2006.