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

WORLDWIDE OBSERVATORY OF
MALICIOUS BEHAVIORS AND ATTACK THREATS

D21 (D4.7) Consolidated report with evaluation results

Contract No. FP7-ICT-216026-WOMBAT

Workpackage	WP4 - Data Enrichment and Characterization
Author	Paolo Milani Comparetti
Version	1.0
Date of delivery	M40
Actual Date of Delivery	M40
Dissemination level	Public
Responsible	TUV

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 is the final deliverable for Workpackage 4 within the WOMBAT project. In this document we discuss the final extensions and improvements to our data collection and analysis techniques that were implemented as part of WOMBAT. Furthermore, we present some additional results obtained from the analysis of data collected within WOMBAT.

1 Introduction (TUV)

This deliverable comes at the conclusion of WOMBAT Work Package 4 (WP4), Data Enrichment and Characterization, which involved a significant effort from WOMBAT partners over a period of 34 months. Research performed within WP4 has been presented in six deliverables (D4.1-D4.6) and published in leading systems security venues (e.g., RAID, NDSS). Furthermore, it has led to the development and deployment of a number of techniques for detecting and analyzing malicious code, attacks and other internet threats by means of their behavioral, structural and contextual characteristics. Although these analysis systems are distinct and are operated by different WOMBAT partners, it is important to underline that they are involved in—and interconnected together by—a complex network of relationships. In such network, both raw data and results of the analysis are shared and processed by different systems that, in many cases, rely on the WOMBAT API (WAPI) for communication.

In this deliverable, we will not provide an overview of all of the developed techniques, because this would be a repetition of the content of previous deliverables. Nor will we present evaluation results for all of our analysis techniques, because, in most cases, these have also already been presented in deliverables D16 (D4.2) and D17 (D4.4). Clearly, this deliverable includes some additional results, yet our focus is on further research that has been performed—within the scope of WP4—largely as a consequence of the WOMBAT collaboration and, specifically, as a consequence of the "feedback loop" between the various WPs. The reason is that in the later phases of the WOMBAT project, research within WP4 has been informed by the effort to understand the root causes of internet threats and to develop early warning systems that was being carried out within WP5.

1.1 Contents

The first two chapters (Chapter 2 and 3) of this deliverable overview research aimed at improving the coverage of our data collection and analysis efforts. The goal of such research was to address "gaps" in our view of the threat landscape that were identified during the course of WOMBAT. Both chapters also present extensive evaluation results obtained from large-scale deployments of the proposed techniques. More precisely, HoneyBuddy, described in Chapter 2, is a honeypot for instant messaging (chat) protocols.

HoneyBuddy relies on dummy accounts that perform a role similar to SPAM traps: Since they do not correspond to real users, any attempts to communicate with these accounts can be assumed to be SPAM. Chat networks are an attractive medium for distributing links to malicious sites, malware, and phishing attacks. In this regard, HoneyBuddy thus fills a gap in WOMBAT's view of the threat landscape, and produces results that can be further analysed by the existing components of the WOMBAT infrastructure. For instance, malware samples pushed to the honeypot can be analyzed and classified with a number of WOMBAT tools. This work was presented at a leading security conference [25]. *d*Anubis [63], presented in Chapter 3, is an extension of the Anubis dynamic malware analysis system, that allows us to analyze kernel-level windows malware (rootkits), thus similarly complementing our ability to observe and analyze a wide variety of threats.

Chapter 4 presents techniques for classifying code injection attacks collected via SGNET. This is done by taking into account both the attack employed and the malware payload that is ultimately delivered as a consequence of the attack. Interestingly, these techniques combine and contrast information obtained from behavioral, structural and contextual characteristics of malicious code, and leverage a variety of WOMBAT data and analysis techniques. Once again, this work has been published in a research paper [54], and includes extensive evaluation results from the real-world deployment of the proposed techniques.

The topic of Chapter 5 is HoneySpider Network (HSN), the large-scale client honeypot for detecting malicious web sites that was developed at NASK as a part of WOMBAT and has been described in previous deliverables. Section 5.1 presents results from the HSN URL-Map tool that was developed to visualize the results of HSN. Section 5.2 describes an additional component of HSN, which leverage machine learning techniques to filter out false positives. This component was developed largely as a consequence of the integration of HSN into the FIRE monitoring infrastructure, that was being implemented within WOMBAT WP5. During integration of HSN as a data-source for FIRE, it became apparent that a completely automated monitoring solution such as FIRE required a "cleaner" dataset, with a lower false positive rate than HSN initially was able to provide.

The final chapter of this deliverable (Chapter 6) focuses on how Argos and Shelia honeypots were improved within WOMBAT. Specifically, Section 6.1 briefly describes additional contextual features that can now be obtained from the Shelia client honeypot, compared to those described in deliverable D15 (D4.5). Section 6.2 introduces the use of Argos as a client-side honeypot that provides much more fine-grained tracking of shellcode execution. Finally, Section 6.3 discusses an extension of Argos that is very useful to remediate malware infections on compromised systems in a semi-automated fashion. The general idea is to automatically identify all changes to the system that directly or indirectly are a consequence of the malware's behavior. This is accomplished by making use of taint propagation techniques across the entire system (including both volatile and persistent storage).

2 HoneyBuddy (FORTH)

2.1 Introduction

Instant messaging is one of the most popular Internet activities. According to an older survey [6], more than 82 million people in Europe and 69 million people in North America use an instant messenger. A more recent study by Leskovec et al. [59] reveals that the number of MSN messenger (the most popular IM client) users has reached 240 million, with 7 billion exchanged messages per day. Reports estimate over 400 million registered Skype users [15], and 2.1 billion instant messages sent per day by AIM users[2].

This large user-base and the fact that IM is a near real-time form of communication, in contrast to other forms such as e-mail, make IM networks an attractive platform for attackers to launch their campaigns. Attackers either exploit vulnerabilities of the IM client software, or steal account information through phishing schemes. Once a user account has been compromised, the attack propagates by targeting the victim's contacts. The attack vectors are either file transfers or instant messages that contain URLs of websites controlled by the attacker. As users tend to trust content sent from their contacts, the probability of users accepting the transfer or clicking the URL is higher than in the case of traditional phishing campaigns or malicious websites.

This work focuses on the characterization and detection of attacks against IM users. Our proposed architecture, called Honeybuddy, is based on the concept of honeypots. Honeypots are closely monitored decoy machines that are not used by a human operator but rather wait to be attacked [70]. In a similar fashion, we have deployed IM honeypots: decoy IM accounts with hundreds of contacts on their friend list, that wait for compromised IM accounts to send them malicious URLs or files. Unlike traditional honeypots which wait for attackers passively, Honeybuddy follows a more active approach. By crawling the web or by advertising the accounts on several sites, we have managed to find accounts of real users and invite them to be our friends. Our decoy accounts have over six thousand users in their contact lists and receive between 50 and 110 unsolicited URLs per day.

In this deliverable, our main goal is to present an in-depth analysis and characterization of the collected URLs and malware. Our results show that 93% of the phishing URLs caught by Honeybuddy are not present in other popular existinging blacklist mechanisms, such as the Google blacklist. Additionally, as much as 87% of all malicious URLs collected by our infrastructure is incorrectly flagged as safe by commercial anti-phishing products, such as Norton Safe Web. We also cross-reference the malware caught by our system with other collection infrastructures and find that 21% of our samples are zero-day malware instances. During our analysis of the top-level domains that host malicious URLs we trace the phishing domains back to a small number of IP addresses, revealing a large network of collaborating attackers. We also found that 10 of those domains belong to fast-flux networks. Our study reveals that scams that propagate through IM networks are specifically crafted for this communication channel and are different from those found in email spam.

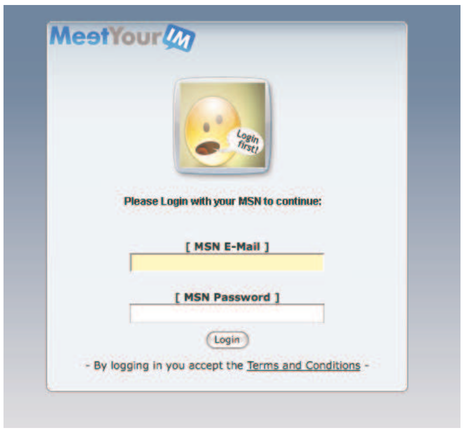
We provide an attacker profile based on our findings and describe two different strategies for deploying spam campaigns. We argue that different technical aspects of the IM attacks lead to radically different spamming strategies. Next, we examine the effectiveness of IM attack campaigns based on the launching of our own (benign) attack campaigns. This experiment reveals that 12% of the users visit URLs sent to them by our dummy accounts, and 4% are willing to run an executable of unknown origin.

2.2 Attacks on Instant Messaging networks

The high population of IM networks makes them an attractive target for attackers that try to exploit them for malicious purposes, such as spreading malware and scamming. We identify four different scenarios of attacks on IM networks.

Malware infection. Recent malware instances [24] can attach to a victim's instant messaging client and start sending URLs that point to malicious websites, or spread themselves by sending executables. In the most common case the malware instance logs in to the IM network, randomly selects users from the victim's contact list, sends the malicious URLs or files and then immediately logs out. In order to be more appealing to potential victims, the URLs point to domains whose name contains the username of the recipient, for example `http://contact_username.party-pics.com` . The vast majority of the attack campaigns we have detected send messages in English. However, we believe that attackers will soon shift towards localized messages, as is the case with one localized phishing site that we have detected.

Compromised accounts. Attackers can also use compromised credentials to log in as several different users and flood the victims' contact lists. Many services, like MSN, use unified credentials for e-mail and instant messaging, making life easier for attackers. Attackers can harvest IM accounts by setting up phishing sites for the service, by planting key-loggers or through social engineering. A relatively known attack campaign is that



Please Login with your MSN to continue:

[MSN E-Mail]

[MSN Password]

- By logging in you accept the Terms and Conditions -

Figure 2.1: Screenshot from an MSN phishing site.

of websites advertising a service that can reveal to users if someone has blocked them. If the user enters her IM credentials in the website, she is redirected to a page from another domain where nothing happens. Later on, the phishing site owner logs in as the user and sends messages to the victim's contact list. A screenshot of such a phishing site is displayed in Figure 2.1. A study of the phishing domains is presented in section 2.4.

Exploiting weak privacy settings. Even in the absence of malware infection or stolen credentials, some messengers provide the option to allow incoming messages from people who are not in the user's contact list. We tested the latest client versions of the most popular IM services: MSN live messenger (version 14.0.8089), Skype (version 4.1), Yahoo (version 10) and AIM (version 7.1). MSN live messenger is the only IM client we tested that has a privacy setting enabled by default that blocks messages from accounts not contained in the contact list. Skype, Yahoo and AIM by default allow anyone to send instant messages to our account, but this setting can be opted-out. Attackers exploit these settings to send unsolicited messages to IM users.

Exploiting client software. IM client software suffers from the problem of monocultures. Once an exploit is discovered, then automatically millions of clients can be infected immediately [23]. While in the case of malware infection exploits take advantage of the IM client to spread, this case involves the attack where the IM client is used to infect the rest of the machine.

2.3 Design and implementation

Honeybuddy was designed taking into consideration the four attack scenarios described in section 2.2. In contrast to previous work[87], Honeybuddy does not use modified versions of open source alternatives. It rather uses the latest version of the original clients, the same software most users install. The main reason for this choice is that direct attacks on IM client software will be detected. The basic concept behind Honeybuddy is to add random accounts to a decoy IM account and monitor the incoming connections. As Honeybuddy is in fact a honeypot, any incoming communication is by default suspicious. For our prototype we chose the MSN service due to its popularity. However, the design of Honeybuddy is generic enough to allow the fast implementation of other services as well, like AIM, Skype and Yahoo messengers. Furthermore, MSN live messenger 2009 inter-operates with Yahoo, and is planned to introduce interoperability with Google Talk, AIM and other services, rendering our architecture deployable for all major instant messaging services[1]. All deployed messengers run in a fully patched Windows XP SP3 system.

2.3.1 Architecture

Honeybuddy has three main components; a harvesting module, a script-based engine that handles the MSN messenger clients and the inspection module.

The harvesting module is responsible for gathering accounts that will later be added to the decoy accounts. All harvested accounts are inserted in CTT files (MSN contact files) that are imported in the messengers and all accounts listed are automatically invited. Another way is to search for e-mail addresses that belong to the @hotmail.com and @live.com domains. Other potential sources are sites where users advertise their MSN account, such as [9]. A more advanced method is to harvest account names from popular social networking sites.

[1]An experimental deployment of Skype and Yahoo honeypots collected too few URLs to extract any conclusions.

The script-based engine starts the messengers and invites all contacts gathered from the harvesting module. Based on the AutoIt software [3] , we can automatically start the application, import CTT files and invite other accounts to our friend list. The AutoIT software allows the manipulation of the windows of an application the same way a user would manually click, monitor the status of the application and check for new windows (in order to check for incoming messages). After encountering an attacker that waited for a reply to his initial message before sending the malicious URL, we modified our system to send a random automated response to each incoming message. When an incoming message comes and includes a request for a file transfer, the engine automatically accepts the transfer. As each messenger can only have a limited number of friends in its contact list, it is preferable to run multiple messengers. For resource efficiency reasons, we used MSN Polygamy [10] in order to run multiple MSN messengers on a single platform without the need of additional virtual machines.

The inspection module monitors the logs of the messengers for malicious URLs. It additionally checks the default download folder for new file transfers. An interesting finding is that we received URLs and malware in the Hotmail inboxes of our accounts. Thus, we extended the inspection module to also fetch and analyze e-mails, so as to extract URLs and executable attachments. All malicious URLs are stored in a database and are queried every one hour to check their uptime status.

2.3.2 Contact sources

We used two major sources for finding and adding contacts. The first one was queries for contact files and e-mail accounts belonging to the @hotmail.com and @live.com domains. Simple queries like "filetype:ctt msn" or "inurl:'@hotmail.com" were able to provide us with thousands of contacts. We also harvested e-mail accounts from other popular sites like `buddyfetch.com`[4], from which we extracted 38,000 hotmail addresses. Overall, we have invited 14,912 contacts to become friends with our accounts. 3,012 of those (20%) accepted our invitation. The exact number of invitations and acceptances per decoy account is displayed in Figure 2.2. The five decoy accounts denoted in Figure 2.2 as decoy accounts 14 to 18, sent a thousand invitations each, to addresses extracted from `buddyfetch.com`. We plan on adding the remaining accounts to our system in the near future. More advanced methods of harvesting [14] can be based on popular social networking sites like Facebook. By crawling such networks, one can collect IM accounts from accessible profile pages.

Other potential sources are sites where users advertise their MSN account, such as `messengerfinder` [9]. The `messengerfinder` site contains more than 25,000 active messenger contacts that are advertised by their owners for social networking purposes.

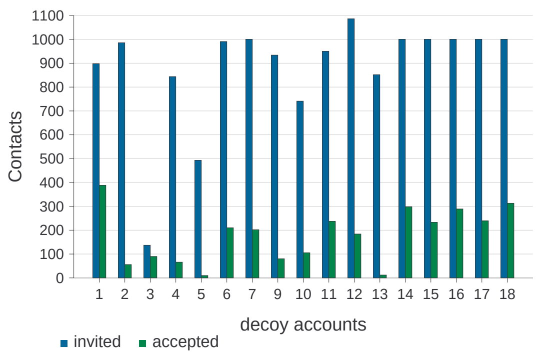


Figure 2.2: Number of friend invitations sent and number of accepted invitations, per decoy account.

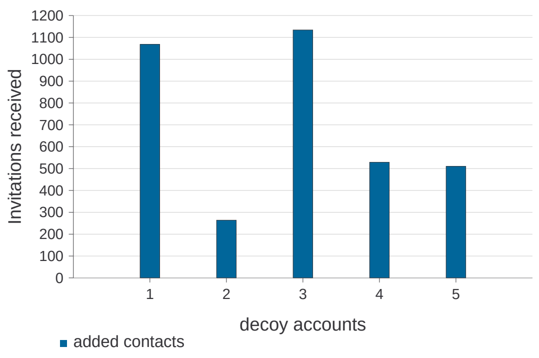
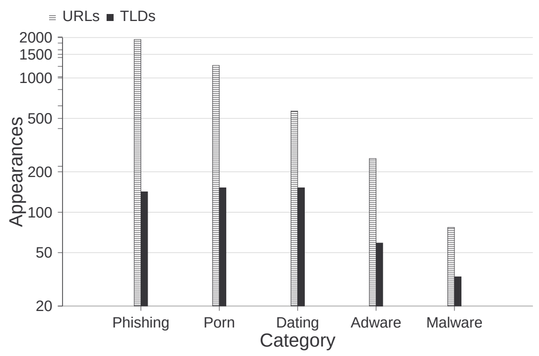


Figure 2.3: Number of invitations our decoy accounts received and accepted after being advertised on messengerfinder.com

We advertised our accounts on this site and instructed our honeypot messengers to accept any friend request. So far, we have added 3,505 contacts while this number increases daily. The exact number of contacts per decoy account is shown in Figure 2.3.

2.3.3 Hardening against direct attacks

Honeybuddy runs the latest version of the original IM client software and is, thus, vulnerable to infections. In order to prevent infections that will render our honeypot useless and make it a platform for further attacks, the honeypot messengers run inside Argos[68]. Argos is a containment environment that is based on memory tainting techniques. In a nutshell, Argos marks all bytes coming from the network as dirty and tracks their flow in the system memory. If a dirty byte is passed to the processor for execution, then we have an exploitation attempt since honeypots never download software to execute. When an exploit is detected, the application under attack is restarted and all attack information is logged to an alert file. This way, our system cannot be used as an attack platform as it is immune to remote exploits. A disadvantage of containment environments is that applications run 20 to 40 times slower than in a vanilla system. However, in the case of Honeybuddy this is not a problem as messengers are hardly demanding in terms of computing power and memory requirements, since most of the time they are idle and wait for incoming messages. In our experiments, each MSN client's memory footprint is



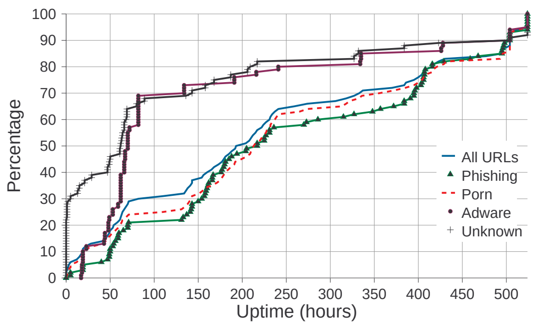


Figure 2.4: Classification of collected URLs Figure 2.5: CDF of uptime of URLs per category

50-80 MB and consumes negligible CPU resources.

We have to note, however, that during our experiments we have not yet encountered any attempts from attackers trying to exploit the IM client software. While this type of attack may not be common now, we believe that when IM attacks become more widespread, our implementation will provide useful information for such attacks.

2.4 Collected data analysis

In this section we provide an analysis of data collected by the Honeybuddy infrastructure. Despite the technical simplicity of our system, we were surprised by the fact that popular defense mechanisms had not detected the majority of our collected data. During the collection period, the Honeybuddy infrastructure collected 6,966 unique URLs that belong to 742 unique top-level domains.

During the first weeks of Honeybuddy operation we were able to fetch all URLs through the *wget* tool. However, malicious sites changed their behavior to avoid these fetches. Their pages now serve an obfuscated piece of Javascript code that changes the window location to a URL like `http://www.malicious.com/?key=<randomkeyhere>`. If a user has not visited the page with the key, then all subsequent requests are ignored and eventually her IP address is blocked for 24 hours. This behavioral change has forced us to fetch URLs through the Crowbar [5] environment that allows running javascript scrapers against a DOM to automate web sites scraping.

Our first step was to provide a simple classification for those URLs. Our five major categories were phishing, porn, dating, adware [2] and malware. The results are summarized in Figure 2.4. 1,933 of the URLs in 142 top-level domains were phishing for MSN accounts, 1,240 were porn, 567 were dating services and 251 sites were adware. While porn, dating and adware can be considered as harmless, the phishing sites pose a security danger for users. Furthermore, 77 URLs redirected to executable files or to pages that contained a direct link to a ".exe" or ".scr" file. We classify these URLs as malware.

We also spotted several sites that advertise subscription-based services for mobile phones. When the users enter their mobile phone number, they receive a subscription request. Once they subscribe to the service, they get charged for receiving SMS messages. These sites claim to give away free mobile devices to the subscribers of the service or promote quiz games that may attract victims, such as love calculators etc. These sites are highly localized. We visited them from different geographic locations using the Planetlab infrastructure [13] and got different pages in the language of the origin country. An interesting fact is that when the site cannot find the geolocation of the user, it redirects her to an MSN phishing site.

Our second step was to analyze the uptime of the collected URLs. The uptime graph can be seen in Figure 2.5. On average, a site is functional approximately for 240 hours (10 days). We also plotted the uptime graph for each category. We notice that porn and MSN phishing sites present much higher uptime than adware and unclassified sites. Half of the MSN phishing sites were alive for up to 250 hours (ten and a half days), while adware present a shorter lifetime of up to 80 hours (three and a half days).

2.4.1 MSN phishing

Attackers try to gather MSN credentials by tricking the user into entering her MSN e-mail and password in a bogus site. These sites falsely advertise a service that will reveal to the user which accounts from her contact list have blocked her. To validate that these phishing sites actually steal user credentials, we created several MSN accounts and entered them into the phishing sites. Each account had one of our decoy accounts as a friend. The decoy account received messages from the stolen MSN accounts that advertised the phishing site. However, the attackers did not change the passwords of any of the compromised accounts. After the attackers had started using our victim accounts, we submited "friend requests" towards these accounts. We wanted to see whether attackers will interfere with such matters and automatically accept requests. None of the submitted requests were accepted.

[2]We characterize sites that promote third-party addons for the MSN messenger (like extra winks, emoticons etc.) as adware sites

All phishing sites we visited shared one of three different "looks". A screenshot of such a site is shown in Figure 2.1. We analyzed the source HTML code of all the three "looks" and there was absolutely zero difference among the pages with the same look. This means the phishing pages with the same look had the exact same size and contained the same images and forms. This indicates that the majority of the different phishing campaigns might be deployed by a number of collaborating attackers. We also detected a localized phishing site which contained translated content, a technique used in e-mail spam campaigns[17]. The number of syntactical and grammatical errors revealed that the text translation was done automatically. For the time being, simple pattern matching for specific text segments is efficient for detecting these sites. Another detection mechanism is to query the various URL blacklists.

We queried the Google blacklist through the Google Safe Browsing API [7] to check if it included the phishing sites we discovered. From the 142 unique top-level domains (TLD) that hosted phishing sites and were detected by Honeybuddy, only 11 were listed by Google blacklist. That means that 93% of the domains captured by Honeybuddy were not listed elsewhere on their day of detection, making Honeybuddy an attractive solution for MSN phishing detection. The average delay from when our system detected one of the 11 sites until it was included in the Google blacklist was around two weeks, leaving a time window of 15 days for attackers to trick users. Firefox, one of the most popular browsers uses the Google Safe Browsing API as an anti-phishing measure. We also compared our findings with the blacklist maintained by SURBL [19] and URLblacklist.com [21]. SURBL detected only 1 out of the 142 MSN phishing domains (0.7%) and none of the adware domains. None of the phishing or adware sites were listed by URLblacklist.com.

A very interesting fact was that when resolved, all the unique top level domains translated to a much smaller number of unique IP addresses. This fact confirms our initial theory that all these phishing campaigns lead to a limited number of collaborating attackers. To further investigate this behaviour, we conducted an experiment for a period of almost two months, presented in Section 2.5.

2.4.2 Malware sample analysis

In this section we provide an analysis of the malware collected by the Honeybuddy infrastructure. Our infrastructure collected 19 unique malware samples. We distinguish the malware collected by the Honeybuddy infrastructure into two categories, the direct malware set and the indirect malware set. We present the two categories and proceed to further analyze the collected samples.

The first category contains malware samples collected either through direct file transfers (uncommon case) or by visiting URLs that were redirected to executable files. In

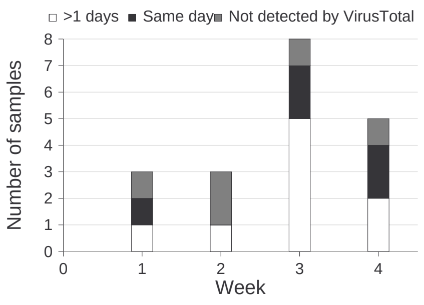


Figure 2.6: Detection delay of collected samples compared to the VirusTotal database. 21% of the samples were previously unseen, while 26% were collected the same day they entered the VirusTotal database.

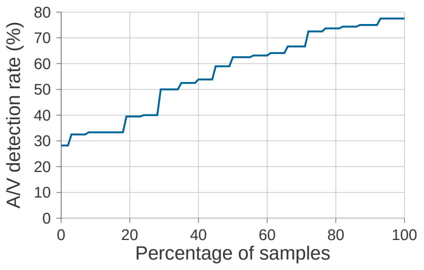


Figure 2.7: Cumulative distribution function of detection rate for collected samples based on VirusTotal reports. 42% of the samples were detected by 50% of the anti-virus engines.

the case of the URLs, the e-mail account of the victim was always appended as a parameter to make it look more realistic. In some cases attackers used popular keywords, like Facebook.

The second category, the indirect one, contains malware samples collected in two types of cases. In the first case, users are presented with a web page that alerts them that they need to download the latest version of the “adobe flash plugin” so as to play a certain video strip, and are prompted to download the installer which is, obviously, malware. In the second case, users are redirected to a page prompting them to install a screen saver. This “.scr” file they are prompted to download and install is a malicious file that infects the machine upon execution.

Due to the small volume of files, we were able to manually check these files using the Anubis analysis center [1]. All of them were characterized as dangerous , while some of them were bots that connected to an IRC C&C server. By joining the IRC network, we downloaded even more malware samples (not listed in this section).

In order to verify how original our samples are, we submitted them to the VirusTotal [22] service. VirusTotal is a large malware collection center with the primary goal of providing a free online virus and malware scan report for uploaded samples. The reason we chose to use VirusTotal is twofold. First, VirusTotal receives around 100,000 samples every day from a multitude of sources resulting in a large database of malware samples. The second and most important reason is that VirusTotal collaborates with a large number of well known anti-virus vendors[3] and uses their anti-virus engines and, therefore, can provide us with an accurate picture of the efficiency of up-to-date, state-of-the-art defense solutions for home users.

Four collected samples had not been seen by VirusTotal before, that is 21% of our samples were previously unseen malware instances. Figure 2.6 shows the relative detection delay compared to the date the samples entered the VirusTotal database. The base bar of the stack graph (solid white) shows how many samples were detected with a delay of one or more days, the middle bar (solid black) displays the number of samples that were detected the same day as VirusTotal while the top bar shows the number of samples not included in the VirusTotal database. Five samples (26%) were collected the same day they entered the VirusTotal database, while the maximum detection delay was five days.

We also checked the VirusTotal analysis reports for the collected samples. 42% of the samples were detected by half of the anti-virus engines, while the maximum detection rate was 77%. However, the dates of the analysis reports were one month after the collection date as we did not submit the samples the day they were captured. The one month delay means higher detection rates for the anti-virus engines as they update their signature files daily. Even in that case, it can be observed that there are samples that are recognized by only one third of the anti-virus products. The cumulative distribution function of detection rates can be seen in Figure 2.7.

2.4.3 Mailbox analysis

In this section we present an analysis of the emails we found in the mailboxes of our decoy accounts. Our analysis focuses on two aspects of the incoming emails. First, whether the body of the email contains any URLs and, second, whether the email contains any attachments. The decoy accounts received a total of 4,209 emails, 403 of which contained a total of 1,136 attachments. The emails contained 5,581 URLs which were passed to our classifier. The goal of the classification was to only identify phishing URLs and URLs that downloaded malware. 26 of the received URLs belonged to phishing domains while 7 downloaded malware samples.

While the majority of the attachments were pictures, several were windows media files and office documents. We checked the VirusTotal database for the MD5 hashes of the files but found no matches. This was expected, since hotmail scans incoming emails

[3] For a complete list refer to `http://www.virustotal.com/sobre.html`

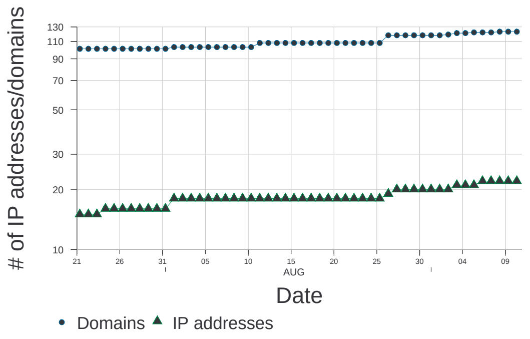


Figure 2.8: Number of distinct phishing domains and the IP addresses they resolve to over time.

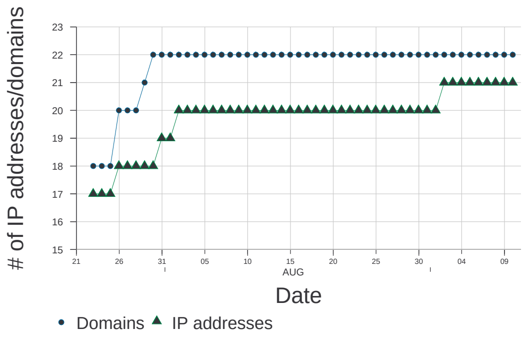


Figure 2.9: Number of distinct malware-distributing domains and the IP addresses they resolve to over time.

for malware and blocks executables. The most interesting attachments were two ".zip" files. Once extracted, the zip files returned a ".lnk" file. Upon inspection, we found that the files were command line scripts that connect to an FTP site and download and execute malicious software. For a more detailed analysis of the file refer to this report by F-Secure [8].

2.4.4 Comparison to email spam

An important aspect of IM based attacks that we wanted to explore was whether they are scams that use other commmunication channels as well or if they are unique to this attack medium. In particular, we wanted to explore similarities between the campaigns collected by Honeybuddy and scams that propagate through emails that are collected by spam traps. We obtained 458,615 spam emails collected by Spam Archive [16]. From these emails we extracted 467,211 unique URLs that belonged to 52,000 unique TLDs. We compared them to our 6,966 unique URLs and found only one common instance of a well-known benign website. This result was expected since URLs in IM attacks usually contain the target's username. Next, we compared the unique TLDs and found 21 common domains, 9 of which were popular and benign domains. From the 12 suspicious domains, 3 were classified as porn, 3 hosted malware, 2 were for dating, 1 was for adware and 3 were not assigned to a known category. None of the common TLDs hosted msn phishing scams, the prevalent threat of IM networks. Therefore, we can conclude that

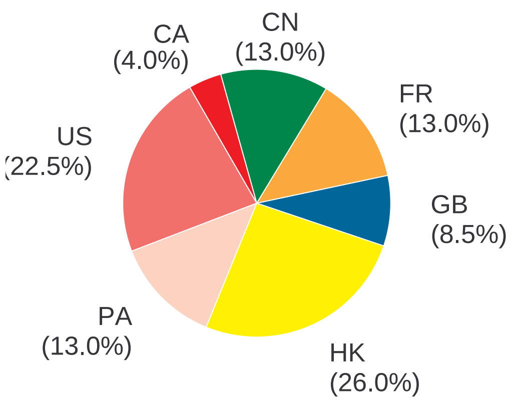


Figure 2.10: Breakdown of countries that host the phishing domains.

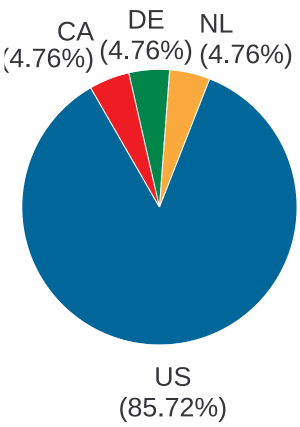


Figure 2.11: Breakdown of countries that host the malware-distributing domains.

attackers have crafted scams specific to this new attack medium that are fundamentally different to existing email spam campaigns.

The results of this comparison are a strong indication that scams that propagate through IM networks are new campaigns and not email spam campaigns that utilized a new attack channel. They present their own unique properties that differentiate them from traditional scams that propagate through spam emails.

2.4.5 Comparison to commercial anti-phishing product

Multiple anti-virus vendors provide software products designed to protect users from malicious sites. To verify whether the URLs we collect have already been seen by large vendors that offer anti-phishing products, we evaluated our URLs using such a service. We used the Norton Safe Web service [11] provided by Symantec [20], where users can submit a URL and receive a report that contains information regarding possible threats from the specific website. We submitted the 2,010 phishing and malware URLs collected by our infrastructure after the collection period. Submitting the URLs after the collection period and not each URL individually upon collection, results in a potentially higher detection rate for the Norton Safe Web service. Even so, Norton Safe Web flagged only 13% of the submitted URLs as dangerous or suspicious. Specifically, 246 phishing and 10 malware-distributing URLs were reported as malicious while all other pages were characterized as safe. That means that over 87% of the malicious URLs collected by Honeybuddy had not been properly categorized as dangerous by one of the largest security vendors.

2.5 Hosting analysis

The fact that all the top-level domains of the URLs collected from our initial experiment translated to a very small number of IP addresses, urged us to conduct a new experiment that might reveal more information. For a period of 50 days, we periodically ran *nslookup*[12] for each of the unique top level domains our system had collected up to that moment, in order to gather more information regarding how and where attackers host phishing and malware-distributing domains. Here we present the results for each category of domains separately and highlight their particular behaviour.

The experiment gave us further insight in regards to the small number of IP addresses that host a multitude of phishing campaigns. All top level domains translated to one or two IP addresses, while 98% of them translated to only one. Furthermore, ten of the top level domains belonged to fast-flux networks and translated to a different set of IP addresses each time. In Figure 2.8 we can see that during the first days of the experiment, all 101 top-level domains translated to only 14 different IP addresses. The TLDs that belonged to fast-flux networks are excluded from the graphs. This behaviour is consistent throughout the duration of the experiment and as new domains were added only a small number of new unique IP addresses appeared.

Next we wanted to track down the country where the domains were hosted. We used the MaxMind [4] database. In Figure 2.10 we can see the breakdown of the percentages of the countries that hosted the top level domains. Honk Kong ranks first hosting 26% of the domains, while the United states follow with 22%. A surprising result is that only 13% of the domains were hosted in China, which is quite lower than what we would expect based on reports [18].

Next we present the results from the experiment regarding domains that distribute malware. Our initial goal was to investigate whether the top level domains of the malware-distributing websites also translate to only a small number of IP addresses.

In Figure 2.9 we present the results from this experiment. We can see that in the case of the URLs that contain malware, the top level domains translated to different IP addresses. Unlike the phishing domains, here each top level domain translated to a different IP address, and only one to three IP addresses overlapped at each moment in time. The IP address that hosted three malware-distributing domains also hosted

[4]http://www.maxmind.com/app/geolitecity

one of the phishing domains and was located in the United States. None of the other IP addresses hosted both a malware-distributing and a phishing domain. The nslookup operation for all the top level domains returned only one IP address, and only one of the domains belonged to a fast-flux network pointing to a different address each time. Since the IP addresses that host phishing domains are more likely to be blacklisted, this result is not surprising. Another interesting fact is that none of the top level domains is in both of the sets, meaning that none of the domains hosted a phishing site and simultaneously distributed malware.

Similarly to the phishing domains, we wanted to trace the country where the malware-distributing domains were hosted. In Figure 2.11 we can see the breakdown of the percentages of the countries. The United States were responsible for hosting the majority of the domains that distribute malware through IM traffic, reaching almost 86%. The remaining three countries, Canada Germany and the Netherlands, hosted an equal amount of domains. Once again, it is surprising that China did not host any of the domains caught by our infrastructure.

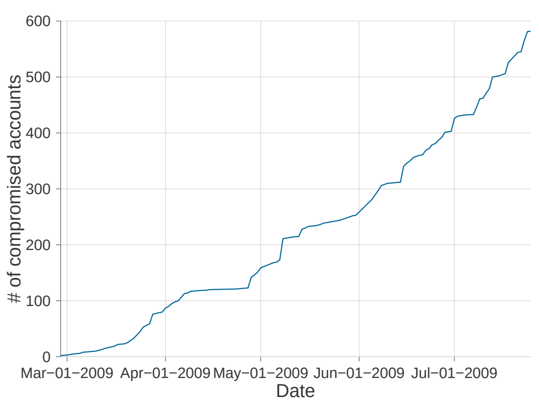
2.6 Attacker profile

In this section we present statistics and observations in an effort to outline the behaviour of IM attackers and recognize different strategy patterns.

First of all, in Figure 2.12 we present the number of unique compromised accounts that had sent URLs to our decoy accounts over time. We can see that the plot line follows a sub-linear curve, with almost 100 new accounts contacting us each month. This indicates that over time legitimate users still follow malicious URLs sent by compromised "buddies" and in turn get infected too.

In Figure 2.13 we can see the CDF plot of the number of URLs sent by each compromised account to our infrastructure throughout the duration of the experiment. One should note that approximatelly 25% of the compromised accounts sent only one URL and 40% up to two URLS. Based on the numbers we can identify one possible strategy that attackers choose to follow.

Eventhough some of these accounts/hosts may have been dis-infected before sending us another URL, it is improbable to assume that all of them were "cleaned up". Therefore, this might indicate a cautious behaviour on behalf of the attackers. With 55% of the compromised accounts sending up to 4 URLs and 75% sending less than 20, it is evident that one strategy that attackers follow is to avoid aggressive spamming behaviours so as not to raise suspicions among the compromised accounts' contacts. Such aggressive behaviours could alert the user and lead to the dis-infection of the account/machine.



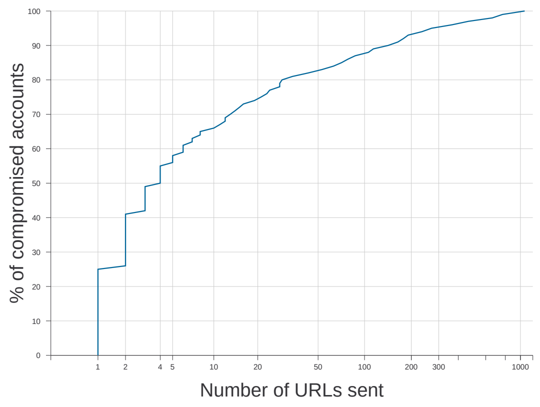


Figure 2.12: Number of compromised accounts that contacted our decoy accounts over time.

Figure 2.13: CDF of the number of URLs sent by compromised accounts to our decoy accounts.

However, this cautious behaviour could also be attributed to technical reasons. If the attack is propagated through a worm that infects the client, then a low rate of worm propagation would be used so as not to trigger antivirus or intrusion detection systems.

Furthermore, approximately 12% of the attackers sent at least 100 URLs to our decoy accounts. This aggressive strategy of massively dispatching spam messages, indicates a category of attackers that don't try to remain beneath a certain threshold. This can also be attributed to technical reasons. Specifically, amongst the top ten compromised accounts that sent us the largest number of URLs, we found all the victim accounts whose credentials we had entered in phishing sites. Therefore, the attackers use tools to send messages from these compromised accounts without relying on worms that infect the IM client software. Thus, we can recognize a second more aggressive strategy, where it is not necessary for attackers to adopt a stealthy propagation rate. Finally, it is interesting to note that attackers send URLs from all the categories, as well as malware, and do not focus on one specific type.

2.7 Real-case Evaluation

We were interested in investigating the potential effectiveness of an IM attack campaign. To do so, we decided to launch our own benign campaign targeting the contacts of one

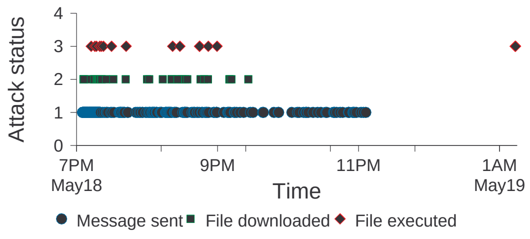


Figure 2.14: Time series of events that occurred during our benign campaign.

of our honeypots. There were two factors to be taken into consideration. The first one was localization. Several users would get suspecious if we sent them messages that were not in their native language. We queried the profile pages of most of the contacts we had at our disposal but unfortunately we could not retrieve country information for most of them, so we decided not to localize the messages sent. The second one was whether a conversation would take place before actually sending the URL. A message containing a URL without any prior conversation might get flagged as suspicious immediately.

Nonetheless, we decided to launch a simple spam campaign that imitated the ones caught by Honeybuddy, that would not provide biased results. We logged in one of our honeypot accounts and sent the following message to the online contacts: "Hey! Check this out: `http://mygallery.webhop.net/gallery1/photo1.jpg`". The URL pointed to a web server controlled by us and redirected the user to a web page that asked the user to download a (benign) executable. The executable was a harmless program that just requested another URL, again from our own web server. That way, we were able to track if and when a user actually executed the file she downloaded. We contacted each online contact only once for the whole duration of the experiment.

The results of the campaign are summarized in Figure 2.14. The bottom series plots the timestamps when the message was sent to an online user. The middle series plots the timestamps when a contact visited the URL contained in the sent message and downloaded the executable. The top series displays the timestamps when a contact actually ran the downloaded executable. During our campaign we sent 231 messages. 27 unique users (11.6%) visited the URL and downloaded the file, while 9 of them (4%) executed the downloaded file. We repeated the same experiment with two other accounts and the results were almost the same.

3 dAnubis (TUV)

3.1 Introduction

Malicious code, or malware, is at the root of many security problems on the internet. Compromised computers running malware join botnets and participate in harmful activities such as spam, identity theft and distributed denial of service attacks. It is therefore no surprise that a large body of previous research has focused on collecting, detecting, analysing and mitigating the impact of malicious code.

The analysis of malicious code is an important element of current efforts to protect computer users and systems from malware. Understanding the impact of a malware sample allows to evaluate the risk it poses and helps develop detection signatures, removal tools and mitigation strategies. Because of the large number of new malware samples that appear in the wild each day, malware analysis needs to be a largely automated process.

The automatic analysis of malicious programs is complicated by the fact that malware authors can use off-the-shelf packers to make their samples extremely resistant to static code-analysis techniques. According to a recent large-scale study of current malware [31], over 40% of malware samples are packed using a known packer. Clearly, this is a lower bound to the amount of malware that is packed because malware authors may be using other, yet-unknown packers or implement their own custom solutions. While many current packers can be defeated by generic unpacking tools [77, 51], packers that use emulation-based packing can currently be fully defeated only after manually reverse-engineering their emulator [76]. Furthermore, packers based on opaque constants [62], while not yet available in the wild, can generate binaries that are provably hard to analyze for any static code analyzer.

Because of these limitations, automatic malware analysis is mostly based on a dynamic approach: Malware samples are executed in an instrumented sandbox environment, and their behavior is observed and recorded. A number of dynamic malware analysis systems are currently available that can provide a human-readable report on the malware's activities [29, 85]. The output of these tools can further be used to find clusters of samples with similar behavior [27, 34, 73], or to detect specific classes of malicious activity [46].

These systems are able to analyse the behavior of malicious code running in user-mode. The analysis of kernel-side malicious code, however, poses additional challenges. First of all, kernel-level malicious code cannot be reliably detected or analyzed unless the analysis is performed at a higher privilege level than the kernel itself. Otherwise, kernel-level malware would be able to tamper with or disable the analysis engine, in a never-ending arms race. This challenge can be overcome by using out-of-the-box, Virtual Machine Introspection techniques [43], or with more recent in-the-box monitoring techniques that leverage modern CPU features to protect the analysis engine [81]. Using such techniques, the injection and execution of code into kernel-space can be reliably detected [72, 74].

Beyond detection, however, understanding the purpose and capabilities of malicious kernel code is also useful. This is challenging because, in contrast to a user-mode process, kernel code is not restricted to its own address space and to interacting with the rest of the system through a well-defined system call interface. When monitoring the behavior of a system infected by kernel-side malicious code, it is not trivial to reliably (a) attribute an observed event to the malicious code or to the benign kernel and (b) understand the high-level semantics of an observed event. In the limit, kernel-level malware could replace the entire operating system kernel with its own implementation, making understanding the differences between the behavior of a clean system and an infected one extremely challenging. In practice, malware authors prefer to perform targeted manipulations of the operating system's behavior using hooking techniques, and to make use of functions offered by the kernel rather than re-implement existing functionality. Therefore, detecting malware hooking behavior has been the focus of a significant body of recent research [84, 90, 52, 75].

One aspect of malicious kernel code that has received less attention is device driver behavior. That is, the malware's interaction with the system's IO driver stacks, and the interface and functionality it offers to userland processes. In this work, we attempt to provide a more complete picture of the behavior of malicious kernel code. We introduce *d*Anubis, an extension to the Anubis dynamic malware analysis system[35] for the analysis of malicious Windows device drivers. *d*Anubis can automatically generate a human-readable report of the behavior of kernel malware. In addition to providing information on the use of common rootkit techniques such as call hooking, kernel patching and Direct Kernel Object Manipulation (DKOM), *d*Anubis provides information about a malicious driver's interaction with other drivers and the interface it offers to userspace. To improve the coverage of its dynamic analysis, *d*Anubis also includes a stimulation engine that attempts to trigger rootkit functionality. Running *d*Anubis on over 400 malware samples that include kernel components allows us not only to validate our tool, but also to perform the largest study of kernel-level malware to date.

In summary, our contributions are the following.

1. We present *d*Anubis, a system for the real-time dynamic analysis of malicious Windows device drivers.
2. Using *d*Anubis, we analyzed over 400 hundred samples and present the results of the first large-scale study of Windows kernel malware. These results give insight into current kernel malware and provide directions for future research.
3. *d*Anubis will be integrated into the Anubis malware analysis service, making it available to researchers and security professionals worldwide.

3.2 Overview

Rootkits provide malware authors with one of their most flexible and powerful tools. The term "rootkit" derives from their original purpose of maintaining root access after exploiting a system, being a "kit" of pieces of technology with the purpose to hide the attacker's presence in the system [44]. This can include hiding files, processes, registry keys and network ports that could reveal an intruder's access to the system. Early rootkits ran entirely in user space and operated by replacing system utilities such as ls, ps and netstat with versions modified to hide the activities of an unauthorized user. Later rootkits included kernel-level code, enabling the attacker to do virtually anything on the target machine, including directly tampering with control flow and data structures of the operating system. Today, the boundaries between different classes of malware have become indistinct; many techniques originally used in rootkits are now employed in other types of malware, such as bots, worms or Trojan horses. In this paper, we will use the term *rootkit* to refer to malware that uses kernel-level code to carry out its operations.

To inject malicious code into the kernel, the attacker can either use an undetected, unpatched kernel exploit, such as a buffer overflow, or – much more convenient – load and install a device driver. The latter method has the disadvantage that it depends on hijacking an administrator account. This is in practice not much of a problem since most Windows machines are operated with Administrator privileges out of convenience for the user. While Windows Vista or 7 at least require the user to confirm administrative actions such as driver loading, Windows XP provides APIs that allow loading an unsigned driver without any user interaction. As a result, a rootkit usually comes as a user-mode executable that loads a device driver, which in turn provides all the powerful functionality.

The goal of *d*Anubis is to provide a human-readable report of a device driver's behavior. Note that detection, that is, distinguishing malicious device drivers from benign ones, is outside the scope of this work. Some behavior, such as directly patching kernel code, may give a clear indication that a sample is malicious. Many types of suspicious behavior, however, may also be exhibited by benign code, especially security tools such as antivirus or personal firewall software. The reason is that these tools may attempt to "outsmart" malware by running deep inside the operating system.

*d*Anubis analyses a driver's behavior from outside the box, using a Virtual Machine Introspection (VMI) approach [43, 47]. Our implementation is an extension of the Anubis malware analysis system, and is based on the Qemu [37] emulator. By instrumenting the emulator, we can monitor the execution of code in the guest OS, to observe events such as the execution of the malicious driver's code, invocation of kernel functions, or access to the guest's virtual hardware. Furthermore, by instrumenting the emulator's Memory Management Unit (MMU), we can observe the manipulation of kernel memory performed by the rootkit. *d*Anubis attempts to reconstruct the high level semantics of the observed events.

One focus of our analysis is to monitor all the "legitimate" communication channels between the rootkit and the rest of the system. That is, all channels provided by the OS for the driver to interact with the kernel, with other drivers and with user-space. This includes the invocation of kernel functions as well as the use of devices to participate in Windows I/O communication and to receive commands from user-space. Additionally, *d*Anubis can detect the use of a number of rootkit techniques such as hooking and runtime patching of kernel routines, and provide precise information on which routines are patched or hijacked. Overall, our tool can thus provide a comprehensive picture of the behavior of malicious kernel code.

3.3 System Implementation

A major drawback of any VMI-based approach is the loss of semantic information about the guest operating system. Instead of objects and well-defined data structures, only a heap of bytes is visible from the host system's point of view. To reconstruct the necessary information we extract all exported symbols and data structure layouts from the Windows OS as a preliminary step. During analysis, we utilize guest view casting of the virtual machine memory as proposed by Jiang et al. [47].

A further problem arises when comparing process-based dynamic analysis, as it is implemented in Anubis or comparable sandboxes, and driver-aware approaches. For userland processes, it is sufficient to watch and trace instructions belonging to the process in question, whose execution context is easily identifiable. Kernel-level code, however, can be triggered by multiple means, like interrupts or system calls, thus possibly running in the context of an arbitrary user-mode process. Therefore, we use the instruction

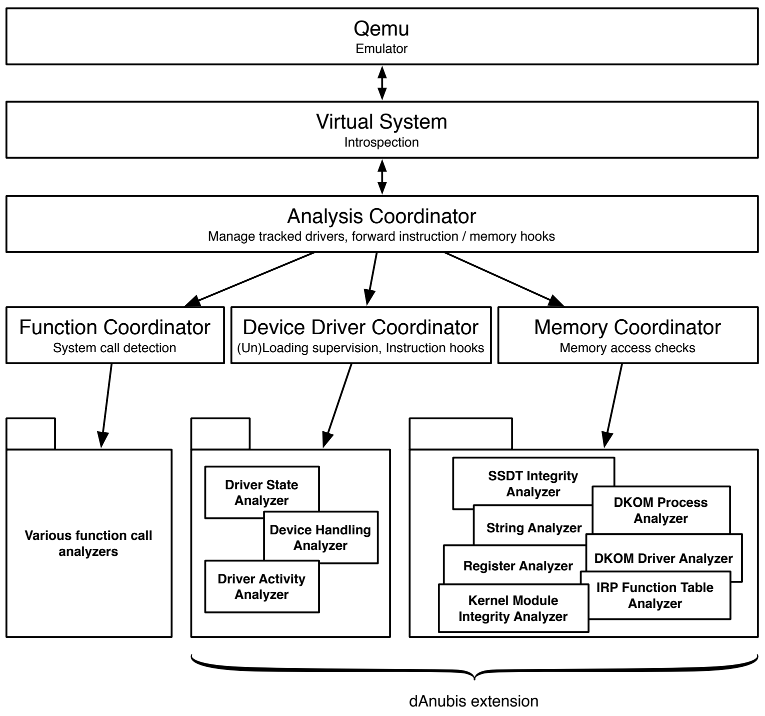


Figure 3.1: Architectural overview

pointer to determine whether the code being executed belongs to the malicious driver.

Our system consists of two major parts. The *Device Driver Coordinator* handles device driver-related operations while the *Memory Coordinator* is responsible for rootkit activity. Specific analysis tasks are carried out by a number of *Analyzers*. Figure 3.1 provides an overview of our system's architecture.

3.3.1 Device Driver Analysis

To analyze the behavior of device drivers, we monitor all available interfaces through which the driver can interact with the rest of the kernel, with other drivers and with userland processes [66]. The first thing to do in this respect is intercepting the low-level load- and unload mechanisms. The Windows kernel objects involved in the loading procedure provide us with import information, above all the codebase location of the driver, which allows us to track instructions belonging to the driver. Furthermore, the function addresses of the driver's entry routine (comparable to the main function of a normal program), its unload routine and its I/O dispatch routines can be gathered. Among the latter are the *major functions*, a set of well-defined functions a driver has to provide in order to participate in Windows I/O device communication. Knowing the function addresses allows us to implement a basic state supervision and relate later analysis events to the context in which they occurred.

Driver communication. The communication endpoint of a driver is the device. To receive I/O requests, a driver has to create a device and provide the aforementioned major functions to handle the requests. If requests require complex processing that can be subdivided in different parts, devices can be arranged in a stack. For example this could be the case for an encrypted file system, where the encryption is handled by the topmost driver in the stack and actual hardware access by the lowest driver. Driver stacking can be exploited for malicious purposes. For example, a rootkit may attach to the filesystem device stack to filter the results of file listings before forwarding them up the stack, while a keylogger can attach to the keyboard device to monitor all keystrokes. Therefore, we monitor whether a driver creates or attaches to a device.

Actual communication between devices and user- or kernel-mode code happens by encapsulating the request parameters in an I/O request packet (IRP) by the Windows I/O manager, which invokes the corresponding major function of the topmost driver in the stack. The IRP is then passed down and up its destined device stack. We intercept calls to attached devices, analyze these IRPs and watch for completion routines, which are invoked upon completion of a request, allowing them to filter results. Larger data amounts are not directly passed within the IRP, rather the way how it can be done – buffered I/O, direct I/O or neither of them – is specified. We also parse this information and detect strings in the data to be able to track further references to them during execution. This is accomplished using dynamic data tainting [64]. Specifically, we use techniques from [34] to detect the use of these tainted bytes in string comparison operations. This can in some cases reveal triggers for conditional behavior of the analyzed binary.

Driver activity. To get a picture of what the driver actually does when one of its functions is executed, we log calls to all exported Windows kernel functions. We expect that rootkit developers will not develop everything from scratch but make use of existing functionality. In our evaluation, we show that this assumption proves correct for most real-world samples.

We also scan the complete driver image for string occurrences and taint them to be able to log any subsequent access to such strings. As we will show in the evaluation, this simple mechanism can in practice reveal trigger conditions in the malicious code, such as the names of files and processes that are to be hidden.

3.3.2 Memory Analysis

Taking a look at the “standard” rootkit techniques, one similarity is obvious: they all somehow tamper with kernel memory. We achieve the goal of detecting malicious kernel memory manipulation by hooking the memory management unit (MMU) of Qemu. This allows us to detect write access independently of the instruction used so that we can put specific memory regions of the guest OS under supervision and analyse malicious changes.

Since some kernel regions are flagged read-only, rootkits often use memory descriptor lists (MDLs) to re-map a desired memory region to a new virtual address and bypass write protection. Therefore *d*Anubis also needs to analyze MDL usage.

Call table hooking. The first interesting memory region is represented by the system service dispatch table (SSDT). This table keeps a list of Windows system call handlers that can be invoked by usermode processes by issuing an interrupt or using the `sysenter` instruction. In a healthy system, the called table entry points to the beginning of the desired service routine. Malicious drivers, however, can overwrite the SSDT entry to point to arbitrary code. When called, this code typically forwards the request to the original service function, receives the response and alters it before passing it to the original caller. Again, this method provides the possibility to exclude rootkit-related information from queries like directory listings or process lists.

The same principle applies to hooks of other call tables, such as the major function dispatch table of device drivers. For incoming IRPs the Windows I/O manager normally looks up the proper handling routine in this table before invoking it. Again, the control flow can be re-routed by changing an entry in this table.

To monitor call table hooking behavior, we watch the complete memory region where a call table resides. If a manipulation occurs in one of the watched memory regions, we will know exactly which system service or major function has been hooked and monitor following calls to the hook.

DKOM. Direct kernel object manipulation (DKOM) modifies important data objects residing in Windows kernel memory. This way it can alter system behavior without changing the control flow. To hide a certain process from the process list, for example, the `EPROCESS` structure has to be altered such, that the forward and backward pointers are directed around the target entry, effectively excluding it. Although this method is more powerful and harder to detect than hooking, it has its shortcomings. It is, for instance, not feasible to mask a file from a directory listing this way.

To detect and understand DKOM activity, it is necessary to know the exact location of the kernel objects as well as their data structure and meaning. In our current implementation, DKOM detection is limited to the process and driver lists, that are frequently targeted by rootkits for the purpose of stealth.

Runtime patching. Rootkits can also affect the system by directly patching existing kernel code in memory. Usually the patch jumps to a detour containing malicious code and then back again to the original code.

To detect runtime patching, we walk through the `PsLoadedModuleList` to get the information on the codebase of the kernel modules and put them under supervision. On an integrity breach we determine exactly which kernel function has been patched by matching the patched addresses against information automatically obtained from the Windows debugging symbols.

Hardware access. In addition to manipulating kernel memory, rootkits can affect the system by directly accessing the underlying hardware. *d*Anubis monitoring of hardware access is currently limited to detecting writes to the `IA32_SYSENTER_EIP` model specific register. This register points to the system service dispatcher routine. Making this register point to malicious code places rootkit code in the execution path of all system calls.

3.3.3 Stimulation

Rootkit functionality often depends on external stimuli. While the entry routine may already perform some malicious activity such as hooking or patching, many types of behavior may only be performed when triggered by specific user behavior. Without the required stimuli, such as keyboard events for keyloggers or process enumeration for process-hiding rootkits, the results of dynamic analysis are bound to be incomplete.

Our goal is to improve code coverage by simulating user activity with a stimulation engine placed in the virtual machine. To this end, we implemented a stimulator that repeatedly issues a number of Windows API calls. For example, the stimulator issues the `EnumProcesses` API call, that lists all currently running processes, triggering process hiding behavior. Similarly, it issues the `RegEnumKeyEx` call, revealing register hiding. The `FindFirstFile` and `FindNextFile` API calls are used as well to reveal file hiding behavior. Note that although the directory we are querying with these calls might not contain files to be hidden, hook code will nevertheless be executed. To trigger network hiding behavior, the `GetUdpTable` and `GetTcpTable` calls are used. Furthermore, random keypresses and mouse actions are injected to simulate user input.

3.4 Evaluation

To evaluate our prototype, we first verified its functionality on a set of rootkits with known behavior. For this, we chose a representative suite of six well-known rootkits that employ the popular techniques described in the previous sections and can be obtained from `www.rootkit.com`. For each of the six rootkits, *d*Anubis was able to correctly identify its characteristic behavior and present it in the human-readable report. Table 3.1 shows which *d*Anubis components were involved in providing information on each of the rootkits.

The first sample we selected is **TCPIRPHook**. This malware modifies the address of `DEVICE_CONTROL` in the major function table of Tcpip.sys, rerouting it to a hooking function. This allows the rootkit to hide open network ports. This hooking behavior was detected by the IRP function table analyzer.

We then selected the **HideProcessMDL** rootkit as a straightforward example of process hiding by SSDT hooking. This rootkit first creates an MDL in order to gain write access to the SSDT. This is recognized by the memory coordinator, that can thus apply the mapping upon write access to the watched memory region. This enables the SSDT integrity analyzer to report the rootkit's hooking of `NtQuerySystemInformation` as soon as the hook is placed. Once the stimulator queries for the running processes, the driver state analyzer detects the call to the hooking routine, which in turn invokes the original `NtQuerySystemInformation` function. Furthermore, The string analyzer reveals that the hooking routine accesses the string "_root_" indicating the name of the process that is to be hidden.

Klog is a key-logger based on layered filter drivers. During driver initialization it creates a log file and a virtual device, which it uses to attach to the keyboard device stack. This behavior, along with name and location of the log file, is revealed by the device handling, driver activity and string analyzer. Upon stimulation of keystrokes, the device handling analyzer further detects that the driver Kbdclass is called by Klog and dynamically adds the completion routine to the state analysis and so execution of the completion routine is subsequently logged.

Migbot uses run-time patching to modify the kernel functions `SeAccessCheck` and `NtDeviceIoControlFile`. The integrity breach along with the names of the functions is immediately reported by the kernel integrity analyzer.

The **FU** rootkit uses DKOM for process and driver hiding. However, it only performs this hiding function when it receives commands from a user-mode program through device communication. To test the DKOM analyzers we manually ordered FU to hide certain processes and drivers. These manipulations along with the corresponding filenames were immediately reported by the DKOM analyzers. Furthermore, the device handling analyzer revealed the string "msdirectx.sys" in the communication with the user-mode program. This is the name of the driver we ordered FU to hide.

Finally, we tested the **sysenter** rootkit to verify that sysenter hooks are correctly recognized.

Analyzer	TCPIRPHook	HideProcessMDL	Migbot	Klog	FU	sysenter
Device driver coordinator	√	√	√	√	√	√
Memory coordinator	-	√	-	-	-	-
Driver state analyzer	√	√	√	√	√	√
Driver activity analyzer	√	√	√	√	√	-
IRP function table analyzer	√	-	-	-	-	-
SSDT analyzer	-	√	-	-	-	-
String analyzer	-	√	-	√	√	-
Integrity analyzer	-	-	√	-	-	-
Device handling analyzer	-	-	-	√	√	-
DKOM process analyzer	-	-	-	-	√	-
DKOM driver analyzer	-	-	-	-	√	-
Register analyzer	-	-	-	-	-	√

Table 3.1: *d*Anubis Testing results

During the analyis of these six rootkits, we also measured the impact of *d*Anubis on the performance of the Anubis sandbox. The overhead added by *d*Anubis was between 14% and 33%. These results are consistent with our goal of integrating driver analyis into a large-scale dynamic analysis framework, because the entire analysis of a malware sample can still be performed in real time, within the six minute timeslot that Anubis typically allocates to an analysis run. This is in contrast to some previous systems, such as K-Tracer [52], that need to perform a heavyweight analysis of detailed execution traces. For instance, K-Tracer needed over two hours to analyze the HideProcessMDL rootkit.

3.4.1 Quantitative results

We used *d*Anubis to conduct a large-scale study of kernel malware behavior. To obtain malware samples for this study, we leveraged the analysis results of the existing Anubis system. We first considered 64733 malware samples successfully analysed by Anubis in the month of August 2009. Among those, we selected the 463 samples (0.72%) that

Driver activity	number of samples exhibiting behavior
Device driver loaded	463
Windows kernel functions used	360
Windows device I/O used	339
Strings accessed	300
Kernel code patched	76
Kernel call tables manipulated	37
MDL allocated	34
Kernel object manipulated	3

Table 3.2: Global analysis statistics

loaded a device driver during Anubis analysis. More precisely, we selected samples that performed the `NtLoadDeviceDriver` system call. We then repeated the analysis of these samples using *d*Anubis. Note that some malware may use different mechanisms to load kernel code, such as the undocumented `NtSetSystemInformation` system call. Therefore, the actual number of rootkit samples in the dataset may have been higher than 463. While *d*Anubis is capable of correctly analysing rootkits loaded using such methods, the legacy Anubis system does not detect and log this behavior.

All samples were automatically processed by our implementation and correctly recognized as drivers. For each test run, we defined a timeout of six minutes, during which the driver had time to carry out its operations. During the entire analysis stimuli where provided by our stimulation engine. Table 3.2 shows which high-level activity of the samples could be observed by *d*Anubis. Three quarters of the samples performed device I/O activity. Among the typical rootkit techniques, MDL-enabled call table hooks and runtime patching seem to be very popular compared to DKOM.

Table 3.3 shows an overview of device-related activity. The majority of the samples – 339 – created at least one device. In 110 cases the device was actively used for communication by a user mode program: It was at least opened, as indicated by the calls to the CREATE major function. Out of these, 86 samples carried out further communication using the device control interface. In the data buffers passed along with the IRPs, meaningful communication strings could be found in 24 cases (an example is shown in Table 3.7). Only two samples attached to a device stack and registered completion routines. These results allow us to draw the conclusion that devices are primarily used for communication with user mode programs whereas hijacking device stacks seems to be far less popular. However, a significant amount of samples – 229

Device activity	number of samples
Device created	339
Driver's device accessed from user mode	110
Strings detected during communication	24
Attaches to device stack	2
Registers completion routine	2

Table 3.3: Device analysis statistics

	SSDT hook	runtime patching	DKOM	IRP hook	Filter driver	total
Registry	5	45	0	0	0	50
File	8	2	0	0	2	12
Process	3	2	3	0	0	8
Driver	0	0	3	0	0	3
Network port	5	0	0	1	0	6

Table 3.4: Hiding statistics: subject vs. technique

– register a device, but this device is never put to any use during the entire analysis run. The most likely explaination for this discrepancy is that the associated executables are merely launchers for the drivers, that in turn wait for further commands to be manually issued by the human attacker. This is the case of the FU rootkit we previously discussed. This means that some of the malicious functionality of these rootkits lies dormant, waiting for activation, and is therefore not covered by the dynamic analysis. This result highlights the need for further research in rootkit analysis. Future analysis systems might be able to automatically trigger rootkits' dormant functionality, although the problem of finding trigger conditions in arbitary code cannot be solved in general [80].

Overall, only 15% percent of the samples carried out rootkit activities. Table 3.4 shows the amount of samples that provided stealth broken down by the techniques employed as well as the type of object being hidden. Clearly, call table hooking and runtime patching are the more widespread techniques: only three samples used DKOM for process hiding. The same samples also used DKOM to hide their device drivers from the list of kernel modules.

Of the 19 samples that employed SSDT hooking, most hooked more than one system call. Table 3.5 shows the most popular system calls hooked. The idea that rootkits strive to provide stealth is confirmed by the fact that the system calls to list files, registry keys and processes are clearly favored by the attackers. In addition to stealth, another common goal of rootkits is to disable antivirus protection. Samples hooking the `NtCreateProcessEx` use this to prevent the launch of anti-malware programs. IRP function table hooks were only employed by one sample, that hooked the `DEVICE_CONTROL` major function of Tcpip.sys. Rerouting the device control interface, that is the main communication access point to the driver, allows this rootkit to hide open network ports. A less subtle method of hijacking the device control interface is to directly hook the `NtDeviceIoControlFile` system call. This technique is used by five samples, also for the purpose of port hiding. None of the samples used sysenter hooks. The StringAnalyzer, that was mainly introduced to reveal trigger conditions, shows its full potential with SSDT hooks. For example in more than half of the cases where `NtQueryDirectoryFile` or `NtQuerySystemInformation` has been hooked, the filenames to be hidden showed up in the analysis. This also demonstrates the importance of event stimulation and the effectiveness of our stimulation engine, as the strings were mainly detected during execution of hooking routines that would not have been called without stimulation.

System service	samples
NtQueryDirectoryFile	8
NtCreateProcessEx	8
NtDeviceIoControlFile	5
NtEnumerateKey	3
NtQuerySystemInformation	3
NtEnumerateValueKey	2
NtOpenKey	2
NtClose	1
NtCreateKey	1
NtSetInformationFile	1
NtSystemDebugControl	1
NtOpenProcess	1
NtOpenThread	1
NtCreateFile	1
NtOpenIoCompletion	1
NtSetValueKey	1
NtDeleteValueKey	1
NtMapViewOfSection	1

Table 3.5: Hooked system calls

About ten percent of the samples used runtime patching. In these cases *d*Anubis took advantage of kernel debugging symbols to automatically identify the patched kernel functions.

Table 3.6 shows that, as is the case for SSDT hooks, functions that take part in processing file, process and registry key queries are among the most popular kernel functions to be manipulated. The `pIofCallDriver` pointer points to the low-level kernel code implementation that invokes the major functions of a driver. Rerouting its control flow allows a rootkit to intercept and manipulate IRPs. The `KiFastCallEntry` function is the default handler of the `sysenter` instruction. By patching this function, a rootkit inserts malicious code into the code path of every system call. In this case the automatic analysis cannot tell us what types of objects are actually being hidden by this rootkit.

Kernel function	samples
NtQueryValueKey	42
NtSetValueKey	2
PsActiveProcessHead	2
NtEnumerateKey	1
IoCreateFile	1
NtQueryDirectoryFile	1
NtOpenKey	1
NtCreateKey	1
pIofCallDriver	1
KiFastCallEntry	1
ObReferenceObjectByHandle	1
KiDoubleFaultStack	1

Table 3.6: Patched kernel functions

3.4.2 Qualitative results

To provide a clearer picture of the types of behavior that can be revealed by the analysis of a kernel malware sample using *d*Anubis, we selected three interesting samples out of the dataset discussed in the previous section. For matters of space and readability the relevant information from the reports has been condensed into tables.

In Table 3.7 a selection of analysis results of Sample A are shown. This rootkit hooks various system calls, among them functions suitable for file and registry key hiding. Process hiding is performed using DKOM. As the hooking functions are very similarly

Driver name	syssrv	
Created devices	\Device\MyDriver	
Rootkit activity	NtOpenProcess hooked	SSDT Hook
	NtOpenThread hooked	SSDT Hook
	NtCreateFile hooked	SSDT Hook
	NtOpenIoCompletion hooked	SSDT Hook
	NtQueryDirectoryFile hooked	SSDT Hook
	NtOpenKey hooked	SSDT Hook
	NtEnumerateKey hooked	SSDT Hook
	NtEnumerateValueKey hooked	SSDT Hook
	NtSetValueKey hooked	SSDT Hook
	NtDeleteValueKey hooked	SSDT Hook
	svchost.exe hidden	DKOM process hiding
	ntoskrnl.exe: PsActiveProcessHead	Runtime patching
Invoked major functions	CREATE	called 5x from user mode
	DEVICE_CONTROL	called 5x from user mode
	CLOSE	called 5x from kernel mode
Detected strings	syssrv	in DEVICE_CONTROL IRP
	\Device\HarddiskVolume1 \WINDOWS\system32\mssrv32.exe	in DEVICE_CONTROL IRP
	SOFTWARE\Microsoft\Windows \CurrentVersion\Run\mssrv32	in DEVICE_CONTROL IRP
	\Device\%s	during entry
	MyDriver	during entry
Used kernel functions	IoCreateDevice	during entry
	KeInitializeMutex	during entry
	ObReferenceObjectByName	during DEVICE_CONTROL
	ObReferenceObjectByHandle	during DEVICE_CONTROL
	ObQueryNameString	during DEVICE_CONTROL
	KeWaitForSingleObject	during DEVICE_CONTROL
	KeReleaseMutex	during DEVICE_CONTROL
	PsLookupProcessByProcessId	during DEVICE_CONTROL
	NtEnumerateKey	during NtEnumerateKey Hook
	ObReferenceObjectByHandle	during NtEnumerateKey Hook
	ObQueryNameString	during NtEnumerateKey Hook
	wcslen, wcscpy, wcscat	during NtEnumerateKey Hook
	KeWaitForSingleObject	during NtEnumerateKey Hook
	KeReleaseMutex	during NtEnumerateKey Hook

Table 3.7: Analysis report, Sample A

structured, the hook of NtEnumerateKey has been chosen as an example. After calling the original function it queries an object and its name. It then performs some string operations, which is usually necessary for filtering information. Furthermore, in the course of the driver's entry function a device is created. This device is then used for user mode communication: DEVICE_CONTROL is called from user mode several times and both a registry key and a file name could be intercepted, that the driver is presumably ordered to hide. DEVICE_CONTROL itself looks up objects according to their name or ID using ObReferenceObjectByName and PsLookupProcessByProcessId – again an indication that these objects are to be hidden.

In Table 3.8 selected analysis results of Sample B are shown. During driver entry, this sample creates a named device (FILEMON701) for communication with user mode. This device is then used to issue commands to the driver via `FastIoDeviceControl` to install filter drivers for sr.sys and mrxsmb.sys. To this end, two unnamed devices are created and attached to the associated device stacks.

The sr.sys driver is the Windows restore filesystem filter driver, that tracks and copies system files before changes. The mrxsmb.sys is the Windows SMB Redirector, a filesystem driver that provides access to remote folders shared over the SMB/CIFS protocol. Our stimulation engine, however, does not perform operations on network shares, nor does it modify system files. Therefore, during anlysis we only observed interception of `QUERY_VOLUME_INFORMATION` of sr.sys, that is used to query free disk space or file types. This highlights the challenge of implementing a stimulation engine that is capable of activating all hooks inserted by a rootkit. Note that generic hook-detection tecniques such as Hookfinder [90] and KTracer [52] cannot detect hooks that are never activated.

The Windows system restore functionality can be used to perform a system rollback based on the information gathered by the sr.sys driver. By attaching to sr.sys, the rootkit can prevent system restore from obtaining the necesary information on system file changes and ensure that the rootkit will not be removed by a rollback. The device name and symbols included in the rootkit's executable suggest that the publicly available sources of the Filemon tool [78] were used as a basis for this rootkit.

Sample C performs SSDT hooking on the `NtQueryDirectoryFile` and `NtEnumerateValueKey` system calls to provide stealth. Furthermore, this sample calls the `PsSetLoadImageNotifyRoutine` to receive a callback whenever a process or driver image is loaded. The sting analyzer reveals that this callback accesses a number of strings hardcoded in the rootkit image, that are shown in Table 3.9. These strings are clearly filenames, most of them related to antivirus software or other security tools. The most logical explanation for these observations is that the rootkit uses this technique to interfere with the loading and execution of anti-malware programs. Manual analysis confirms that the callback uses the `ZwTerminateProcess` function to kill these processes. We could not directly observe

Driver name	FILEMON701	
Created devices	\Device\Filemon701 unnamed device 1 unnamed device 2	
Attached to devices	sr MRxSmb	
Completion routine	QUERY_VOLUME_INFORMATION	for device "sr"

Invoked I/O functions	CREATE
	QUERY_VOLUME_INFORMATION
	CLEANUP
	CLOSE
	READ
	FastIoDeviceControl
		from user mode
		from kernel mode
		from kernel mode
		from kernel mode
		from kernel mode

Used kernel functions	IoCreateDevice IoCreateSymbolicLink IoGetCurrentProcess ZwCreateFile IoCreateDevice IoAttachDeviceByPointer	during entry during entry during entry during FastIoDeviceControl during FastIoDeviceControl during FastIoDeviceControl

Table 3.8: Analysis report, Sample B

vsdatant.sys	watchdog.sys	zclient.exe	bcfilter.sys	bcftdi.sys
bc_hassh_f.sys	bc_ip_f.sys	bc_ngn.sys	bc_pat_f.sys	bc_prt_f.sys
bc_tdi_f.sys	filtnt.sys	sandbox.sys	mpfirewall.sys	msssrv.exe
mcshield.exe	fsbl.exe	avz.exe	avp.exe	avpm.exe
kavsvc.exe	klswd.exe	ccapp.exe	ccevtmgr.exe	ccpxysvc.exe
issvc.exe	rtvscan.exe	savscan.exe	bdss.exe	bdmcon.exe
cclaw.exe	fsav32.exe	fsm32.exe	gcasserv.exe	icmon.exe
nod32krn.exe	nod32ra.exe	pavfnsvr.exe	kav.exe	kavss.exe
inetupd.exe	livesrv.exe	iao.exe	Windows-KB890830-V1.32.exe	

Table 3.9: Processes targeted by Sample C

this behavior during analysis because none of the listed processes are executed in our analysis environment. This attack on security software highlights the need for a secure execution context for analysis software. The watchdog.sys file that is among those targeted by the rootkit is a driver that is also used by CWSandbox [85], a malware analysis system that is not based on VMI but on in-the-box monitoring using a kernel driver.

3.5 Related work

In this section we will discuss related research in the area of detection and analysis of malicious kernel code.

Integrity checking. In [72] the authors implement a tamper-resistant rootkit detector for Linux systems that uses VMI. To detect runtime patching, this system verifies the integrity of the kernel by hashing portions of clean memory considered critical and regularly comparing the hashes with their up-to-date counterparts. A limitation of this approach lies in the need to balance security with performance in selecting how often to perform hashing. In any case, such a system cannot guarantee that no injected code will ever be executed. To protect kernel code, an improved solution is offered by Nickle [74]. Nickle instruments the memory management unit of an emulator to redirect code fetches performed in kernel mode to a protected memory region. This way, it can detect code injected into the kernel as soon as its first instruction is executed.

Cross-view detection. Hiding an intruder's presence on a compromised system is a widespread goal of rootkits. This very behavior, however, can be exploited to detect kernel compromise. For this, cross-view detection approaches [43, 36] compare system information obtained from a high-level abstract view, e.g. the Windows API, with information extracted from a lower level view, in order to reveal hiding. [72] uses a cross-view approach to detect process, kernel module and network port hiding. To detect process hiding, the authors of [49] use Antfarm [48] to determine implicit process information, without prior knowledge of the monitored guest's OS. They then perform cross-view comparison, detecting process hiding. While OS-independence is an attractive advantage of this approach, the technique employed cannot be easily extended to other types of stealth behavior. A drawback of cross-view is that it can detect the fact that something has been hidden, but cannot provide any information on how this has been done. Moreover, for information arranged in more complex data structures cross-view soon becomes impractical. For example, it can only detect file hiding if the contents of every directory on the system are compared.

Hooking detection. Hooking is another characteristic aspect of rootkit behavior that can be exploited for detection purposes. In [84], the authors present Hookmap, a system that can systematically discover possible hooking points in the execution path of system calls, enabling detection of rootkits that use these hooking points. Hookfinder [90] uses dynamic taint propagation to monitor the impact of a rootkit on the kernel. It detects a hook when a tainted value is loaded to the instruction pointer. In [52], the authors introduce k-tracer, a system that performs a sophisticated analysis of malware hooking behavior. For this, k-tracer first records an execution trace for a system call, reaching from the `sysenter` to the `sysexit` instruction. Then, an offline analysis is applied to the trace, by performing forward slicing to identify *read access* and backward slicing to reveal *manipulation* of sensitive data. This approach is however not compatible with our performance requirements for large-scale malware analysis, since k-tracer may require hours to analyze a single rootkit.

Rootkit analysis. Closely related to *d*Anubis is recent work on the dynamic analysis of rootkit behavior. In [88], the authors present rkprofiler, a system for the analysis of Windows kernel malware that is also based on VMI using Qemu. This system can reveal which system calls have had their execution paths modified to include injected code. Both [88] and [75] address the problem of understanding the semantics of rootkit modifications to dynamically allocated kernel memory. For this, they introduce techniques to recursively infer the type of an object in memory based on the type of the pointers that are used to access it, starting from the known structure of static kernel objects and function parameters.

3.6 Limitations

Our evaluation demonstrates that *d*Anubis can provide a substantial amount of information on malicious drivers. Nonetheless, our system suffers from a number of limitations.

Rootkit detection. To be able to analyze a rootkit's behavior, *d*Anubis must first detect the rootkit's presence in the analysed system. That is, it must be aware that extraneous code has been inserted into kernel space. For this, *d*Anubis relies on hooking system calls used for loading drivers. Therefore, we are unable to analyse rootkits injected through kernel or device driver exploits. This is a design choice, because it allows most *d*Anubis instrumentation to remain disabled until a driver is loaded, improving performance on the majority of analysed samples that *do not* load a driver. At the cost of some performance, this limitation could be addressed by integrating techniques from [74], that can reliably detect the execution of injected code. The detection of return-oriented rootkits [45], however, remains an open problem, since these rootkits do not inject any code into kernel space.

Dynamic analysis coverage. A general limitation of dynamic approaches to code analysis is that only code that is actually executed can be analyzed. In order to cover as many code paths as possible, we strive to stimulate typical rootkit functionality. Behavior that is triggered by benign user activity can be emulated to a certain extent by our stimulator. However, our large scale study has shown that many samples waits for commands to be issued from userspace through a device interface and never receive any such commands during analysis. The rootkit behavior associated with these commands is therefore not covered by our analysis. Related work in this field [75, 52, 88] does not specifically address this issue. Future research in rootkit analysis could attempt to design a stimulator capable of automatically issuing valid commands to malicious device drivers.

Related to the problem of coverage is the issue of detection of virtual environments and of analysis environments in general. If malware can detect our analysis environment it can thwart analyis by simply refusing to run. Unfortunately, implementing an undetectable virtual environment is infeasable in practice [42], although attackers may be reluctant to make their malware not function on widely deployed virtual environments. Defeating VM detection is largely a reactive, manual process. However, recent research [67, 28] has shown that it may be possible to automatically detect previously unknown virtualization detection techniques.

Event attribution. In order to differentiate between legitimate and malicious actions, the origin of these actions has to be determined. To attribute a write access to a monitored driver we take the program counter of the instruction that carried out the manipulation and compare it with the codebase of the driver. While this technique works in practice, it can easily be fooled if the malicious driver uses a legitimate kernel function to manipulate the desired memory region. [79] introduces secure control attribution techniques based on taint tracking to tackle a similar problem in the context of (malicious) shared-memory browser extensions. Since Anubis provides tainting support, these techniques could also be adapted for integration in *d*Anubis.

3.7 Conclusions

The analysis of malicious code faces additional challenges when the code to be analyzed executes in kernel space. In this work, we discussed the design and implementation of *d*Anubis, a system for the dynamic analysis of Windows kernel malware. *d*Anubis can provide a comprehensive picture of a device driver's behavior and its interaction with the operating system, with other drivers and with userland processes.

We used *d*Anubis to conduct a large-scale study of kernel malware behavior that provides novel insight into current kernel-level threats. In this study, we analysed more than

400 recent rootkit samples to reveal the techniques employed to subvert the Windows kernel and, in most cases, the nefarious goals attained with these techniques. These results demonstrate that *d*Anubis can be an effective tool for security researchers and practitioners. We therefore plan to make it publicly available as part of the Anubis malware analysis service.

This work was published in the proceedings of the International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) [63].

4 Exploiting diverse observation perspectives to get insights on the malware landscape. (SYM/IEU)

We are witnessing an increasing complexity in the malware analysis scenario. The usage of polymorphic techniques generates a new challenge: it is often difficult to discern the instance of a known polymorphic malware from that of a newly encountered malware family, and to evaluate the impact of patching and code sharing among malware writers in order to prioritize analysis efforts.

Several indicators suggest an exponential explosion of the number of newly generated malicious samples per day. For instance, according to [38], the amount of samples submitted to VirusTotal [83], a popular virus scanning platform, is in the order of millions of samples per month. Such numbers translate into a daily load of approximately 30,000 samples per day. This load can be partially explained by the easiness with which malware writers can generate new code by personalizing existing code bases, or by re-packing the binaries using code obfuscation tools [61, 60, 89]. In addition, malware sample counts are biased by the increasing usage of polymorphic techniques [32]. For instance, worms such as Allaple [41] take advantage of these techniques to mutate the binary content of the executable file at each propagation attempt.

In previous deliverables, we have presented different approaches to deal with the problem of polymorphism in malware analysis. In D16 (D4.2) we have presented the results obtained through the application of clustering techniques to the behavioral profiles generated by the execution of each sample in a sandboxed environment [34]. In D17 (D4.4) we have presented the results obtained by applying a highly scalable and simple technique to cluster malware samples according to their structural characteristics, called "feature-based" malware clustering. Finally, in D18 (D4.6) we have described the contextual information generated by SGNET honeypots for each collected malware sample. All these techniques and datasets contribute to the generation of a global picture of the threat landscape by providing different means to assess the relationships and commonalities among different malware sample instances.

In this deliverable, we present the outcomes of an experiment carried out on a large portion of the SGNET dataset in an attempt to evaluate the quality and the usefulness of the information generated by these different methodologies. The outcomes of this study have been presented at the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks [54] and show the complimentarity of these techniques at building information on the malware landscape.

The experiment underlines important lessons that should drive future research on the topic:

- Despite the general consensus in the research community that static approaches are not suitable to cluster and classify malware, we show in this work that most polymorphic engines in the wild have a low level of sophistication and that simple static techniques can still be useful for clustering malware. More specifically, we show the effectiveness of a very simple pattern generation technique in classifying the propagation strategy and the structural characteristics of the malware samples observed by the SGNET dataset.
- We show how, by combining clustering techniques based on either static or behavioral characteristics of the malware samples, we are able to detect clustering anomalies generated by different environmental causes. Moreover, we show how the combination of the two feature sets offers insights on patching and code sharing practices in the observed malware samples that would be invisible to any clustering technique based on a single feature type.
- Many malware collection techniques solely focus on the collection of malware binaries. We underline, through practical examples, the usefulness of combining the knowledge generated by malware classification approaches with contextual information on malware propagation generated by the SGNET deployment. We show the importance of such information to generate rich, structured knowledge that helps the security analyst to obtain a better understanding of the "economy" of the different threats and on the modus operandi of the individuals at their root cause.

4.1 Methodology

We have based our analysis on the information generated by SGNET, described in detail in D13 (D3.3)[55]. In contrast to other malware collections, SGNET focuses on the collection of detailed information on code injection attacks and on the sources responsible for these attacks. Each malware sample collected by the deployment is, therefore, enriched by contextual information on the attacks, the evolution of the attack in time, and on the structure of the code injection itself. The contextual information provided by the SGNET dataset is generated by an information enrichment approach

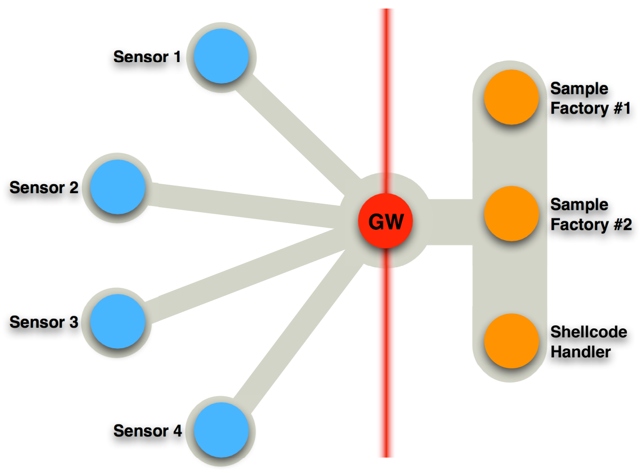


Figure 4.1: SGNET architecture

[56] that aggregates data generated by different analysis tools such as VirusTotal [83] and Anubis [33, 1].

4.1.1 The SGNET dataset

SGNET is a distributed honeypot deployment focusing on the study of code injection attacks. SGNET was originally presented in [55], and then presented with further details in [53]. SGNET takes advantage of protocol learning techniques to address a trade-off between the need to retrieve rich information about the observed activities by prolonging as much as possible the conversation with clients and the need to reduce the resource and maintenance costs inherent in a distributed deployment. By using ScriptGen [57, 58], SGNET honeypots are able to model protocol conversation through a Finite State Machine (FSM) model and use such models to respond to clients for well-known activities. Whenever a new/unknown activity is encountered, SGNET honeypots are able to dynamically proxy the conversations to a honeyfarm, and take advantage of the real service implementation to handle the interaction.

Figure 4.1 shows the main components of the SGNET deployment. SGNET is composed of multiple low-cost sensors whose FSM model is kept in sync by a central entity, the gateway. Whenever a new activity is encountered, SGNET honeypots require the instantiation of a new *sample factory* to the central gateway. The *sample factory*, based on Argos [69], acts as an oracle and provides to the sensors the required protocol interaction and, through memory tainting, detects and provides information on successful code injection attacks. Such information is used by the gateway to apply the ScriptGen algorithm and refine the FSM knowledge. After having seen a sufficient number of samples of the same type of interaction, SGNET sensors are, therefore, able to handle autonomously future instances of the same activity leveraging the newly built FSM refinement.

The memory tainting information generated by Argos, combined with simple heuristics, allows SGNET honeypots to identify injected shellcodes. SGNET takes advantage of part of the Nepenthes [26] modules to understand the intended behavior of the observed shellcodes and emulate the network actions associated to it.

All the information collected during the interaction of the different SGNET entities is stored in a database, and fed into an information enrichment component [56] that is in charge of adding additional metadata on the attacking sources, and on the collected malware. Among the different information sources, the most relevant to this work are the behavioral information generated by Anubis [33], and the AV detection statistics generated by VirusTotal [83]. Every malware sample collected by the SGNET infrastructure is, in fact, automatically analyzed by to these two services, and the resulting analysis reports are stored in the SGNET dataset to enrich the knowledge about the injection event.

4.1.2 EPM clustering

The SGNET dataset provides information on the different phases of each observed code injection attack. In [55], we show that SGNET is structured upon an epsilon-gamma-pi-mu (EGPM) model, an extension of the model initially proposed in [40]. The EGPM model structures each injection attack into four distinct phases:

- **Exploit (ϵ).** The set of network bytes being mapped onto data which is used for conditional control flow decisions. This consists in the set of client requests that the attacker needs to perform in order to lead the vulnerable service to the failure point.
- **Bogus control data (γ).** The set of network bytes being mapped into control data which hijacks the control flow trace and redirects it to somewhere else.
- **Payload (π).** The set of network bytes to which the attacker redirects the control flow through the usage of ϵ and γ.
- **Malware (μ).** The binary content uploaded to the victim through the execution of π, and that allows the attacker to run more sophisticated operations that would

be impossible in the limited space normally available to the payload π.

In order to exploit the SGNET information, it is necessary to cope with the degree of variation introduced by the different propagation strategies. An example of this variation is the polymorphic techniques used to mutate the content of the payload and of the malware. However, we can have even simpler cases, such as the usage of a random filename in the FTP request used to download the malware sample to the victim.

There are several existing techniques for the quick classification of polymorphic samples starting from static features. Because of the unique characteristics of the dataset at our disposal, we have chosen not to reuse these techniques, and to introduce a new, simple pattern discovery technique, sufficiently generic to be applied independently to exploit, shellcode and malware features. We call this technique EPM clustering. The EPM clustering technique is a simplification of the multidimensional clustering technique described by Julisch in [50] and used in the context of IDS alerts. This technique is intentionally simple, and could be easily evaded in the future by more sophisticated polymorphic engines. Nevertheless, although it is simple, in practice, it proved to be sufficient to deal with the current sophistication level of the malware samples observed in the SGNET dataset. EPM clustering is based on the assumption that any randomization performed by the attacker has a limited *scope*. We assume that any polymorphic or randomization technique aims at introducing variability only in certain *features* of the code injection attacks. Since the randomization of each feature has a cost for the attacker, we can assume that it will always be possible to take into account a sufficiently large amount of features to detect a certain amount of *invariants* useful for recognizing a certain class of attacks.

For instance, we observed that polymorphic techniques such as that of the Allaple worm [41] obfuscate and randomize the data and code sections of the executable files, but do not normally perform more expensive operations such as relinking, or leaving invariants when looking at the Portable Executable (PE) headers.

EPM clustering is composed by 4 different phases, namely feature definition, invariant discovery, pattern discovery and pattern-based classification. The technique is applied independently to the three distinct dimensions of the EGPM model: ϵ, π and μ.[1]

Phase 1: feature definition

The feature definition phase consists of defining a set of features that have proven to be useful in our experiments to characterize a certain activity class. Table 4.1 shows the

[1]We do not consider γ in the classification due to lack of host-based information in the SGNET dataset.

Dim.	Feature	# invariants
Epsilon	FSM path identifier	50
	Destination port	3
Pi	Download protocol (FTP/HTTP/...)	6
	Filename in protocol interaction	22
	Port involved in protocol interaction	4
	Interaction type (PUSH/PULL/central)	5
Mu	File MD5	57
	File size in bytes	95
	File type according to libmagic signatures	7
	(PE) Machine type	1
	(PE) Number of sections	8
	(PE) Number of imported DLLs	7
	(PE) OS version	1
	(PE) Linker version	7
	(PE) Names of the sections	43
	(PE) Imported DLLs	11
	(PE) Referenced Kernel32.dll symbols	15

Table 4.1: Selected features

list of features that we have taken into consideration to characterize the code injections in each dimension of the Epsilon-Pi-Mu space.

Most of the exploit classification is based on the information provided by the interaction of the attacker with the ScriptGen FSM models. Because of the way FSM models are built [58], a given FSM path incorporates together protocol features *and* specificities of a certain implementation. If all the network interactions take advantage of the same username, or the same NetBios connection identifiers, such parameters will be part of the generated model, and will be differentiated from other exploit implementations.

The Nepenthes shellcode analyzer provides information on the intended behavior of each shellcode collected by the SGNET deployment. Such information includes the type of protocol involved in the interaction (e.g., FTP, HTTP, as well as some Nepenthes-specific protocols), the filename requested in the interaction (when available) and the involved "server" port. Depending on the involved protocol and on the entity interacting with the victim, we distinguish also between three types of interaction: 1) PUSH-based, in which the attacker actively connects to the victim and pushes the sample. 2) PULL-based (also known as phone-home), in which the victim is forced to connect back to the attacker and download the sample. 3) Based on a central repository, in the case in which the PULL-based download interacts with a third party different from the attacker itself.

The malware dimension, because of the extensive use of polymorphic techniques, requires a higher number of attributes in order to correctly classify samples. We have taken into consideration simple file properties, such as its size, as well as Portable Executable information extracted from the samples taking advantage of the PEfile [39] library. In order to identify non-polymorphic samples, the MD5 of the sample was also considered as a possible invariant. Looking at the level of sophistication of the SGNET malware collection, PE header characteristics seem to be more difficult to mutate over different polymorphic instances. These characterictics, thus, are good discriminants to distinguish different variants, since a change in their value is likely to be associated to a modification or recompilation of the existing codebase.

Clearly, all of the features taken into account for the classification could be easily randomized by the malware writer in order to evade our static clustering approach. More complex (and costly) polymorphic approaches might appear in the future leading to the need to generate more sophisticated techniques. This justifies the interest in continuously carrying on the collection of data on the threat landscape and on the study of its future evolution.

Phase 2: invariant discovery

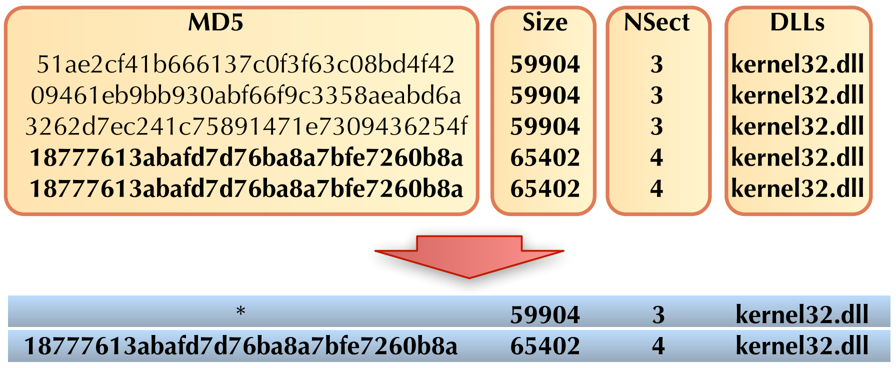
For each of the features defined in the previous phase, the algorithm searches for all the *invariant* values. An invariant value is a value that is not specific to a certain attack instance (it is witnessed in multiple code injection attacks in the dataset), that is not specific to a certain attacker (the same value is used by multiple attackers in different attack instances) and is not specific to a certain destination (multiple honeypot IPs witness the same value in different instances). An invariant value is, therefore, a "good" value that can be used to characterize a certain event type.

The definition of invariant value is threshold-based: Throughout this work, we consider a value as invariant if it was seen in at least 10 different attack instances, and if it was used by at least three different attackers and witnessed on at least three honeypot IPs. Table 4.1 shows the amount of invariant values discovered for each dimension.

Phase 3: pattern discovery

Once the invariant values are defined on each attack feature, we look at the way by which the different features have been composed together, generating patterns of features. As exemplified in Figure 4.2, we define as pattern a tuple $T = v_1, v_2, ..., v_n$ where n is the number of features for the specific EPM dimension and v_i is either an invariant value for that dimension or is a "do not care" value.

The pattern discovery phase looks at all the code injection attacks observed in the SGNET dataset for a certain period, and looks at all the possible combinations of dis-



MD5	Size	NSect	DLLs
51ae2cf41b666137c0f3f63c08bd4f42	**59904**	**3**	**kernel32.dll**
09461eb9bb930abf66f9c3358aeabd6a	**59904**	**3**	**kernel32.dll**
3262d7ec241c75891471e7309436254f	**59904**	**3**	**kernel32.dll**
18777613abafd7d76ba8a7bfe7260b8a	**65402**	**4**	**kernel32.dll**
18777613abafd7d76ba8a7bfe7260b8a	**65402**	**4**	**kernel32.dll**

*	59904	3	kernel32.dll
18777613abafd7d76ba8a7bfe7260b8a	65402	4	kernel32.dll

Figure 4.2: Pattern discovery starting from a limited number of invariants (in bold)

criminant values for the different features.

Phase 4: pattern-based classification

During this phase, the previously discovered patterns are used to classify all the attack instances present in the SGNET dataset and group them into *clusters*. Multiple patterns could match the same instance: For example, the instance $1, 2, 3$ would be matched by both the pattern $*, 2, 3$ and the pattern $*, *, 3$. Each instance is always associated with the most specific pattern matching its feature values. All the instances associated to the same pattern are said to belong to the same EPM cluster.

4.1.3 Clustering using static and behavioral features

The EPM classification technique described in Section 4.1.2 provides a fast and simple tool to explore the interrelationships between exploits, injected payloads and the resulting malware. By looking independently at exploit, shellcode and malware features, the EPM classification allows us to group attack events into clusters. For the sake of clarity, we will refer from now on to *E-clusters*, *P-clusters* and *M-clusters* to identify all the groups sharing the same classification pattern over the exploit (E), payload (P) and malware (M) dimension respectively.

In parallel to the EPM classification of the malware and of its propagation strategies, we take into consideration the behavior-based malware clustering provided by Anubis [33, 1].

The clustering mechanism of Anubis has been described in detail in [30]. It is a behavior-based clustering system that makes use of the Anubis dynamic analysis system for capturing a sample's behavior. More concretely, the system works by comparing two samples based on their behavioral profile. The behavioral profile is an abstract representation of a program's behavior that is obtained with the help of state-of-the-art analysis techniques such as data tainting and the tracking of sensitive compare operations. The clustering system is scalable by avoiding the computation of all $O(n^2)$ distances.

Clusters of events associated to the same behavior according to Anubis behavior-based clustering will be referred to as *B-clusters*.

In the next session, we explore the information at our disposal in the SGNET dataset by means of EPM relations. We show how the combination of approaches based on static or behavioral characteristics of malware samples can be of great help in obtaining a better understanding of the different threats. Finally, we underline the value of the contextual information on malware propagation to add semantics to the different malware groups and acquire insights about the modus operandi of the malware writers.

4.2 Results

For this work, we analyzed all information that was collected by our SGNET deployment in the period from January 2008 to May 2009. During this period, the deployment collected a total of 6353 malware samples, 5165 of which could be correctly executed in the Anubis sandbox. This is consistent with the information reported in [38]. Note that due to failures in Nepenthes download modules, some of the collected samples, unfortunately, are truncated or corrupted, and, as a consequence, cannot be analyzed by a dynamic analysis system.

4.2.1 The big picture

By running the EPM classification technique described in Section 4.1.2, we have discovered a total of 39 E-clusters, 27 P-clusters and 260 M-clusters corresponding to the same number of groups. The analysis of the behavioral characteristics of the 5165 samples led to the generation of 972 B-clusters.

Figure 4.3 graphically represents the relationships between E-, P-, M- and B-clusters. Starting from top to bottom, the first layer of groups corresponds to the exploits, the second to the payloads, the third to the malware grouped according to static information, and the last one to malware grouped according to its behavior. Because of space limitations, we have represented here only the E-, P-, M-, and B- clusters grouping to-

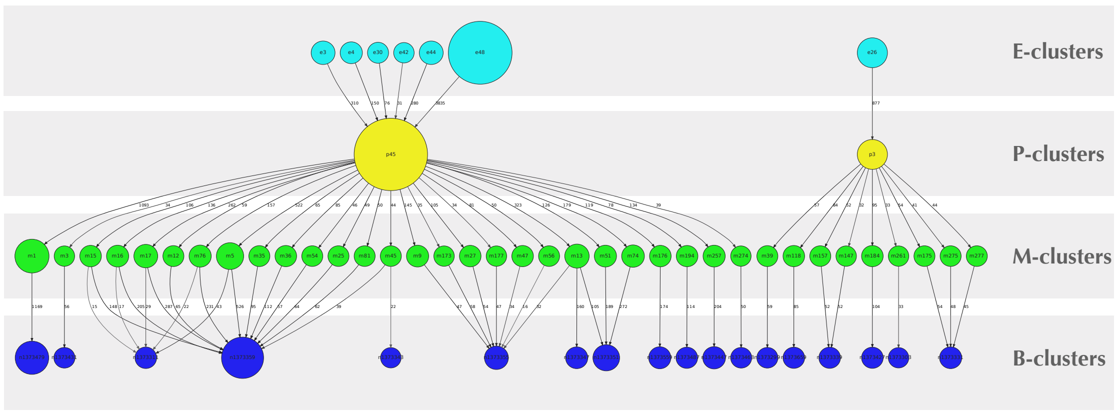


Figure 4.3: EPM relationships in the SGNET dataset and their comparison to B-clusters

gether at least 30 attack events. Figure 4.3 is therefore a simplified, yet, representative view of the reality. We can identify a set of interesting facts:

- The number of exploit/payload combinations is low with respect to the number of different M-clusters. Most malware variants seem to be sharing few distinct exploitation routines for their propagation.
- The same payload (P-cluster) can be associated to multiple exploits (E-clusters).
- The number of B-clusters is lower than the number of M-clusters. Some M-clusters are likely to correspond to variations of the same codebase, and, thus, maintain very similar behavior.

In the following section, we dig deeper into some of the identified relations to underline the value of considering different standpoints.

4.2.2 Clustering anomalies

Clustering techniques relying on behavioral features are known to be subject to misclassification due to anomalies in the extraction of these features. On the one hand, the amount of variability in the behavioral profiles of some samples and their interaction with clustering thresholds may lead to clustering artifacts. On the other hand, the sample behavior may be affected by external conditions (e.g., availability of C&C servers) that affect its profile and can lead to misclassifications.

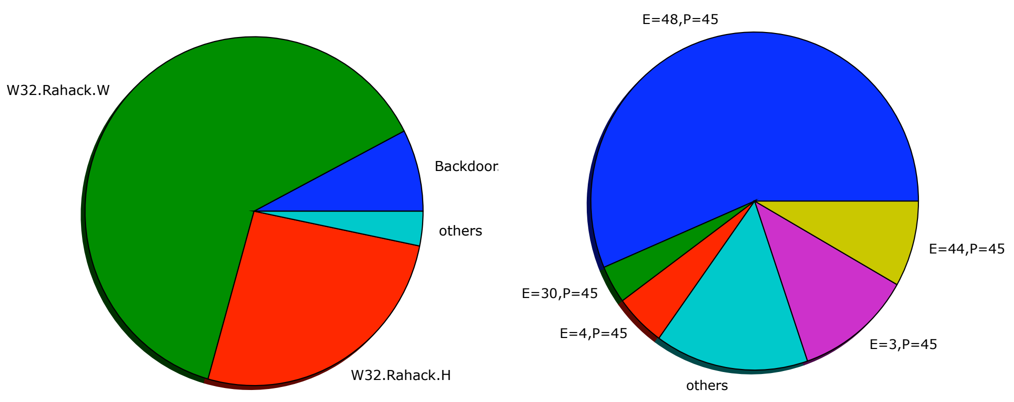


Figure 4.4: Characteristics of the size-1 clusters. On the left, AV names associated to the malware samples by a popular AV vendor. On the right, propagation strategy used by the samples in terms of combination of E and P clusters.

These problems are potentially very difficult to identify. In this section, we show how the combination of clustering approaches based on both static and dynamic information can be helpful in systematically identifying and filtering out clustering anomalies.

When going into the details of the relationships between M-clusters and B-clusters (details that do not appear in Figure 4.3 because of the filtering choice), we discovered a large number of B-clusters composed of a single malware sample. 860 B-clusters out of 972 are composed of a single malware sample and are associated to a single attack instance in the SGNET dataset.

Comparing these size-1 B-clusters with the associated M-clusters, we can identify a small number of size-1 B-cluster having a 1-1 association with a static M-cluster. These cases are likely to be associated to infrequent malware samples whose infection was observed by the SGNET deployment a very limited number of times.

In most of the other cases, instead, the size-1 B-clusters are associated to larger M-clusters, mostly associated to at least another, larger B-cluster. By manually looking at the behavioral profiles of the samples affected by this anomaly, we could not discern substantial differences from the behavior of the larger B-cluster. We believe these clusters to have been incorrectly clustered, a bias in clustering likely to be associated to the employment of supervised clustering techniques (single linkage hierarchical clustering) in Anubis clustering [30].

This is corroborated by other information available in the SGNET dataset: Figure

4.4 shows information on the names assigned to these misclassified samples by a popular AV vendor (left), and on the propagation strategy observed in the SGNET dataset in terms of EP coordinates (right). Most of the samples are classified as being different variants of the Rahack worm (also known as Allaple), and have been all pushed with a very specific P-pattern, P-pattern 45. This P-pattern is characterized by a PUSH-based download, in which the attacker forces the victim to receive the sample on a specific port, TCP port 9988. For all these reasons, all these samples seem to be highly related.

This anomalous condition would have been impossible to detect by looking solely at B-clusters: it would not have been possible to discern behaviors uniquely associated to a specific malware sample from this type of misclassification.

We have also detected more subtle clustering anomalies generated by changes in the context of the malware execution. For instance, M-cluster 13 is a polymorphic malware associated to several different B-clusters. The cluster is characterized by a very specific size, number of sections, and even by specific names for the different sections as it is possible to see by looking at the pattern invariant features:

```
{
   MD5='*', size=59904,
   type='MS-DOS executable PE for
         MS Windows (GUI) Intel
         80386 32-bit',
   machinetype=332,
   nsections=3,
   ndlls=1,
   osversion=64,
   linkerversion=92,
   sectionnames='.text\x00\x00\x00,
                 rdata\x00\x00\x00,
                 .data\x00\x00\x00'
   importeddll='KERNEL32.dll',
   kernel32symbols='GetProcAddress,
                    LoadLibraryA'
}
```

Seeing the rather large number of invariant features (only the MD5 hash is associated to a "do not care" field), we can, therefore, say that the cluster is rather specific.

This cluster is peculiar for the amount of similarities, and yet differences, with the behavior normally associated with the Allaple worm. Looking at the EPM classification, this M-cluster shares the same propagation vector as the samples classified by AV vendors as Allaple/Rahack. The associated pattern in the M dimension is slightly different though: different size, linker version and differences in the Kernel32 symbols. In both cases, the samples are polymorphic: the MD5 is not an invariant feature of the cluster. Interestingly, the polymorphic pattern is different: according to the SGNET data, Allaple mutates its content at each attack instance, while malware samples belonging to the M-cluster 13 seem to mutate their content according to the IP address of the attacker. The same MD5 hash appears multiple times in multiple attack instances generated by the same attacking source towards multiple honeypots, but because of the relevance constraints defined in Section 4.1.2, the hash is not chosen as a relevant feature.

Looking manually at the different behavioral reports for some of the behavioral clusters, we see that a number of samples exhibit different behavior because of environmental conditions. Upon execution, they try to resolve the name “iliketay.cn”, and download from that domain additional components to be executed. Among the different behavioral clusters, one cluster is characterized by the download and execution of two separate components from the malware distribution site, while another cluster is characterized by the successful download of a single component. In both cases, the activity leads to the connection to an IRC server, that provides commands instructing the process to download additional code from other web servers. In a different behavioral cluster, the DNS resolution for the name “iliketay.cn” is unsuccessful: the entry was probably removed from the DNS database[2].

This real-world example shows the usefulness of clustering based on static features in pinpointing clustering aberrations produced by the analysis of behavioral profiles. Samples that are apparently unrelated when purely looking at their behavioral profiles, are discovered to be related when analyzing their static characteristics. Once detected, these problems can be fixed by, for instance, re-running the misconfigured samples multiple times. We have experimentally seen on a limited number of samples that re-execution is indeed very effective in eliminating these anomalies. While the generation of multiple behavioral profiles for all the samples would be too expensive and probably unnecessary in most of the cases, the comparison with static analysis techniques allows to detect small groups of samples for which it would be useful to repeat the behavioral analysis to improve consistency.

4.2.3 The value of the propagation context

When looking at Figure 4.3, we can identify a considerable amount of cases in which a B-cluster is split into several different M-clusters. In order to get a better understanding

[2]As of today, any resolution attempt for the above name fails, and the name appears in many popular blacklists

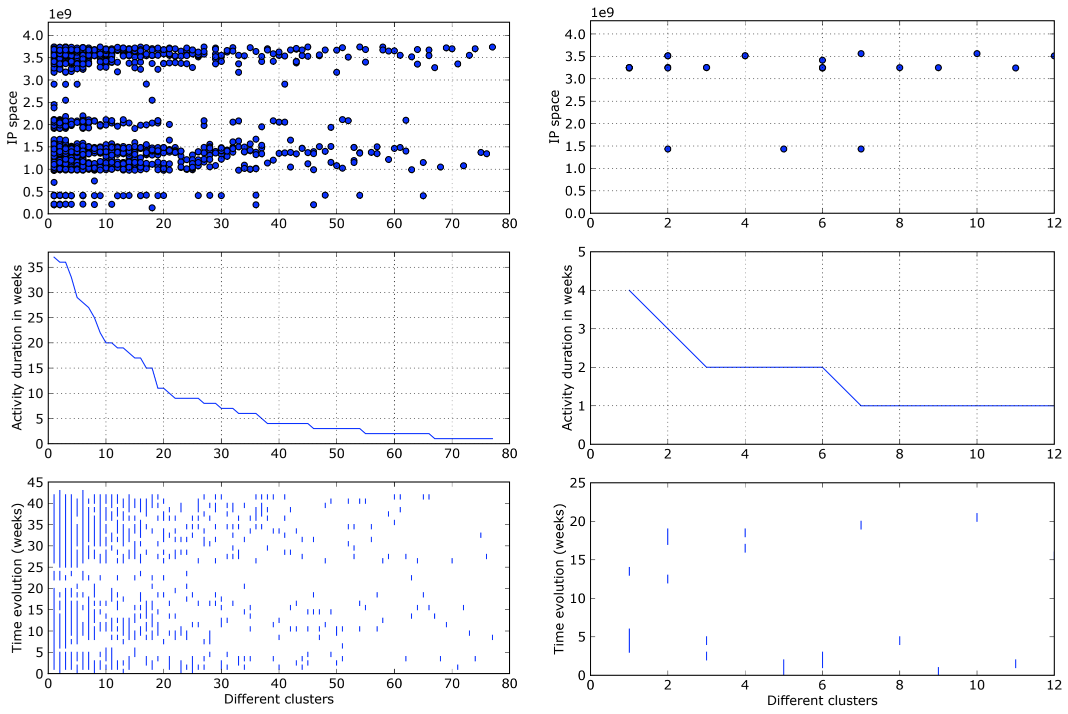


Figure 4.5: Propagation context information for two behavioral clusters.

on the motivations underlying these splits, we have selected two of the biggest B-clusters associated to multiple M-clusters.

The behavioral profiles generated by Anubis are obviously limited in time. At the time of writing, each behavioral profile corresponds to an execution time of four minutes. While such a behavioral profile includes information on the generated network traffic, this information is not sufficient to understand the dynamics of the malware propagation. This is especially true for bots, whose behavior depends on external commands generated by a bot-herder according to his will. Honeypots are instead helpful in better understanding and studying these dynamics.

Figure 4.5 shows contextual information for the two B-clusters under consideration. The X axis splits the B-cluster into the different M-clusters associated with it, while the different graphs show, from top to bottom, the distribution of the infected hosts over the IP space, the number of weeks of activity of the different classes, and the timeline of activity.

In the first case, we see that the different malware variants are associated with different population sizes in terms of infected hosts scanning the Internet. The different population sizes, combined with the small coverage of the SGNET deployment in terms of the number of monitored IPs (at the time of writing, 150 IPs are monitored by the deployment in 30 different network locations), makes the smaller groups account for only a few hits in the dataset. Nevertheless, the distribution of the infected hosts over the IP space, as shown in Figure 4.5, is very similar and generally widespread over most of the IP space. The specific case taken into consideration is indeed a cluster composed of multiple Allaple variants. Allaple is a self-propagating worm exploiting the Microsoft ASN.1 vulnerability (MS04-007) to propagate to its victims, infect HTML files present on the local system and carry on simple DoS attacks against specific targets. The analysis of the Allaple infection revealed the existance of several modifications and improvements to the original code [82], but due to the lack of a Command & Control channel, the worm lacks self-updating capability. Therefore, hosts infected by the different versions and patches of the original codebase coexist on the Internet. While all these different modifications lead to the generation of two different B-clusters, they generate almost 100 different static clusters. One of the main differentiation factors among the different classes is the file size: while the polymorphic routine used by Allaple modifies the binary content without modifying the size, we can detect in the SGNET dataset a variety of M-clusters, all linked to the same B-clusters, but characterized by different binary sizes. In some cases, the different variants also have different linker versions, suggesting recompilations with different versions of the compiler.

Looking at the right side of Figure 4.5, we can immediately identify important differences in the size of the infected populations, and on their distribution over the IP space. In this case, we face rather small populations in terms of numbers of attackers, with most of the static clusters associated to very specific networks of the IP space. All the activities are bursty, and in some cases, the temporal evolution exposes complex behaviors that we consider likely to be associated to bot-like coordinated behavior. For instance, looking in depth at the temporal evolution of one of the M-clusters, we obtain the following sequence:

- 15/7 - 16/7: observed hitting network location A
- 18/7: observed hitting network location B
- 26/7: observed hitting network location B
- 2/8-4/8: observed hitting network location A
- 27/9: observed hitting network location B

Such coordinated behavior suggests the presence of a Command&Control channel.

We have, therefore, looked closer into the behavioral profiles for these samples, and have tried to gather evidence of the presence of an IRC Command & Control server. While not all the samples were executed by Anubis during the activity period of the C&C server, we have been able to associate some of the M-clusters taken into consideration to the corresponding IRC server. Table 4.2 shows how, in most cases, each M-cluster is characterized by connections to a specific IRC server. In the minority of cases in which distinct M-clusters operate on the same IRC channel and receive commands from the same room name, they are likely to be associated to different code variants or "patches" applied to the very same botnet. But even when looking at M-clusters operating on

Server address	**Room name**	**M-clusters**
67.43.226.242	#las6	282, 70
67.43.232.34	#kok8	263
67.43.232.35	#kok6	23, 277
67.43.232.36	#kham	170
67.43.232.36	#kok2	37
67.43.232.36	#kok6	195, 275
67.43.232.36	#ns	234
72.10.172.211	#las6	266
72.10.172.218	#siwa	140
83.68.16.6	#ns	112

Table 4.2: IRC servers associated to some of the M-clusters

different IRC channels, the characteristics of the IRC channels reveal very strong similarities: many IRC servers are hosted in the same /24 subnet, and send commands to the bots from rooms with recurring names or name patterns. This suggests the operation of a specific bot-herder or organization that is maintaining multiple separate botnets.

The combination of malware clustering techniques (i.e., based on both static and dynamic features) with long term contextual information on their evolution is helpful in practice. That is, such techniques allow us to understand better how malware is developed and propagated. Our work shows the importance of leveraging different information sources for studying the threat evolution, and the "economy" of its driving forces.

4.3 Conclusion

In this experiment, we evaluate and combine different clustering techniques in order to improve our effectiveness in building intelligence on the threats economy.

We take advantage of a freely accessible honeypot dataset, SGNET, and propose a pattern generation technique to explore the relationships between exploits, shellcodes and malware classes while being resistant to the current level of sophistication of polymorphic engines. Despite the simplicity of the approach and the easiness with which it could be evaded by malware writers, we show that the current level of sophistication of polymorphic techniques is very low and that simple clustering techniques based on static features often work well in practice. Furthermore, we show the importance of these techniques in detecting and "healing" known problems in dynamic analysis, such as the dependence of the execution behavior on external conditions (e.g., such as the availability of a command and control server).

Moreover, we offer insights into the relationships between different malware classes, their propagation strategies and their behavioral profiles. We show with practical examples how different techniques offer different perspectives over the ground truth. The propagation context allows to build semantics on top of the malware clusters, providing information that is not easily available through standard dynamic analysis techniques. We show how the propagation vector information can be used to study code-sharing taking place among malware writers, and how temporal information and source distribution can be effectively used to provide semantics on the threat behavior to the security analyst.

5 HoneySpider Network (NASK)

As reported in Deliverable D18, NASK's work in WP4 concentrated around contextual features identified in the HoneySpider Network (HSN) dataset. In this chapter we will present two separate advancements related to this dataset which are direct consequences of involvement in the WOMBAT project.

The first section of this chapter covers the HSN URL-Map visualisation tool described earlier in Deliverable D18. The section aims to evaluate the usefulness of the tool by presenting how it was applied to some real cases of malicious behavior.

The second section presents the Capture False Positives Reduction tool, which was developed to enhance the quality of results provided by HSN. The research leading to its development was sparked by difficulties encountered during integration of HSN with FIRE carried out as part of WP5, hence it can be viewed as part of the feedback loop between workpackages 4 and 5.

5.1 Clustering and visualization of HSN results

The visualization tool was developed to aggregate, store and refactor data gathered by the HoneySpider Network system of client honeypots. It aims at presenting new and useful information to aid operations of Incident Handling Teams and HSN system operators. The software presents the processed data in a graph form allowing quick assessment of threat and launching an investigation if it's serious.

5.1.1 Preliminary tests and results

The visualization tool was tested on a database snapshot of the HoneySpider Network system. The database contained about 40000 of URLs among which about 500 was found malicious. The data gathered by the HoneySpider Network system is structured in a tree-based graph with the root representing the main URL posted for processing. The tool gathers all the data and connects the tree-based graphs into a map of redirections between URLs. This leads to creation of maps of interconnected URLs, sometimes creating maps of many different domains redirecting to a single one. In case of malicious URLs, which are the main area of research, compromised sites usually redirect to one or a small fixed number of malicious landing sites. During tests many graphs showing how compromised websites redirect to malicious servers were found. The following are example cases of investigated and confirmed malicious activity.

Case 1: Compromised Polish websites redirecting to external malicious domain

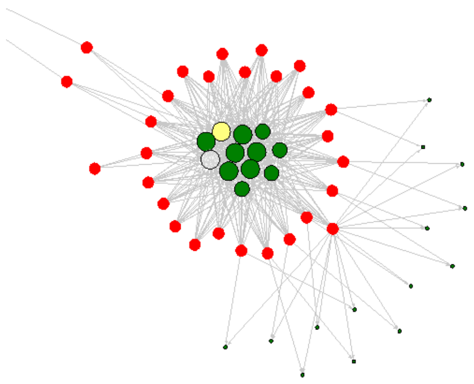


Figure 5.1: URL map of compromised Polish websites redirecting to external domain classified as suspicious

The Case 1, as presented in Figure 5.1, was investigated and confirmed to be a real threat. Each of the red nodes represents a URL classified as malicious. The yellow one is the website where all the compromised websites redirected to, and the green ones are benign requests (other website content like graphics). The website marked in yellow (`http://alexandrepatoworld.com/images/image.php`) was found to inject a JavaScript code dropping an IFrame which redirected further to another malicious domain (`http://iss9w8s89xx.org/in.php`). On the graph it is marked in gray, because the URL quickly stopped responding probably due to blacklisting of the IP addresses the scan was made from. The compromised websites came from two different domains, spread on different subdomains. The infection was made by injecting a request for external JavaScript into the code of each website. The type of infections represented by this graph are the most common ones.

Case 2: Wide-scale compromised websites

Next case (Figure 5.2) presents results of classification of four distinct sets (hand-marked with blue rectangles) of URLs which belong to four different domains. The URLs were

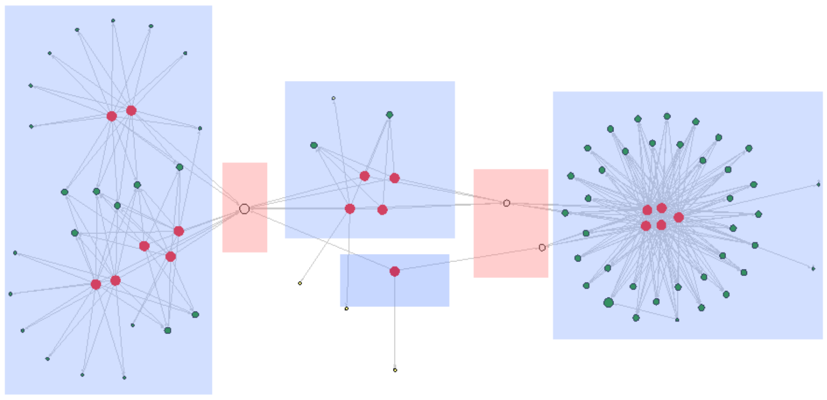


Figure 5.2: Wide-scale compromised websites graph

confirmed to be malicious and were compromised by injecting JavaScript code redirecting to external domains. The domains JavaScript redirected to (`http://hornalfa.com/in3.php`, `http://oughwa.com/in4.php` and `http://sodanthu.com/in6.php` – hand-marked with red rectangles) are gray nodes in the graph because the URLs are no longer active. The example differs from the previous case, because more than one redirecting URL was used, sometimes even for URLs compromised in the same domain. The observed attack could also be a part of a malicious operation on a much larger scale because not only Polish domains were affected by it. HSN classified one non-Polish domain marked in the middle bottom part of graph as malicious. It also used two different redirection sites for the infection purposes.

The tool proved to be very useful, especially in quick interpretation of results produced by the HoneySpider Network system. The HSN system aims at bulk processing thousands of URLs of which many are classified to be malicious and need further investigation. Representing gathered data as graph is a natural way of mapping URL relations and significantly shortens the amount of time needed to discover the specific one responsible for infection or redirection to malicious server. Without help of the visualization and correlation software, the HSN operator is forced to manually investigate each of URLs entered for processing and create a kind of similar representation by hand. The tool simplifies it and allows to quickly find the root cause of infection without manually traversing the URL redirection tree. Also, when many websites are found to be compromised and redirect via the same URL it is much easier to notice it on the graph, than manually correlate the HSN system results.

5.1.2 Statistics of the observed URL map

The dataset used to test the visualization tool was obtained with the HoneySpider Network system and contained about 40000 URLs. The visualization software took into account only URLs classified as malicious or suspicious which was about 5000 URLs. This amount of URLs translated into about 21000 distinct correlated requests visualized on the URL map in Figure 5.3.

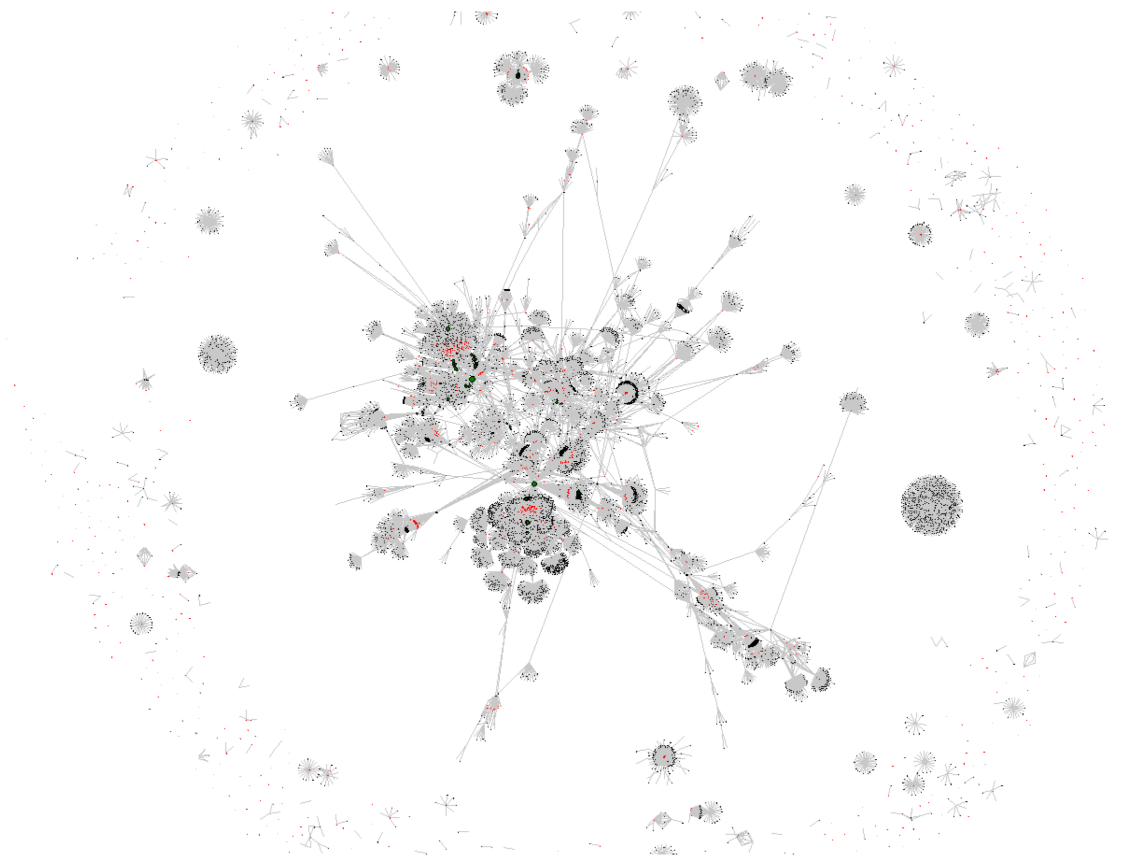


Figure 5.3: URL map of 5000 potentially dangerous URLs together with all requests made during crawling

The main graph presents an extensive network of interconnected nodes proving that even between seemingly not related URLs in most cases exists at least one common point. In the biggest cluster presented on the map at least 4 central points can be identified. These are:
`http://www.google-analytics.com/ga.js`,
`http://pagead2.googlesyndication.com/apps/domainpark/show_afd_ads.js`,
`http://pagead2.googlesyndication.com/pagead/show_ads.js` and
`http://home.hit.stat.pl/cachedscriptxy.js`. If one of them ever became malicious many of the existing Internet websites would also indirectly lead to infection of a users computer.

Table 5.1: Classification statistics

Classification	Benign	Suspicious	Malicious	Not active
Count	24283	4414	473	12408

During tests many of the classified URLs was found to be inactive, as can be seen in Table 5.1. This can be the result either of blacklisting the IP addresses if the scan was done multiple times or because of short life-span of malicious domains. Usually domains of this kind are registered as 'domain tasting' for the time of two weeks after which the domain stops resolving.

Many of the processed URLs gave HTTP error 404 or 500 or the web-page contained some suspicious JavaScript code. The HSN was able to obtain it but not always perform full analysis because of extreme obfuscation techniques used to hide the true actions of the compromised website. In such cases the URL is represented usually by a single node. The nodes with degree equal to 0 can be seen on the edge of the URL map. These are usually inactive domains, URLs returning HTTP errors or websites with unrecognized JavaScript codes which were not classified as malicious by either low or high interaction component of HSN.

The histogram of graph node degrees (Figure 5.4) shows the most common types of nodes on the URL map. There are only a handful of nodes with very high degrees being intersection points on the map, the highest being one node with degree of 1058. Existence of such nodes representing URLs being referred to creates a potential threat of mass scale exploitation if one of these websites is compromised.

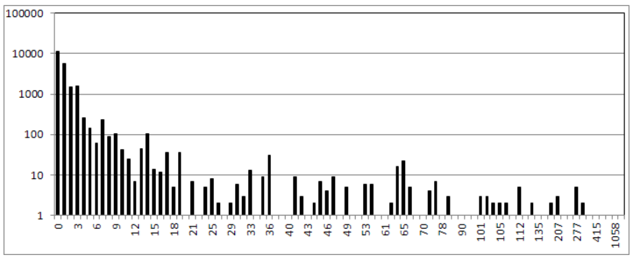


Figure 5.4: Histogram of node degree on the URL map in logarithmic scale (X-axis: node degree, Y-axis: number of nodes)

5.1.3 Future work

The tool is fully functional, but still in active development. Improvements are especially needed in the user interaction functionality. Based on feedback received from IRT the tool will be equipped with a notification module. The module will produce alerts on new discovered malicious websites or forming clusters which are compromised websites redirecting through common URLs. The clusters will be marked on the graph, which will ease the process of analysis. Also search mechanisms will be improved to allow easily finding the desired information on the URL map and a possibility to remove unwanted nodes from the map (e.g. to temporarily break up the huge clusters created by connections via analytics and advertisement URLs) will be added.

5.2 Reduction of false positives in high interaction module of HSN

To show the reasons behind this part of our work, it is necessary to start with a short explanation of the structure of the HoneySpider Network client honeypot system developed by NASK. The central module of the system, called simply HSN Center, handles the workflow, GUI and several minor data enrichment functions (e.g. geolocation). It does not, however, perform the essential functionality of a client honeypot – detection of malicious webpages. This is done in detection modules. The two main detection modules of HoneySpider Network are LIM and HIM.

LIM is the low interaction client honeypot. It is essentially a web crawler, equipped additionally with several detection methods which work on the downloaded content. Since the malicious code is not actually running (there is nothing to exploit), the decision whether the URL is malicious or not must use static analysis and is mostly heuristic – there is a possibility of rule-based or signature-based detection of known threats, but these are not very effective. In fact, LIM does more than just static analysis – some parts of the content are actually executed, e.g. JavaScript. This is done as part of browser emulation, to find URLs in obfuscated scripts, etc. Still, LIM is not the natural environment for this code, so its behavior is not fully representative.

While LIM provides a fast, but deep analysis of the content – analysing the request structure, identifying the suspicious requests, etc. – HIM was planned as a complementary subsystem. HIM is a high interaction honeypot, based on Capture-HPC [71]. It works by opening the URLs in a real browser inside a virtual machine, registering the behavior of the system and performing a simple anomaly analysis. If any abnormal activity is recorded, the URL must be considered malicious. This approach is slower and requires more hardware, but in theory it should validate the classification performed by LIM. Of course, both false positives and false negatives are still possible. The virtual machine may lack a specific version of some plugin which is exploited by the scanned page, or some rare but benign activity may occur, which is not listed as allowed.

After deployment, the level of false positives proved very hard to control. This meant that HIM could not be trusted by default – manual verification was necessary before any action would be taken. Still, this did not severely limit the usability of the system in our applications.

During our work on integrating HSN with FIRE (part of WP5) we have encountered a serious problem with the identification of URLs to be passed to FIRE. The selection of URLs takes into account both HIM and LIM classification. However, unlike in everyday operation of HSN, FIRE is not interested in the URL originally submitted to HSN, but the URL of the actual exploit. We have designed good algorithms to extract that information from LIM results. However, before the result could automatically be passed to FIRE, we needed to be sure, that the URL was indeed malicious. HIM results would be very useful here – unfortunately, the level of false positives was far too high to make this workable. It became clear, that we needed to find a way to reduce this problem if the results are ever to be used in a fully automatic way.

The situation is a good example of the feedback loop between workpackages 4 and 5. The results of work in WP5 made it clear, that data quality can and should be increased. The modification is clearly part of WP4 – as described in the next sections, we are using the information gathered while scanning other URLs to enhance the results for new URLs.

The following sections ignore the LIM, Center and other modules, concentrating on HIM and – more specifically – on the new module added to reduce the false positives rate.

5.2.1 Design and implementation

The general structure of the high interaction component of HSN is presented in Figure 5.5. The main detection engine is a modified version of the Capture-HPC client honeypot system. HIM consists of the following elements:

HIM Manager This is the module responsible for communication with the Center, HIM queue management, temporary storage of results and correlation of scans with results from the network monitoring module (NMM) and antivirus module (AVM), which monitor the traffic generated by HIM and apply traditional, rule- and signature-based detection methods to it to enrich the results.

Capture Server This is a modified version of the Capture-HPC server module. Its tasks include communication with the Capture Client modules and management of the virtual machines, including reverting the machines to a clean state after each detected infection.

Capture Client This part of the system runs inside a virtual machine and consists of two parts – the drivers, which include hooks into the system libraries and record all file, process and registry actions in the system, and the actual capture client program, which communicates with the Capture Server, drives a web browser to URLs received from the server, filters the registered actions using a black/white (B/W) list, keeps copies of all files modified during the visit to a URL and finally sends the results to the server, including actions not allowed according to the list and a compressed archive with the modified files.

Log Classifier The newly introduced log classifier is an additional processor of the results provided by Capture (Server + Client).

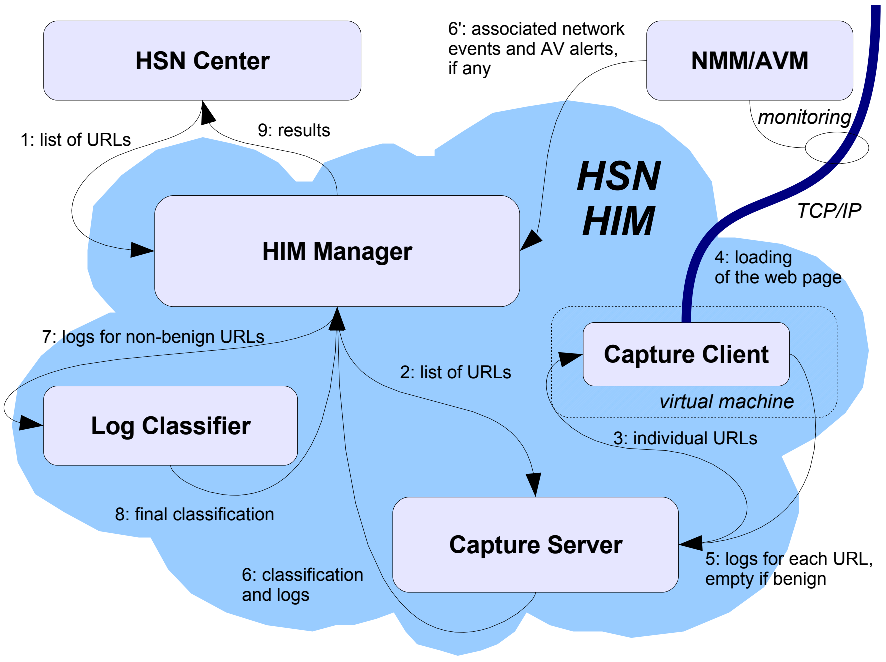
In the following discussion we mostly ignore the HIM Manager, as it is not directly involved in the detection process. We will treat the Capture Server and the Capture Client as a single entity – a black box providing lists of actions for URLs given on input – and we will refer to it as CP. We will refer to the Log Classifier as LC.

On this level of abstraction the data flow inside HIM is linear: the input of the system is a list of suspicious URLs, the URLs are processed by CP and the results are passed to LC for post-processing. The output of LC is the final output of the system.

CP – the client honeypot

CP is the actual client honeypot – the detection engine of the presented system. The solution is essentially Capture-HPC, although several modifications were applied, includ-

Figure 5.5: Cooperation of modules of HSN HIM



ing a change of virtualization technology and numerous bugfixes increasing the stability of the solution. The active part of CP, responsible for actual detection of malicious behaviour, is running inside a virtual machine using Windows as the guest operating system. CP drives a real web browser to visit each URL submitted to the system, recording actions – calls to some of the operating system functions.

The recorded actions include file operations, process operations and registry modifications. All of these types of actions are normally performed during web browsing. The difficult task of separating the malicious actions from benign ones is handled by B/W lists using regular expressions. These lists contain templates of normal, benign actions expected while visiting a URL. CP compares each action with the list, in parallel with their normal execution. Matching actions are ignored, others are reported to the server part of CP (running in the host system or on a different computer). These entries form a log file which is the result of analysis. After the page fully loads and a few seconds pass, the analysis is complete and the next URL can be processed. If any entries appeared in the log, then the URL is considered probably malicious and the virtual machine is reverted before proceeding to the next URL to avoid contamination of the results of the next scan with the effects of possible previous infections.

False positives in this setting are a consequence of imperfect B/W lists – they cannot predict all possible benign actions, like automatic updates of different software components. The lists are very hard to prepare – Windows is a quite complicated system and to foretell every possible benign action is almost impossible. The task is made even more difficult by the timing constraints of CP – if too many regular expressions are specified, then CP will fail to capture some of the actions, since they will be reported faster than they can be processed. Limiting the length of the list by using more general regular expressions results in false negatives. In other words, there is no perfect solution. Capture-HPC comes with default lists, which eliminate the most frequent benign actions. We have developed new lists, customized for our configuration, eliminating many other benign actions, but false positives are still relatively frequent.

Our approach to solving this problem is to process the logs generated by CP using a trained LC, which classifies the logs into two groups: malicious and benign. The rest of the paper concentrates on this module.

Implementation details

The Capture Client module, including the drivers responsible for capturing actions, is implemented in C++, while the Capture Server and the entire LC are implemented in Java.

LC uses Weka based classifiers [86]. Neither training nor classification are performed directly on logs received from CP – they work on binary vectors of features extracted from the logs in a preprocessing phase. The features can be quite general (e.g. "does the log contain entries reporting writing to a file", or "how many process creation entries are included in the log"), but they have to be specified manually – for each feature a class capable of identifying its presence in a log must be implemented. Still, the management of features is quite comfortable – they have a well defined interface and exist in a form of java *.jar* packages. Using reflection, sets of features can be easily switched without any changes in the implementation, making the solution flexible.

Format of logs generated by CP

Logs are plain text files, containing a list of system actions captured by CP while visiting websites (URLs). The actions are divided into three main groups: File (with operations: Write, Delete, Read, Open), Process (Created, Terminated), Registry (OpenKey, CreateKey, CloseKey, EnumerateKey, EnumerateValueKey, QueryKey, QueryValueKey, SetValueKey, DeleteValueKey). Listing I presents an example of a typical malicious log.

It can be seen in the log that *IEXPLORER* loads an *exe* file and then executes it.

Listing I: Example of a log generated by CP (full paths are replaced with dots)

```
"file","C:\...\IEXPLORE.EXE","Write","C:\...\LocalSettings\ ↩
Temp\tmp64555.exe"," − 1"
"process","C:\...\IEXPLORE.EXE","created","2384","C:\...\ ↩
LocalSettings\Temp\tmp64555.exe"
"process","C:\...\IEXPLORE.EXE","terminated","2384","C:\ ↩
...\LocalSettings\Temp\tmp64555.exe"
```

5.2.2 Selection of features

The log is analyzed by the classifier as a vector of boolean values, where each element indicates whether a given feature is present in the log file. Many of the features are naturally boolean, while others are numeric and must be discretized before processing. They are replaced by groups of mutually exclusive boolean features – each one is present only if the value falls into an interval defined for that feature.

The features defined in this project can generally be split into three groups:

- Row features. These features do not describe the log as a whole, but individual entries. For example, a set of boolean features could be specified for file operations, identifying the type of file by matching the filename extension to predefined groups. These features are processed before all others, eliminating benign entries and marking the particularly suspicious ones.
- Set-type log features. These features describe the log as a whole, but treat it as a set of entries, ignoring their order. For example, we could define for each of the 15 different operations a boolean feature reporting the occurrence of that type of operation in the log or a numeric feature counting these occurrences.
- Sequence-type log features. These features interpret the log as an ordered sequence of actions (which it is) and look for specific combinations of actions. A feature of this type usually corresponds to some typical malicious behaviour and is almost always boolean.

During our experiments we have tried different sorts of features, but the initial results quickly led to the conclusion that the row features are not very useful and may in fact interfere with log features by removing rows which seem benign on their own, but provide additional information. Since the amount of included features affects the quality and speed of training and classification, we also removed most of the generic set-type features, leaving only the ones most useful in classification. As a consequence, almost all of the features in the final set are directly interpretable as occurrences of some suspicious behavior.

The manual specification of features may seem similar to defining B/W lists for the CP module. However, the entries in the lists are defined per action and absolute – a given action is either benign or malicious. The features defined for LC can analyze combination of actions. They also do not have to provide final classification – their actual usefulness is determined during the training phase, depending on the composition of features appearing in the logs from the training set.

Most features used in our project were created as direct result of a manual analysis of about 60 different logs classified as malicious by an expert. They represent typical behaviours, common for many types of malware. Some others were added based on our general experience, even though they did not appear in these samples (e.g. features C, G and F described below). The following list shows the features we use:

Feature A Writing files of a specific type (e.g. exe, sys, dll, bat, tmp, vbs, com, log, tmp) to special folders (listed in Table 5.2). The special folders are often used by malware software. These are system folders or folders where normal programs do not write anything at all. A larger list of special folders can be found in [65].

Feature B A log file contains a specific sequence of activities:

Feature B.a Write→Created – some file is saved to a hard disk (e.g. by Internet Explorer) and then the same file is started as a process (e.g. by Internet Explorer).

Feature B.b Created→Registry – some program is started as a process and then the same program writes to the system registry.

Feature B.c Write→Created→Registry – some file is saved to a hard disk, some process is started, some process writes to the system registry.

Feature C Firewall bypass by editing registry keys is listed in Table 5.3.

Feature D Deletion of a specific file type (e.g. evt, log).

Feature E Hiding a file extension in a filename (e.g. file.exe.txt).

Feature F Specifies the number of different processes started within a single log file:

- **Feature F.a** one process
- **Feature F.b** two processes
- **Feature F.c** more than two processes

Feature G Some popular program (e.g. RealPlayer, Java):

- **Feature G.a** reads any files
- **Feature G.b** writes any files
- **Feature G.c** deletes any files
- **Feature G.d** uses system registry

Table 5.2: Folders often used by malicious software

C:\Windows\System32\
C:\Windows\
C:\Documents and Settings\%\Local Settings\Temp\
C:\Documents and Settings\%\Desktop\
C:\
C:\Documents and Settings\%\

Table 5.3: Registry keys modified to bypass the firewall

HKCU\Software\Microsoft\Windows\CurrentVersion\Internet ↩ Settings\ZoneMap\ProxyBypass
HKCU\Software\Microsoft\Windows\CurrentVersion\Internet ↩ Settings\ZoneMap\IntranetName
HKCU\Software\Microsoft\Windows\CurrentVersion\Internet ↩ Settings\ZoneMap\UNCAsIntranet

5.2.3 Selection of classification algorithm

Several different machine learning algorithms were tested: decision trees (J48, ADTree), support vector machine (SMO), algorithm based on Bayesian network (AODE) and

Table 5.4: Autostart places

C:\autoexec.bat
C:\config.sys
C:\WINDOWS\wininit.ini
C:\WINDOWS\winstart.bat
C:\WINDOWS\win.ini
C:\WINDOWS\system.ini
C:\WINDOWS\system\autoexec.nt
C:\WINDOWS\system\config.nt
C:\Documents and Settings\All Users\Start Menu\Programs\Startup
C:\Winnt\Profiles\All Users\Start Menu\Programs\Startup
C:\Documents and Settings\All Users\Start Menu\Programs\Startup
C:\windows\start menu\programs\startup
C:\Documents and Settings\%\Start Menu\Programs\Startup

nearest neighbor like algorithm (NNge). We ran several tests to examine the efficiency of our solution and its usefulness in a malware detection process. At the same time the aim was to keep the false positives rate very low.

In the tests we used a set of logs manually classified and chosen by our specialists. All logs were obtained from a normally operating HSN HIM – no artificial examples were added. The full set contained 180 logs, including about 60 malicious logs and about 120 benign logs.

Two different sets of B/W lists were prepared for the CP module. The so called default lists are the B/W lists provided with Capture-HPC, after a simple adaptation to the Polish version of Microsoft Windows OS. The expert lists are the lists normally used in the HoneySpider Network – prepared by specialists and extensively tested to obtain a possibly small number of false positives without sacrificing efficiency. The logs in the manually classified set, used for training and – in most tests presented in this section – for testing, were generated using expert lists.

Direct comparison of algorithms

In this test different Weka classifiers were compared using only manually classified logs, with the goal of obtaining the lowest possible false positives rate. The algorithms chosen for comparison were: J48 (C4.5 decision tree), ADTree (Alternating Decision Tree learning algorithm), AODE (Averaged One-Dependence Estimators), NNge (Nearest Neighbor With Generalization) and SMO (Sequential Minimal Optimization). For each classifier 300 tests were performed. Each test used a different training and testing set. The sets were randomly selected from the manually classified set – half of the malicious set and half of the benign set. The remaining logs were used as testing sets.

We trained the classifiers with a whole set of features presented in section 5.2.2. Table 5.5 presents the results for each algorithm. The values given are as follows:

avg. F-P count This column presents the average (over 300 tests) number of benign logs classified as malicious. E.g. while testing the J48 algorithm, 9 such logs were generated in total, resulting in an average of $9/300 = 0.03$ logs per test.

avg. F-N count This column shows the average number of malicious logs classified as benign, computed in the same way as the average F-P count.

max. F-P count This column presents the largest number of logs wrongly classified as malicious in a single test.

max. F-N rate This column shows the highest ratio of malicious logs classified as benign in a single test. The value is computed as the number of misclassified logs divided by the number of malicious logs in that test (due to random selection this varies between tests), both as a fraction (to show both values) and as percentage.

As can be observed in the table there is not much difference between the tested classifiers. The average false positive rate is very low for all classifiers (even equal to zero for AODE and NNge). The average false negative rate determines the ability of the classifier to detect malware. It can be observed that AODE and NNge present the best ability in malware detection over 300 tests. The worst case for false positives is quite satisfactory – in no test more than one log was wrongly marked as malicious. The worst case for false negatives is not as good, but more importantly the log counts in this column are much higher than the averages. This happens, because the random division of the set of logs may sometimes produce low quality training sets. Using larger sets should lower this ratio considerably. Still, since our focus is on reducing false positives, even 14% of false negatives (worst test of J48 – the absolute worst case) is not a serious problem.

Comparison with subsets of features

Since the previous direct comparison of the classification algorithms proved inconclusive, another set of tests was performed, this time omitting selected features, both in the

Table 5.5: Comparison of Weka classifiers

Class. name	avg. F-P count	avg. F-N count	max. F-P count	max. F-N rate
J48	0.03	2.36	1	7/51=14%
ADTree	0.003	1.12	1	6/54=11%
SMO	0.03	1.10	1	6/54=11%
AODE	0.0	0.83	0	5/70=7%
NNge	0.0	0.83	0	6/54=11%

training and testing phase. The analysis of the resulting false positives and false negatives rates would allow us to assess the sensitivity of different algorithms to the quality of available features. Since the malware behaviors are constantly changing, the selected feature set may soon become outdated. While updating this set is a necessary activity, it would be a clear advantage if the selected algorithm would be able to perform reasonably well until new features are identified and implemented. Additionally, the test allowed us to verify the quality of the existing set, identifying features which could be dropped from the set without impacting the classification.

The tests were performed in the same way as in section 5.2.3. The same 300 combinations of training and testing sets were processed for each combination of algorithm and subset of features.

The results of this test are presented in Table 5.6. For completeness, the results with the full set of features are also presented in the first two rows (obviously with the same values as in Table 5.5).

The first conclusion we can draw from the results is that the NNge algorithm is very different than the others – it defaults to malicious classification, while all others default to benign. As an effect, as features are removed, most algorithms show a growing number of false negatives while the number of false positives remains relatively low, while NNge shows the opposite tendency.

The results seem to show that no feature is redundant. The results in the first rows, where all features are examined, are better than most of the others. Two possible exceptions may be found, though.

Results without feature A are comparable to those obtained using all features, although the number of false negatives is significantly higher. This holds for all algorithms, except NNge, which has both false negative and false positive rates much higher than

Table 5.6: Relevance of features – average F-P and F-N counts are presented

Features	Result type	J48	ADTree	SMO	AODE	NNge
all	F-P	0.03	0.003	0.03	0.00	0.00
	F-N	2.36	1.12	1.10	0.83	0.83
$all - \{A\}$	F-P	0.107	0.50	0.34	0.33	2.95
	F-N	3.58	2.38	3.38	3.06	1.62
$all - \{A, B.a\}$	F-P	4.65	3.76	5.69	3.92	8.53
	F-N	6.12	5.75	6.81	8.10	1.15
$all - \{A, B.b\}$	F-P	0.12	0.49	0.33	0.32	2.89
	F-N	3.59	2.40	3.50	3.34	1.62
$all - \{A, B.c\}$	F-P	0.09	0.64	0.16	0.23	4.53
	F-N	3.60	2.97	3.41	3.44	1.62
$all - \{A, B.*\}$	F-P	8.31	8.33	7.93	8.97	17.24
	F-N	5.92	5.07	5.10	6.50	1.18
$all - \{F.*, G.*\}$	F-P	0.00	0.00	0.00	0.00	5.71
	F-N	2.43	1.60	1.99	1.40	0.83
$all - \{A, B.*, G.*\}$	F-P	9.59	12.55	9.72	10.08	32.65
	F-N	17.87	15.38	17.73	17.21	0.73
$all - \{A, B.*, F.*, G.*\}$	F-P	4.12	4.97	4.10	4.97	42.68
	F-N	25.67	25.01	25.65	25.01	0.00

with all the features. We can conclude that feature A is useful, although not crucial.

The features F.* and G.* are a different case. Again, they are absolutely necessary if NNge algorithm is used, but for all others there is a tradeoff – removing these features completely eliminates false positives, but at the cost of a higher number of false negatives – still relatively small, but noticeably higher. This result must be considered inconclusive, as the test is performed on a very small sample. The number of false positives obtained with all features is low enough that lowering it to zero may not be worth the additional false negatives. After all, we could easily reach zero by simply classifying all logs as benign. The considered features do carry valuable information – while they add little when all other features are examined, the last two rows show that when features A and B are ignored, these features are definitely necessary.

The test seems to indicate that the AODE algorithm is the best choice for the final solution, as it performs best with a large set of features and remains competitive as features are removed from the set. One more test was performed to confirm this choice.

Estimation of false positives rate reduction for default lists

The last test comparing the different algorithms aimed to measure the ability to reduce the false positives rate. This time the classifiers were trained using the whole set of manually classified logs. On the other hand, the quality of the logs in the testing set was deliberately lowered by using the default B/W lists in CP. This results in many false positives and generally longer logs with many benign entries. Since the training set used different B/W lists, the logs are also less similar to the ones in the training set.

The testing set included about 5800 presumably benign logs. The logs were obtained by scanning a list of URLs from the `www.alexa.com`. The URLs on this page, being popular, high-traffic sites, are assumed to be benign (although there is a small risk of infection of one or two of the sites). The test was expected to result in most of the logs being classified as benign. Manual analysis of the hopefully small number of remaining malicious logs would then eliminate the rare true positives and give an estimation of the false positive rate reduction factor (compared to CP using default black/white lists). Manual verification of all webpages in the testing set was of course not feasible. Again, we perform the tests with different subsets of features.

Table 5.7: False-positives rates for *www.alexa.com* with different sets of features

Features	J48	ADTree	SMO	AODE	NNge
all	0.73%	0.81%	0.80%	0.73%	0.81%
$all - \{A\}$	0.05%	0.61%	0.57%	0.24%	47.08%
$all - \{A, B.a\}$	3.77%	3.79%	23.25%	21.64%	49.80%
$all - \{A, B.a, B.b\}$	23.97%	23.18%	23.25%	21.97%	70.02%
$all - \{A, B.*\}$	68.5%	68.5%	68.6%	68.5%	71.5%
$all - \{A, B.*, G.*\}$	48.78%	48.78%	48.78%	48.78%	74.95%
$all - \{A, B.*, F.*\}$	21.81%	21.81%	21.79%	21.79%	96.3%
$all - \{F.*, G.*\}$	0.73%	0.73%	0.73%	0.73%	1.42%
$all - \{A, B.*, F.*, G.*\}$	0%	0%	0%	0%	100%

In this test we lack the ability to identify true negatives, but we assume their number to be very low. This lets us count all logs classified as malicious toward the false positive rate. The results, as shown in Table 5.7, are very good – less than 1.0 percent of all 5800 logs were classified as false positives when all features were analyzed. These logs were mostly non-malicious Java and RealPlayer system updates. Note, that all these logs are probably false positives on the CP level – the 1% result is not really the false positives rate of the whole setup, but only the reduction factor of LC.

We have run the test for all classifiers from Table 5.5 and the differences in results were insignificant. Thus, we obtained about 1% of false positives for all classifiers.

We can observe an analogy between Table 5.6 and Table 5.7 according to false positives and false negatives rates. The last column of Table 5.7 shows that classification without features A, B, G and F is totally useless. We got 100% false positives rate for NNge and while the false positives rate for other classifiers is 0%, this result is too good to be true – the previous test already showed that the false negatives rate is extremely high in this case.

While we did not check all the logs in this test, we have looked at some of the ones selected by the LC and found two suspicious enough to warrant examination. Verification confirmed that our LC element managed to find malicious URLs among considered logs. The `www.shooshtime.com` and `www.51edu.com` turned out to serve malware at the time of the test. The sites were ranked respectively in the 4328th and 4553th place in the list of one million `www.alexa.com` top sites. This should be treated as another proof that high popularity does not imply adequately high security and that the alexa dataset does contain true positives.

An interesting result are the impressively good results of the decision tree algorithms (J48 and ADTree) in the test omitting features A and B.a. The algorithms performed quite well throughout the test, making it clear, that selection of the classification algorithm is not entirely clear – depending on the test, all algorithms are capable of achieving good results. As a final result, we decided to use the AODE algorithm, which is consistently reliable in this application. While with a full set of features the NNge algorithm comes very close, its sensitivity to the quality of features makes it definitely a worse choice.

5.2.4 Tests of the implemented solution

The tests described in this section use the final version of the log classifier, based on the AODE algorithm. The classifier is trained on the entire manually classified set and uses all available features. The goal of these tests is to verify the practical value of the LC module.

Performance test

In this case we measured the time that LC needs to classify the logs used in the last test from the previous section. The goal was to verify whether LC can keep up with CP or – preferably – with several instances of CP.

The set of 5800 logs was classified in about 3 minutes and 10 seconds. It means that our solution can classify about 1200 logs per minute – an order of magnitude faster than CP. Thus it is more than fast enough in practical usage of LC element. A single LC module can easily process logs generated by several CPs. The test was run on Intel 2.83 GHz, 2GB RAM personal computer.

Final test on Alexa database

In this test we compare false positive rates obtained for default B/W lists and B/W lists prepared by a security expert. We tested top 30 thousand URLs from `www.alexa.com`. The test was run in parallel on two PCs for default and expert lists. Note that there can be more than one log generated per URL – CP may rescan a given URL if problems were detected during first visit.

In Table 5.8 the dataset names alexa1, alexa2 and alexa3 stand for first, second and third top tens of thousands URLs from `www.alexa.com` list. Each of these sets was independently scanned by two instances of CP – one with the default B/W lists, and one with the expert lists. The first two rows show the number of logs generated by CP and how many of them were classified as malicious by LC.

All logs generated with expert lists and all logs classified as malicious by LC were verified by a security specialist – the number of confirmed malicious logs is presented in the next two rows. Due to the overwhelming amount of logs no attempt was made to estimate the number of true positives in the logs generated by CP with default lists before processing by LC.

The final rows present the rate of false positives before and after processing by LC. Since CP with default lists decided to rescan many URLs and actually generated more logs than URLs, we did not compute the false positives rate for this case – we didn't keep the relation between logs and URLs and it was impossible to verify how many URLs were actually classified as benign. In any case, the false positives rate in this case would be extremely high, nearing 100%. The default lists do shorten the logs dramatically (compared to no lists at all), but are too weak to filter out all benign actions, except maybe for extremely simple web pages.

Table 5.8: Verification on `www.alexa.com` dataset

dataset	alexa1		alexa2		alexa3	
B/W list	expert	default	expert	default	expert	default
CP logs	164	15152	172	12824	183	12862
LC malicious	0	277	4	13	3	9
confirmed CP malicious	0	-	4	-	4	-
confirmed LC malicious	0	31	1	0	3	1
false positives rate CP	1.64%	-	1.72%	-	1.83%	-
false positives rate LC	0.0%	1.62%	0.03%	0.10%	0.0%	0.06%

The set alexa1: 1-10000. Using expert lists, the CP element generated 164 logs. LC classified all of them as benign. Verification by an expert confirmed, that all of the logs were false positives, so in this case we have indeed reached both 0 false positives and 0 false negatives rate – unless of course CP itself failed to identify some true negatives, which is beyond the scope of this research (and which it did, as explained in the next paragraph).

Using default lists, the CP element generated 15152 logs. 277 of them were classified as malicious by LC. Of those, 31 were confirmed by an expert to be malicious logs. Final false positives rate is equal to 1.62%. This result also confirms that CP with expert lists may pay for the lower false positives rate with a higher false negatives rate – there were indeed truly malicious URLs in the set, quite many actually. As stated in the previous paragraph, expert lists did not let CP catch them.

Note that in this case using default lists and LC gives basically the same false positives rate as using expert lists without LC.

The set alexa2: 10001-20000. Using expert lists, there are 4 confirmed malicious logs in 172 logs generated by CP. LC eliminated all but 4 logs, but unfortunately only one of them is really malicious. It means that 3 malicious logs were false negatives in LC.

Filtering of the numerous logs generated by CP with default lists resulted in only 13 logs classified as malicious, all of them really benign.

This leaves us with a very low false positives rate, but unfortunately also with some false negatives. The false positives rate of LC working on results of default lists is even significantly better than that with expert lists.

The set alexa3: 20001-30000. For this set, expert lists generated 183 logs, including 4 truly malicious ones and LC managed to eliminate all false positives, leaving only 3 malicious logs. It means that one malicious log was misclassified, but the overall result is very good.

From the logs generated with default lists, the LC element eliminated all but 9. One malicious log was found in that set. Surprisingly, this is not one of the logs detected with expert lists.

Even more interestingly, when the site `http://kb120.com` was scanned with expert lists, it was properly classified as malicious, but the log generated with default lists for the same site was really benign. The reason could be the limited reporting ability of CP, or intelligent website behaviour detecting CP, or maybe simply randomized serving of malware.

5.2.5 Summary

We showed experimentally that machine learning algorithms are efficient for the purpose of false positives rate reduction with our log format. We achieved a decent minimum false positives rate (and false negatives rate as presented in Table 5.5) for all the classifiers we tested. We showed that the small (about 180 logs) training set that we collected and classified manually can be useful in practical malware detection process. We have selected a set of features and confirmed its usability. We also decided on the classification algorithm, showing that the AODE classifier performs very well in all tests.

The tests show that the false positives rate is also quite low for a real world collection of websites. We did not analyse the false negatives rate for URLs from `www.alexa.com` as we did not know any appropriate tool that would provide reliable classification for such a large collection in a reasonable time (in fact, if we did, we would not need to perform this research at all). In exchange for it we caught some malware from a source we would expect to list only benign websites. The LC module works as intended and can filter out most of the false positives from the stream of logs coming from CP. An unexpected side effect was the conclusion that CP itself generates many false negatives, more surprisingly different ones depending on the lists used. The fact itself is quite natural, but the amount of such mistakes is higher than we thought. Before introducing LC we were unable to observe this, as false positives obstructed the view. It is now clear that the B/W lists and other aspects of CP will require further research, although the current implementation is already usable in practice.

To protect users from being infected, sources of the malicious software have to be localized and eliminated. This requires scanning of huge amounts of suspicious URLs, most of which will in fact prove to be benign. Unfortunately, for such a large number of URLs to scan, the number of logs generated by CP is adequately huge. Currently a manual analysis by a security expert is necessary for each log, in most cases resulting in classification as false positive – in other words wasted effort.

The security experts are the bottleneck of the system – the speed of scanning can be easily increased by adding more CP instances, but if the stream of generated logs exceeds the processing capability of the experts, results become useless. Adding more experts is not a good solution – they are expensive and require training. Only automated methods of reducing the amount of false positives can widen the bottleneck. The increased reliability of classification not only reduces the amount of cases for manual analysis, it also makes the results more usable even without confirmation by an expert. In this case our solution can partially replace a security expert and discard most of the benign websites, leaving only a small set of websites supposedly serving malware. This enables the use of HoneySpider Network results in an Early Warning setting, e.g. through integration with FIRE. While false positives may still happen, their number is further reduced by comparing results with the low integration module. Anyway, their number after applying LC is small enough, that the occasional false positive can be accepted and handled manually.

The reduction of false positives rate by two orders of magnitude is a huge improvement. For example, a typical HoneySpider Network installation might generate about 4000 CP logs daily. Assuming 10 seconds of manual analysis per log (realistic if most of the logs are trivially benign), 11 hours of work are the absolute minimum necessary to process the results, enough work to keep two full time security experts occupied. LC can easily analyze generated logs in real time, leaving 50-100 logs for manual analysis, requiring less time for quick verification by one expert than a typical lunch break. Far more in-depth analysis is possible in this case and more instances of CP can be installed to increase throughput.

An alternative way to reduce the false positives rate is to prepare better black/white lists for CP. It is again a time consuming process – an expert has to prepare the lists manually. These lists have to be tested and modified with every change of the setup of the CP's virtual machine. Supporting multiple configurations of CP (e.g. different patchlevels or language versions) can be very involving and even the best black/white lists can not realistically eliminate all false positives. Our solution reduces the required effectiveness of the lists – tests show, that even with default lists, LC reduces the output to a level similar to the best lists in our possession. The results with the expert lists are of course even better – there's a tradeoff between the amount of time spent on analysis of false positive logs and the time devoted to maintenance of the black/white lists.

Numerous websites on the Internet require dedicated software for their content presentation. Web browsers and plugins tend to update automatically. There are many cases where automatic installation of software is not in fact a malicious activity (although this is debatable if no confirmation is necessary). In order to distinguish between benign updates and malware infections, it is possible to either update the black/white lists, or simply retrain the LC, perhaps implementing more features. An additional advantage of our solution is that the features can detect sequences of actions, where the order does matter – CP does not offer that capability (although it could be modified to do that, resulting in much more complicated black/white lists).

Admittedly, the set of expert-classified logs we used for training is relatively small. We are currently building a larger set, trying to eliminate duplicates and collect as many different malicious and suspicious but benign cases as we can. This should have a positive effect on the results achieved by LC. Still, the results with the set available at the time of writing are already quite satisfactory and the solution can be deployed in practice. Additional tests using sources with high percentage of truly malicious websites will be performed to verify the false negatives rate of LC, but the tests performed so far suggest that it is acceptable.

Work on the feature set is of course also never complete – as malicious behaviours change, so must the features. Nevertheless, the stability of the feature set is much better than that of the B/W lists in CP. Retraining on new examples will often give good results without changes to the feature set.

6 New and better contextual features in Argos and Shelia (VU)

In Deliverable D15, we discussed Shelia, a Windows-based honeypot for client-side applications[1]. Shelia detects exploits of applications like browsers, mail clients, photo viewers, etc. As far as client-side honeypots are concerned, Shelia is interesting, as it is raises virtually no false alarms. In other words, if Shelia claims that something is malicious, you can be sure that it really is malicious. For incident management purposes, this is a very desirable property.

On the other hand, Shelia also has some disadvantages:

1. Limited contextual information. We designed Shelia originally as a research tool, and never put much effort in really maximizing the contextual information that can be gathered. Specifically, Shelia as described in Deliverable D15 is mostly limited to:
    a) target application (which program is attacked?);
    b) system address width (is it 32 or 64 bit system?);
    c) process identifier;
    d) timestamp;
    e) attack object (e.g., the file or URL used to compromise the target);
    f) payload (the exploit - but not very precise);
    g) payload address (address of the payload in the victim's memory);
    h) malware (name, length, hashes and binary);
    i) activity context (API calls and arguments of victim under attack).
2. Shelia is Windows-specific (and needs continuous updating with new versions of the OS).
3. The shellcode analysis in Shelia, while interesting for a human analyst, is not powerful enough for automated analysis. This is an example of the Wombat feedback

[1]`http://www.cs.vu.nl/~herbertb/misc/shelia`

loop, where the actual use of the dataset showed that the information was not quite good enough. The main problem is that the recorded shellcode is quite imprecise and shellcode and unpackers are not clearly distinguished.

To remedy the above issues, we pursued two parallel tracks. First, we improved Shelia to record more contextual information (Section 6.1). Second, we designed a new client-side honeypot, based on our popular Argos honeypot technology that is much more accurate in detecting unpackers, and shellcode (Section 6.2). The new honeypot also tracks and analyses how the malicious code affects files and processes after the infection (Section 6.3. Because of its disk tracking abilities, the new honeypot version will be referred to as Argos DiskDuster (Argos-DD).

6.1 More contextual information in Shelia

We augmented Shelia to gather more contextual information using a variety of sources. The enhancement operates as a python program that improves the default contextual information that is gathered for each new alert. Specifically, Shelia is now capable of also gathering:

1. the domain, the ASN, the geographical information (where is the server located?), and other suspect IP addresses in that domain (by crosslinking with the NoAH and Harmur databases);
2. domain registrant (with email address);
3. port numbers on which the clients contacted the malicious server;
4. HARMUR classification of threat id and threat class (crosslinking with HARMUR);
5. operating system of the malicious server;
6. for the malware, Shelia stores the VirusTotal description, the detecton ratio of various AV products at the moment of attack, the meta-names, and the linked DLLs.

6.2 Argos as client-side honeypot

As a much more substantial task, we initiated an approach to deal with the two stickier problems: lack of portability and lack of accuracy in handling shellcode. We decided to adapt the Argos server-side honeypot to make it suitable as a client-side honeypot. More specifically, we adapted it in such a way that it can almost function as a drop-in replacement for Shelia (except that it does not gather the same information). As our new version also tracks infections on disk (see Section 6.3), we will refer to the client honeypot as Argos-DD.

A client honeypot based on Argos is attractive as it can potentially handle a variety of operating systems (with minimal effort to extract platform-specific information) and is very accurate (virtually no false positives) with a much wider detection range than Shelia (fewer false negatives). In our work, we stil focus on Windows, but the fact that Argos-DD itself is capable of handling a host of operating systems (including Linux, BSD, and a variety of Windows) is reassuring.

Moreover, we shall see that the Argos-DD is very precise in detecting unpackers and shellcode. In summary, we created a client-side honeypot that can automatically detect a memory corruption attack and dynamically analyze the injected payload. The analysis reveals the execution trace that includes the API calls that give a high level view of the low level payload and labels the various decompression, decryption, execution stages.

6.2.1 Argos-DD shellcode tracking

Argos-DD targets memory corruption attacks. Memory corruption bugs originate in programs written in a language that defers memory management to the programmer. An attacker is able to abuse these bugs, for instance by injecting code into a buffer and diverge the flow of control to this buffer by writing beyond the boundary of a buffer, overwriting critical values. Most of the time this results in total control of the system for the attacker.

Various partial protections against memory corruption attacks exist, but currently there is no comprehensive protection. Dynamic taint analysis is perhaps the most reliable technique to detect memory corruption, but it is currently too slow to run on production machines. For this reason, it is typically used in honeypots. Argos-DD is based on dynamic taint analysis.

Moreover, from an analysis point of view, existing protection solutions are unable to provide detailed information about the injected payload. It takes manual reverse engineering to extract the shellcode from its layers of protection to analysis its intent and effect on the system. A highly specialized and time-consuming task.

Our new version of Argos-DD is designed to overcome this problem. Like its Argos predecessor, Argos-DD is a full system emulator that is able to detect a code-injection attack—which is fairly common. In addition, it can provide detailed information about the payload—which is not.

In a nutshell, Argos-DD works as follows. First, it uses dynamic taint analysis to detect the point where an attacker changes the control flow and diverges it to the injected payload. Next, as soon as it detects that injected code is executing, it logs all information relevant to the post-attack analysis. During the execution it vets the instructions executed by the payload to make sure the payload does not propagate itself to other systems. It is only allowed to attack our (honeypot) system, which we will reset after finishing the analysis of the payload.

The collected information our solution provides includes the low-level code, all memory read and written, symbolic information about operating system calls made by the payload, and the stages of the instructions (more later). From the symbolic information we are able to extract a higher level overview of the shell-code. The stages help us separate the different parts of the payload, such as the unpacker and the shell-code. Decisions on appropriate actions can be facilitated by the collected information, but it also useful in analysis of the payload itself.

6.2.2 Argos-DD shellcode tracking architecture

Dynamic taint analysis marks incoming data from an untrusted source with a taint flag. During the lifetime of the data, we track the propagation of the data and decides whether the taint should propagate with the data.

DTA allows us to perform checks on the usage of untrusted data. In our case we are interested in the case where untrusted data is used to subvert the control flow of an application, because this is common behavior of an application that is attacked using a memory corruption vector. Argos-DD can detect this case and alerts the user if such a case occurs. When the instruction after the execution flow change is tainted, we know that we have detected a code-injection attack. After the attack detection we prolong the attack and give it the opportunity to execute the injected code[2].

An attack's payload typically consists of a NOP sled, one or more unpackers and the actula shellcode itself. For analysis, it would be extremely useful to separate these three components. Moreover, it helps if the components are presented not just as a long list of instructions executed, but rather in a more structured way, where repeating sequences are recognised as loops and calls to system function are presented symbolically. We certainly would prefer not to present the disassembled code of library functions—as this code has nothing to do with shellcode. Rather we just want to that the shellcode called functions X and Y from library Z.

[2]As mentioned in a previous deliverable, doing so was not as trivial as it sounds, and required us to modify the underlying Qemu emulator

A payload that is packed needs an unpacker to reverse the packing. Since separating the stages can aid in the analysis, during payload execution, we explicitly mark to which stage an instruction belongs. Stages are enumerated from zero upto n (where stage n represents the final shellcode). The stage of an instruction is determined by the bytes that form the instruction using the following algorithm:

1. When we start tracking, all instruction start with stage equal to zero.
2. When an instruction is written by the payload, we increase the stage by one.
3. When an instruction of the payload is executed, we store the assigned stage.

This algorithm is based on the assumption that unpackers decode the encoded instructions. The taint of an encoded instruction will propagate with the derived decoded instruction. With multiple layers of protection in a packer, the instructions will be decoded multiple times and the stage number increases accordingly. In the case of multiple stages we can extract the following information.

- Instructions with stage zero belong to the nop-sled and the packer stub.
- Instructions with a stage greater than zero but smaller than the maximum stage belong to intermediate protections layers of the packer.
- Instructions on the top stage represent the shell-code.

While we are interested in what the payload does on our controlled system (which API calls it makes, etc.), we also want to prevent the payload from attacking other connected systems. Since the payload must use the provided system interfaces to interact with the system, we control the payload by subjecting it to an API filter.

An API filter is a white list of interfaces that code is allowed to use. Any use of an interface not defined in the white list will halt the execution. For example, we allow the payload to load dynamic libraries for additional interfaces, but we prevent the spawning of a remote shell. With the white list we want to steer the execution towards the point where the payload provides control to the attacker or downloads and executes a second stage of the attack (e.g a rootkit). The whitelist is in Argos-DD is extremely flexible and can be specified on a site-by-site basis using a simple configuration file, similar to the one shown in Figure 6.1.

```
# Default whitelisted exports that are used by almost all payloads.
[kernel32.dll]
LoadLibraryA
LoadLibraryExA
LoadLibraryW
LoadLibraryExW

# Whitelist for the Metasploit bind tcp payload.
[kernel32.dll]
#CreateProcessA
#WaitForSingleObject
[ws2 32.dll]
WSAStartup
WSASocketA
bind
listen
accept
closesocket

# Whitelist for the Metasploit dowload exec payload.
[kernel32.dll]
GetProcAddress
GetSystemDirectoryA
#WinExec
[urlmon.dll]
URLDownloadToFileA
```

Figure 6.1: Example whitelist

6.2.3 Argos-DD contextual information

Argos-DD detects detects attacks that are different from the attacks detected by Shelia. During our evalution of Argos-DD with life attacks (exposing it to real malicious websites), we found several attacks that were not covered by Shelia. Moreover, the information is considerably more precise and accurate.

Concretely, Argos-DD detects the following structural information for shellcode analysis.

- The different stages, neatly separated.
- The executed instructions (with a friendly presentation).
- The memory read and written by the instructions.
- Symbolic information (e.g., the library function names).
- The state of the CPU (e.g., the value of the registers.).

As we shall see in the next section, Argos-DD also tracks the effects malicious code has on the file system and how these file system changes in turn affect other programs.

6.3 Tracking malware to disk and back

Despite a plethora of defense mechanisms, attackers still manage to compromise computer systems. Sometimes they do so by corrupting memory (by means of buffer overflows, say)—often injecting a small amount of shellcode to download and install the real malware. Sometimes the users themselves install trojanized software. Worse, the malware may be active for days or weeks, before it is discovered. Longer, if the malware hides itself well.

Upon discovery of a compromised machine, one of the most challenging questions is: what did the malware do? Maybe it was active for weeks. Which files has it modified? Did the attackers change my financial records? Did they create new accounts? Can I still trust any of the files created after the compromise, or should I check each and every one, manually? Did the initial attack spread to other programs or other machines?

Without such information, it is very difficult to recover from an attack. Currently, the only sane state a system can revert to is the last known good backup. This leaves the question of what to do with all changes to the system that occurred since then. Ignoring them completely is safe, but often unacceptable—losing valuable data generally is. Accepting them blindly is easy, but not safe—modifications may be the result of the malware's actions. However, the alternative of sifting through each of the files (or even blocks) on disk one by one to see whether it can still be trusted may be much too time-consuming. Given precise information about what the malware changed on disk, which other processes it infected (and what *they* changed on disk, etc.), will greatly reduce the recovery time.

Besides tracing attack effects in memory (e.g., for shellcode analysis), Argos-DD also traces the effects of malware on the file systems of a compromised system, separating the good data from the bad. In summary, Argos-DD uses dynamic taint analysis (DTA) to monitor all the malware's actions. It taints all disk writes by the malware as malicious. Taint propagates whenever the malicious bytes are read, copied, or used in operations. If such bytes compromise other processes, Argos-DD traces those also.

Tracking an infection requires tracking the actions and data produced by the malicious code. Specifically, we should keep a tab on where its data ends up and what actions and data depend on such data. Where almost all existing intrusion recovery solutions construct dependency graphs explicitly, Argos-DD tracks dependencies directly, by means of dynamic information flow tracking (taint analysis) and at byte-level granularity.

6.3.1 Assumptions and limitations

By tracing what data was directly or indirectly generated by the attack, and what data was not, we reduce recovery time significantly. For our system, we make the following assumptions:

1. Attacks can infect both user processes and the kernel.

2. Argos detects is capable of detecting the attack and tracing it back to the moment of infection.

3. Like most DTA implementations, Argos-DD does not track implicit flows. Implicit flows often lead to overtainting. If we cannot trust the taint value, administrators still have to double check everything. However, skipping them entirely may lead to false negatives. Argos-DD partially handles the problem by taking a conservative approach for malicious data on disk; whenever a process has read malicious bytes, all subsequent writes are marked 'possibly inconsistent'[3] and should be handled explicitly (either by recovery scripts or by human intervention).

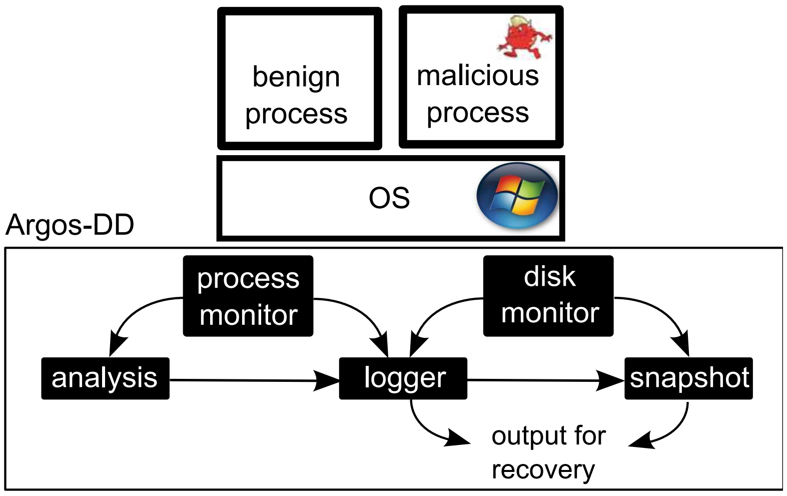


Figure 6.2: Argos-DD architecture

6.3.2 Architecture

Figure 6.2 illustrates the components of Argos-DD. Our work extends the existing Argos DTA engine with the modules colored in black. All these components operate at the emulated hardware level and work with unmodified operating systems. The monitoring modules trace both process execution and disk input/output. Specifically, the *disk* monitor keeps track of all reads and writes to disk, while *process* monitor tracks running processes–as described in Section 6.2. When Argos-DD detects an attack, it marks the compromised process as 'malicious'. For all malicious processes, the process monitor activates the analysis module.

The *logger* stores all information generated by the two monitor modules. The logs always include details about which process wrote which bytes on disk. In the case of a compromised process, they also contain detailed information about the context of the process. The *snapshot* module takes a snapshot of the disk drive based on user-specified policies.

[3]So far, we saw no reason to also mark I/O unrelated to the FS as possibly inconsistent, although it would be easy to do so.

Process monitoring and analysis

The process monitor traces malicious processes. Since it resides at the emulated harware level, normal process semantics as defined by the operating system do not apply, and we need an OS-agnostic way to identify processes. Fortunately, we can use the x86 control registers for this purpose. In particular, Argos-DD uses the CR3 register, a 32 bit, page aligned number that points to page directory. Since it is page aligned, the least significant 12 bits are not used and we use them to store additional information about a process. For instance, we use one bit to distinguish malicious processes from benign ones.

Argos-DD considers a process malicious if one of the available detection mechanisms detected malicious behavior and marked the process as such. Malicious processes in turn may infect other processes and it is important that we track all of them. The detection mechanism subsequently selects an appropriate analysis module to analyze the malicious process. Currently we support the analysis of compromised processes (good processes gone bad, e.g., due to remote exploits) and pure malware (e.g., trojans).

Compromised process Argos-DD considers a process compromised when an attack diverts the process' control flow. A typical control flow diversion occurs because of, say, a buffer overflow due to data arriving from the network. However, Argos-DD also considers a process compromised if it is attacked by local processes—for instance, when it uses a DLL created by a malicuous process, or suffers from a control flow diversion attack caused by another malicious process. The process monitor collects information about such processes for the attack analysis phase. For instance, information to identify the compromised process, but also the execution context, such as the loaded dynamic libraries.

As mentioned earlier, the process monitor can enforce a containment policy to prevent a compromised process from subverting the system, but still provide enough information to extract the payload's behavior. The policy can, for example, prevent the execution of malware downloaded by a download-exec shell-code; the collected information includes unpacking stages, API function resolving, and calls to API functions.

Malware process We consider a process pure malware, if a detection technique such an AV scanner, marks it malicious by a detection, or if it is created by malicious code. In the case of malware, the process monitor informs the disk monitor of a malicious process capable of altering persistent memory.

Tracking data on persistent memory

Argos-DD logs all operations which modify data on persistent storage. The information must provide precise information about the modified data (such as location, length, etc.) as well as a way to distinguish between the operations of different processes. Since write operations execute in the context of a process and since we already identify process by means of the CR3 control register, we reuse the process identification for disk write attribution.

Unlike many other DTA implementations, Argos-DD tracks taint not just for memory, but also for disk writes. The disk monitor uses a shadow memory for the drive to hold taint values, where taint values include information about the process that modified the data on disk. When input/output operations are made, the map is updated as necessary and the taint information is propagated to other shadow memories—for instance, when a program reads tainted bytes from disk into main memory.

Whenever a process write bytes to disk, the disk monitor checks whether the operation is made by a process from the list. If so, it marks the written bytes in the map as malicious. If not, the disk monitor checks the main memory taint map to see if the bytes to be written come from an untrusted source, in which case they are marked as suspicious in the map, otherwise they are clean. When the program reads disk data into memory, the disk monitor propagates the taint from the disk map into the main memory map.

The disk monitor module records every write to disk and classifies data in four types: (a) clean, which is known to be generated by benign applications, (b) malicious which is generated by malicious applications, (c) suspicious which enters the system via an untrusted source such as the network card, and (d) possibly inconsistent–as discussed next.

Possible inconsistencies

Once a process has accessed malicious bytes, we should be careful with subsequent writes, even if the written data is clean. The reason is that sometimes later writes only make sense if the earlier ones also happened. The problem is one of implicit flows and existing approaches typically ignore it, or simplistically terminate a process as soon as it reads malicious data.

There is no generic solution to this problem. However, by marking all clean writes of a process as 'possibly inconsistent' after it accessed malicious data, Argos-DD greatly reduces the cost of sifting through the writes to determine which ones should be retained. The only bytes to consider are those explicitly labelled.

Moreover, possible inconsistencies are often easy to resolve automatically using domain-specific knowledge. The Argos-DD architecture allows for recovery scripts to handle these cases. For instance, the disk defragmenter does read and write malicious bytes, but we can safely mark all writes that do not contain malicious bytes as benign. The same is true for disk encryption software, or AV scanners.

Similarly, some files may be shared by benign and malicious applications alike, without putting the benign applications at risk. Consider the equivalent of a UNIX password file. An attacker may add an entry when creating a new user account. Benign processes may read the modified file in its entirety in memory (e.g., to look up legitimate users), but unless these processes acually use the malicious data (which we track), we can consider all subsequent writes by these readers as benign.

Besides DLLs, Windows processes frequently access shared registry keys. Many trojans, worms, and backdoors make sure they will run after a reboot by introducing autorun keys and values in the Windows registry—typically by adding keys under:
`HKLM\Software\Microsoft\Windows\CurrentVersion\Run`, or
`HKLM\System\CurrentControlSet\Services\`< name>.

These keys are not likely to affect benign processes, including the parent process that reads the keys and starts up the appropriate programs at boot time (`explorer.exe`).

6.4 Summary

In this chapter, we explained how–as a response to issues that emerged out of the Wombat feedback loop–we improved the contextual information produced by the high-interaction honeypots Shelia and Argos. As a stop-gap measure, we improved Shelia to generate more contextual information, while as more fundamental approach, we completely re-designed Argos to :

1. operate as a client-side honeypot;
2. constrain malicious code executing on the analysis engine;
3. produce more and more reliable information
   - analysis of shellcode (separating nop sled, unpackers, and real shellcode)
   - track malicious processes across over disk operations.

The resulting system provides a wealth of information to security analysts, as well as administrators that need to recover their systems after an attack.

Bibliography

[1] Anubis: Analyzing unknown binaries. http://anubis.iseclab.org/.

[2] AQABA Search Engine Demographics. http://http://www.aqaba-sem.com/search_ed.htm/.

[3] AutoIt. http://www.autoitscript.com/autoit3/index.shtml/.

[4] BuddyFetch. http://buddyfetch.com/.

[5] Crowbar. http://simile.mit.edu/wiki/Crowbar.

[6] Europe surpasses north america in instant messenger users, comscore study reveals. http://www.comscore.com/press/release.asp?press=800.

[7] Google safe browsing api. http://code.google.com/apis/safebrowsing/.

[8] H1N1 Shortcut Malware. http://www.f-secure.com/weblog/archives/00001738.html.

[9] MessengerFinder, Find people online. http://messengerfinder.com/.

[10] MSN Polygamy. http://www.softpedia.com/get/Internet/Chat/Instant-Messaging/MSN-Messenger-7-8-Polygamy.shtml/.

[11] Norton safe web from symantec. http://safeweb.norton.com/.

[12] nslookup. http://en.wikipedia.org/wiki/Nslookup.

[13] Planetlab, an open platform for developing, deploying and accessing planetary-scale services. http://www.planet-lab.org.

[14] Scraping facebook email addresses. http://kudanai.blogspot.com/2008/10/scraping-facebook-email-addresses.html.

[15] Skype Fast Facts, Q4 2008.
http://ebayinkblog.com/wp-content/uploads/2009/01/skype-fast-facts-q4-08.pdf.

[16] Spam Archive. http://untroubled.org/spam/.

[17] The state of spam a monthly report august 2007. http://www.symantec.com/avcenter/reference/Symantec_Spam_Report_-_August_2007.pdf.

[18] StopBadware Blog : China Hosts Majority of Badware Sites. http://blog.stopbadware.org/2008/06/24/china-hosts-majority-of-badware-sites.

[19] Surbl. http://www.surbl.org.

[20] Symantec. http://www.symantec.com/index.jsp.

[21] Urlblacklist.com. http://www.urlblacklist.com/.

[22] Virustotal, online virus and malware scan. http://www.virustotal.com/.

[23] Vulnerability in PNG Processing Could Allow Remote Code Execution. http://www.microsoft.com/technet/security/bulletin/MS05-009.mspx.

[24] W32.Bropia. http://www.symantec.com/security_response/writeup.jsp?docid=2005-012013-2855-99&tabid=2.

[25] S. Antonatos, I. Polakis, T. Petsas, and E. P. Markatos. A Systematic Characterization of IM Threats Using Honeypots. In *Network and Distributed System Security Symposium (NDSS)*, 2010.

[26] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling. The Nepenthes Platform: An Efficient Approach to Collect Malware. In *9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2006.

[27] 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.

[28] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient detection of split personalities in malware. In *Network and Distributed System Security (NDSS)*, 2010.

[29] U. Bayer. Ttanalyze a tool for analyzing malware. Master's thesis, Vienna University of Technology, 2005.

[30] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Annual Network& Distributed System Security Symposium*, 2009.

[31] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. Insights into current malware behavior. In *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[32] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. Insights into current malware behavior. In *LEET'09: 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats, April 21, 2009, Boston, MA, USA*, Apr 2009.

[33] 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.

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

[35] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[36] D. Beck, B. Vo, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 368–377, 2005.

[37] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41. USENIX Association, 2005.

[38] J. Canto, M. Dacier, E. Kirda, and C. Leita. Large scale malware collection: Lessons learned. In *IEEE SRDS Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems*, 2008.

[39] E. Carrera. Pefile, `http://code.google.com/p/pefile/`.

[40] J. R. Crandall, Z. Su, and S. F. Wu. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *12th ACM conference on Computer and Communications Security*, pages 235–248. ACM Press New York, NY, USA, 2005.

[41] F-Secure. Malware information pages: Allaple.a, `http://www.f-secure.com/v-descs/allaplea.shtml`, December 2006.

[42] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: Vmm detection myths and realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, 2007.

[43] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed Systems Security Symposium (NDSS)*, 2003.

[44] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel.* Addison-Wesley Professional, 2005.

[45] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[46] G. Jacob, H. Debar, and E. Filiol. Malware behavioral detection by attribute-automata using abstraction from platform and language. In *Recent Advances in Intrusion Detection (RAID)*, 2009.

[47] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[48] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, 2006.

[49] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 91–100. ACM, 2007.

[50] K. Julisch. Clustering intrusion detection alarms to support root cause analysis. *ACM Transactions on Information and System Security (TISSEC)*, 6(4):443–471, 2003.

[51] M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *ACM Workshop on Recurring malcode (WORM)*, 2007.

[52] A. Lanzi, M. Sharif, and W. Lee. K-tracer: A system for extracting kernel malware behavior. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, 2009.

[53] C. Leita. *Automated protocol learning for the observation of malicious threats.* PhD thesis, Université de Nice-Sophia Antipolis, December 2008.

[54] C. Leita, U. Bayer, and E. Kirda. Exploiting diverse observation perspectives to get insights on the malware landscape. In *DSN 2010, 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2010.

[55] C. Leita and M. Dacier. SGNET: a worldwide deployable framework to support the analysis of malware threat models. In *7th European Dependable Computing Conference (EDCC 2008)*, May 2008.

[56] C. Leita and M. Dacier. SGNET: Implementation Insights. In *IEEE/IFIP Network Operations and Management Symposium*, April 2008.

[57] C. Leita, M. Dacier, and F. Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sep 2006.

[58] C. Leita, K. Mermoud, and M. Dacier. Scriptgen: an automated script generation tool for honeyd. In *21st Annual Computer Security Applications Conference*, December 2005.

[59] J. Leskovec and E. Horvitz. Planetary-Scale Views on a Large Instant-Messaging Network. In *Proceedings of WWW 2008*, April 2008.

[60] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *10th ACM conference on Computer and communications security*, pages 290–299. ACM New York, NY, USA, 2003.

[61] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[62] A. Moser, C. Kruegel, and E. Kirda. Limits of Static Analysis for Malware Detection . In *23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[63] M. Neugschwandtner, C. Platzer, P. M. Comparetti, and U. Bayer. danubis: Dynamic device driver analysis based on virtual machine introspection. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, DIMVA'10, pages 41–60. Springer-Verlag, 2010.

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

[65] NirSoft. Special folders viewer website. `http://nirsoft.net/utils/special_folders_view.html`.

[66] P. Orwick and G. Smith. *Developing Drivers with the Microsoft Windows Driver Foundation*. Microsoft Press, 2007.

[67] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[68] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proceedings of ACM SIGOPS Eurosys 2006*, April 2006.

[69] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *ACM Sigops EuroSys*, 2006.

[70] H. Project. Know your enemy: Learning about Security Threats. *Pearson Education, Inc.*, 2004.

[71] T. H. Project. Capture-hpc project website. `https://projects.honeynet.org/capture-hpc`.

[72] N. A. Quynh and Y. Takefuji. Towards a tamper-resistant kernel rootkit detector. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 276–283. ACM, 2007.

[73] K. Rieck, T. Holz, C. Willems, P. Duessel, and P. Laskov. Learning and classification of malware behavior. In *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[74] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[75] R. Riley, X. Jiang, and D. Xu. Multi-aspect profiling of kernel rootkit behavior. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 47–60, New York, NY, USA, 2009. ACM.

[76] R. Rolles. Unpacking virtualization obfuscators. In *3rd USENIX Workshop on Offensive Technologies. (WOOT)*, 2009.

[77] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *22nd Annual Computer Security Applications Conf. (ACSAC)*, 2006.

[78] M. Russinovich. *Filemon*, 2010. `http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx`.

[79] P. Saxena, R. Sekar, M. R. Iyer, and V. Puranik. A practical technique for containment of untrusted plug-ins. Technical Report SECLAB08-01, Stony Brook University, 2008.

[80] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Network and Distributed System Security (NDSS)*, 2008.

[81] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *ACM conference on Computer and communications security (CCS)*, 2009.

[82] D. Smith. Allaple worm (ISC diary), `http://isc.sans.org/diary.html?storyid=2451`.

[83] VirusTotal. `www.virustotal.com`, 2007.

[84] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering persistent kernel rootkits through systematic hook discovery. In *Recent Advances in Intrusion Detection (RAID)*, 2008.

[85] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 2(2007), 5.

[86] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufman, 2005.

[87] M. Xie, Z. Wu, and H. Wang. HoneyIM: Fast detection and suppression of instant messaging malware in enterprise-like networks. In *Proceedings of the 2007 Annual Computer Security Applications Conference (ACSAC07)*.

[88] C. Xuan, J. Copeland, and R. Beyah. Toward revealing kernel malware behavior in virtual execution environments. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, 2009.

[89] T. Yetiser. Polymorphic viruses - implementation, detection, and protection , `http://vx.netlux.org/lib/ayt01.html`.

[90] H. Yin, Z. Liang, and D. Song. Hookfinder: Identifying and understanding malware hooking behaviors. In *Network and Distributed Systems Security Symposium (NDSS)*, 2008.