Malware Mitigation and Remediation Strategies

Tesi di
Emanuele Passerini

Relatore
Prof. Danilo Bruschi
Coordinatore del Dottorato
Prof. Ernesto Damiani

Anno Accademico 2008/2009
In today’s Internet connected environment, there is no doubt that the web is the main vector of attacks for cybercriminals, and the huge amount of everyday new threats suggests this situation isn’t going to change anytime soon.

Many of the most malicious attacks start as an apparently innocuous spam message with few words of text with bad spelling and a single URL. These messages often fool classical spam engines that probably simply look for keywords. All it takes is one distracted or curious click of the mouse and the site is visited: web attack is launched or the malicious payload is downloaded, probably providing remote access to the user’s computer, and maybe to the internal network. Criminals through Internet control networks of such compromised computers, called botnets, and use them to commit a multitude of frauds and thefts. Furthermore, botnets provide bandwidth which is used to launch distributed denial-of-service attacks (DDoS) and to send unsolicited email (spam), as well as IP address diversity to make IP address-based blocking strategy ineffective. Spam emails sent during those attacks are themselves vectors of attacks, used to deceive people into entering login credentials (phishing), installing malicious software (malware), or sharing bank account information (scams). And so the cycle repeats: email related threats create new botnet hordes, which are able to launch a flood of spam to drown our mailboxes.

Recently botnets, through the adoption of particular techniques, become more resistant to discovery and counter-measures, and one of these new technique is named fast-flux. This dissertation presents FluXOR, the system we developed to detect and monitor fast-flux service networks. FluXOR monitoring and detection strategies entirely rely on the analysis of a set of features observable from the point of view of a victim of the scams perpetrated through the botnets. The
huge amount of information gathered through FluXOR can be used to mitigate botnet threats, actively defending benign hosts against underground attacks.

The spreading of the botnet phenomenon reveals a very critical point of the Internet Infrastructure, namely the web applications, which are the most used attack vector used so far for the malware spreading.

Attackers have also been increasingly using the web both client and server side attacks in order to steal information from targets and redirect victims to compromised sites containing malicious codes. In this dissertation we also presents a hybrid analysis framework for the detection of vulnerabilities in web applications that blends together the strengths of static and dynamic approaches. Our technique, that is completely transparent to the applications, represents a promising approach to mitigate injection attacks, also on legacy systems, mainly because the improvement with respect to a taint analysis entirely dynamic is significant.

In response to all these threats, security industries, academic researchers and organizations have developed a myriad of strategies and tools to defend their systems and networks. However, the attackers seem to be always one step ahead, they are still succeeding, and the defensive tools are failing. In fact, to defend against malicious programs, users typically rely on malware detectors, which try to detect and prevent threats before the system is damaged. Unfortunately, in many cases detection and prevention are not possible, for example for the lack of the appropriate detection signature. In such a situation, post-infection remediation remains the only solution to get rid of a malware and of the damages it may have caused to the system, other than reinstalling the entire system. However, it has been shown that sometimes automatic remediation procedures could cause more problems than they would solve, together with a lack of a testing methodology to assess the effectiveness of remediation procedures commonly used by commercial malware detectors. As a final contribute, in this dissertation we describe the architecture, the implementation and the evaluation of a framework for remediation procedure automated testing, showing that none of the tested commercial solutions turned out to be complete. Motivated by obtained results, we developed our own system for automatically generating malware remediation procedures, able to remove from almost any infected system all the resources (e.g., files, processes and registry keys) created during the infection, but also to recover the original state of the resources that have been altered by the malware sample used to generate the procedure. The results of the evaluation witness the effectiveness of our contribution. Moreover our remediation procedures outperform in completeness and soundness those currently available in top-rated commercial anti-malware software.
Acknowledgement

The work presented in this dissertation could not have been possible without the help, the presence and support of several people and friends.
I want to thank my adviser, Prof. Danilo Bruschi for having always allowed me to freely choose interest topics and develop ideas the way that I liked the most. During these years I have been able to work with good friends on topics that I really enjoyed.
I am grateful to the three external referees of my dissertation: Christopher Kruegel, Engin Kirda and Jonathon Giffin. Thank you so much for your valuable feedback.
I have been fortunate to collaborate with two good friends as Roberto Paleari and Lorenzo Martignoni: we have spent a wonderful time together working hard and trying to follow crazy ideas in short time (don’t worry in the next week we will finish...). Your help has been very important.
Particular thanks go to my good friends and colleagues Giampaolo Fresi Roglia (Gianz), Andrea Lanzi and Lorenzo Cavallaro.
Thanks to Mattia Monga for the encouragement and long talks in the lab. During these years I have been very lucky to be part of the hacking team named ”The Chocolate Makers” (I Cioccolatai). With these crazy friends we participate to the various worldwide hacking contexts often ending up in the higher positions and sometimes we also won! Many thanks to all ”old” members of the team: Aristide Fattori (Oreste), Luca Giancane (Jack), Davide Marrone (Skebby Brown), and Antonio Nappa (Jeppo). Thank you also for making the LaSeR lab a better and funnier place (thanks for all the coffee we took).
To conclude I wish to thank my family for all the support I received during these long years in Milan. I am grateful to my parents Ornella and Gino, who have
always patiently believed in me, encouraging and supporting me in any choose
I made. Many thanks goes to Valentina, my heart: she always encourages and
support me, in the good and in the hard times. But now it’s your turn!
I am sure quite sure that I forgot to mention somebody, please forgive me.
to Vale, Ornella and Gino
1 Introduction ix
  1.1 Motivations and contribution of the dissertation xi
    1.1.1 Mitigation strategies xi
    1.1.2 Remediation strategies xii
  1.2 Summary of the contributions xii
  1.3 Organization of the dissertation xiii

2 Terminology and malware basics 1
  2.1 Botnets 1
    2.1.1 Creation 1
    2.1.2 Infection strategies 2
  2.2 Web applications attacks 3
    2.2.1 SQL injection 4
    2.2.2 Cross-Site-Scripting 4

3 Related work 6
  3.1 Fast-flux networks detection 6
  3.2 Mitigation of Web application attacks 8
  3.3 Malware effects: analysis and remediation strategies 9

4 FluXOR 12
  4.1 Introduction 12
  4.2 Problem description and solution overview 15
  4.3 FluXOR: an outline 18
  4.4 Characterizing fast-flux service networks 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.1</td>
<td>Features characterizing the domain name</td>
<td>19</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Features characterizing the degree of availability of the network</td>
<td>20</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Features characterizing the heterogeneity of the agents</td>
<td>21</td>
</tr>
<tr>
<td>4.5</td>
<td>Combining the features for detection</td>
<td>24</td>
</tr>
<tr>
<td>4.6</td>
<td>Clusterization of fast-flux service networks</td>
<td>25</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Clusterization</td>
<td>26</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Clusterization algorithm</td>
<td>27</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Periodical re-clusterization</td>
<td>29</td>
</tr>
<tr>
<td>4.7</td>
<td>Architecture and implementation of the system</td>
<td>29</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Collector</td>
<td>30</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Monitor</td>
<td>30</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Detector</td>
<td>31</td>
</tr>
<tr>
<td>4.7.4</td>
<td>Clusterizer</td>
<td>32</td>
</tr>
<tr>
<td>4.8</td>
<td>Experimental results</td>
<td>32</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Detection accuracy</td>
<td>33</td>
</tr>
<tr>
<td>4.8.2</td>
<td>Empirical analysis of the fast-flux service networks phenomenon</td>
<td>35</td>
</tr>
<tr>
<td>4.8.3</td>
<td>Correlations among different fast-flux service networks</td>
<td>39</td>
</tr>
<tr>
<td>4.8.4</td>
<td>Correlation between fast-flux botnets and spam</td>
<td>43</td>
</tr>
<tr>
<td>4.8.5</td>
<td>Fast-flux domains lifetime</td>
<td>45</td>
</tr>
<tr>
<td>4.9</td>
<td>Mitigation</td>
<td>45</td>
</tr>
<tr>
<td>4.10</td>
<td>Discussion</td>
<td>46</td>
</tr>
<tr>
<td>5</td>
<td>Web attacks mitigation with Phan</td>
<td>48</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>Hybrid analysis of web applications</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>Phan: a hybrid analyzer for PHP applications</td>
<td>54</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Off-line analysis</td>
<td>55</td>
</tr>
<tr>
<td>5.3.2</td>
<td>On-line analysis</td>
<td>59</td>
</tr>
<tr>
<td>5.4</td>
<td>Implementation details</td>
<td>60</td>
</tr>
<tr>
<td>5.5</td>
<td>Experimental results</td>
<td>60</td>
</tr>
<tr>
<td>5.6</td>
<td>Discussion</td>
<td>62</td>
</tr>
<tr>
<td>6</td>
<td>Testing Remediation Procedures</td>
<td>63</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>63</td>
</tr>
<tr>
<td>6.2</td>
<td>The importance of remediation</td>
<td>64</td>
</tr>
<tr>
<td>6.3</td>
<td>Testing methodology</td>
<td>66</td>
</tr>
<tr>
<td>6.3.1</td>
<td>The ideal remediation procedure</td>
<td>66</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Testing the completeness of a remediation procedure</td>
<td>68</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.4</td>
<td>Implementation</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>6.4.1 Tracing the malware sample</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>6.4.2 Analysis of the system call trace</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>6.4.3 Filtering of intangible high-level</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>transitions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.4.4 Execution and evaluation of the remediation</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>procedure</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>Experimental results</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>6.5.1 Experimental setup</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>6.5.2 Evaluation of state-of-the-art malware</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>detectors</td>
<td></td>
</tr>
<tr>
<td>6.6</td>
<td>Discussion</td>
<td>78</td>
</tr>
<tr>
<td>7</td>
<td>Generating remediation procedures</td>
<td>80</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>80</td>
</tr>
<tr>
<td>7.2</td>
<td>Motivations and goals</td>
<td>82</td>
</tr>
<tr>
<td>7.3</td>
<td>Overview of the system</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>7.3.1 High-level behaviors extraction</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>7.3.2 Behaviors generalization</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>7.3.3 Remediation procedures generation</td>
<td>87</td>
</tr>
<tr>
<td>7.4</td>
<td>Architecture details</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>7.4.1 High-level behaviors extraction</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>7.4.2 Behaviors clustering</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>7.4.3 Behaviors generalization</td>
<td>94</td>
</tr>
<tr>
<td>7.5</td>
<td>Evaluation</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>7.5.1 Experimental setup</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>7.5.2 Completeness of the generated remediation</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>procedures</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.5.3 Soundness of the generated remediation</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>procedures</td>
<td></td>
</tr>
<tr>
<td>7.6</td>
<td>Discussion</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion and Future Work</td>
<td>101</td>
</tr>
</tbody>
</table>
As operating systems and software applications become larger and more complex, they are more likely to contain new implementation errors, also known as bugs, which may allow attackers to try to exploit vulnerabilities, gain illegitimate access to systems and steal private data. Everyday critical vulnerabilities are reported on a wide variety of operating systems and applications, and malicious activities perpetrated through Internet are quickly becoming the number one security problem, which ranges between large scale social engineering attacks and exploiting critical vulnerabilities. Recent sophisticated attacks use polymorphism and even metamorphism mixed with cryptographically strong algorithms and self-updating functionality which make analysis and defense increasingly difficult. Nowadays a fast and reliable mechanism to mitigate, discern and generate vaccines for such attacks is vital for the successful protection of networks and systems. Also the nature of malicious code shifted recently from trying to disrupt services or cause damage to actively seeking financial gain, as a matter of fact, today malware are designed to steal sensitive information such as credit card numbers, social security numbers, accounts, pin codes, and passwords and send the information to the miscreants for evil purposes including identity theft. The underground economy, a growing market where stolen information is actively sold online, is incessantly growing. Symantec in a recent report observed the increasing of professionalization in the underground economy: miscreants are coordinated, specialized and, in some cases, competitive for the production and distribution of customized malicious code and phishing kits.

In Figure 1.1 are reported malicious code threats detected in 2008 (1,656,227) by Symantec representing over 60 percent of the approximately total (2.6 million) malicious code threats detected in total, over time. The increase of complexity and sophistication of such attacks, the professionalization of attackers and evolution of attack patterns represent the major changes in the current threat land-
scape, and justify the need for cooperation among academics, security companies and governments to fight these threats. As usual, the first recommendation is always to patch vulnerabilities found in operating systems and applications, install a good anti-virus solution (and keeping it up to date!) and use a firewall. But, as we know, apply all these solutions is not always enough; malware attacks can be mounted via different vectors or attack methods on a specific weak point, and these methods through which malware can compromise a system are sometimes referred to as threat vectors, and represent the areas that require the most attention when designing an effective solution to help reduce malware risks. Unfortunately, attackers have become skilled in circumventing such as traditional defenses. For example, even encrypted web transactions may not protect sensitive information if the victim’s computer has been previously infected.

To this end, when developing strategies to help reduce malware effects, it’s fundamental to define required operational key points where malware detection and/or prevention can be implemented. Today threats complexity could not be fought using a single solution or technology as a single line of defense but methods including a layered approach and using proactive, reactive and remediating mechanisms should be preferred. Antivirus software plays, still now, a key role in protecting systems from malicious codes; however, it should not be the only instrument used to determine malware attacks, because, as recently seen, it can fail. Finally, if a system has been compromised, there must be a common known

*Figure 1.1: Evolution of new malicious code threats as reported by Symantec.*
way to handle such a situation, being able to minimize the damage and try to remediate it as soon as possible and in the best way. As computer attacks and malware evolve, as quickly as possible new responses and solutions are essential.

1.1 Motivations and contribution of the dissertation

The dissertation contributes to the field of malware defense in two directions: to propose two mitigation strategies that try to fight the diffusion of malicious activities in Internet, and an architecture for automatically building and testing remediation strategies that operates in the malware post infection phase.

1.1.1 Mitigation strategies

The first part of the dissertation presents two proposals for mitigating the propagation of attacks and malicious code in the Internet.

Motivated by the conviction that prompt detection and monitoring of malicious activity can be a successful strategy, mainly in the case of botnet (group of malicious hosts), we focused our attention on a development of frameworks and strategies for recent attacks mitigation that can be easily used in real systems, assuring also a high level of automatism and flexibility. For the involved criminals, it can be a costly exercise to continuously develop new families of malware in order to maintain their underground activities at sufficient levels. The action of constantly register new domains results much more economical for them, and by spreading malicious code across as many as possible different websites and domains, the longevity of each malware example is enhanced. These techniques combined with the use of “fast-flux” strategy let criminals be sure that malicious websites are not taken down quickly in case of charge. High level automation process employed by criminals require minimal or no monitoring activity, assuring constant work in compromising as many websites as possible and registering new domains.

To fight some of these processes we developed FluXOR, a system to detect and monitor fast-flux service networks. FluXOR monitoring and detection strategies entirely rely on the analysis of a set of features observable from the point of view of a victim of the scams perpetrated thorough these botnets. FluXOR results can be used in real-time to deny DNS services or reject connection to or from malicious domains or hosts. During our tests we noticed ourselves that a variety of web-based attacks are perpetrated using fast-flux techniques, including fraud and impersonation as well as malware infections.
As a second proposal, we introduce a technique for protect web applications, one of the most used vector to spread malware and others attacks. We designed and implemented a hybrid analysis framework named Phan, that blends together the strengths of existing static and dynamic approaches. The preliminary static analysis phase helps reducing the typical run-time overhead connected with dynamic monitoring. Laboratory results indicate that the improvement on run-time overhead with respect to the classic entirely dynamic taint analysis on web application is significant. We think that developed framework represents a promising approach to automatically mitigate web applications injection attacks such as XSS and SQL injections, also on legacy systems, and motivate further work in this direction.

1.1.2 Remediation strategies

The last part of the dissertation addresses some open issues in the generation and testing of the remediation procedures used by anti-virus and malware-detector to clean up malware compromised systems. We define remediation procedures those procedures that can be executed on infected systems to revert the effects of an infection. First we noticed the lack of a testing methodology to assess the effectiveness of remediation procedures commonly used by commercial malware detectors and to this end we describe the architecture, the implementation and the evaluation of our automated testing framework, showing that none of the tested commercial solutions turned out to be complete (able to revert all the effects of the execution of the malware). Motivated by these results, we developed our own system for automatically generating malware remediation procedures, able to remove from any infected system all the resources (e.g., files, processes and registry keys) created during the infection, but also to recover the original state of the resources that have been altered by the malware sample used to generate the procedures. The results of the evaluation witness the effectiveness of our contribution. Moreover our remediation procedures outperform in completeness and soundness those currently available in top-rated commercial anti-malware software.

1.2 Summary of the contributions

To summarize the dissertation makes the following contributions:

(DIMVA 2008) FluXOR a system to detect and monitor fast-flux service networks. This new kind of networks of compromised hosts are used by miscreants,
as proxies, for the most illegal activities (e.g., sending spam emails, denial-of-service attacks, phishing and other web scams) with the advantage of having the identity of the core components protected and a high level of network availability. FluXOR monitoring and detection strategies entirely rely on the analysis of a set of features observable from the point of view of a victim of the scams perpetrated by these kind of botnets.

(SESSION 2009) A hybrid analysis framework that blends together the strengths of static and dynamic approaches for the detection of vulnerabilities in web applications: a static analysis, performed just once, is used to reduce the run-time overhead of the dynamic monitoring phase. Designed and implemented tool, called Phan, is able to statically analyze PHP bytecode searching for dangerous code statements; then, only these statements are monitored during the dynamic analysis phase.

(DIMVA 2009) A testing methodology to assess the quality (completeness) of the remediation procedures used by malware detectors to revert the effect of an infection from a compromised system. Developed prototype has been used to test six of the top-rated commercial malware detectors currently available on the market. The results of evaluation witness, that in many situations, the tested malware detectors fail to completely remove the effects of an infection.

(Recently submitted, currently under review) A system to generate automatically remediation procedures, that is, procedures that can be used to remediate all and only the effects of the execution of a particular malicious program in any possibly infected system. We have implemented a prototype of the system and used it to generate remediation procedures for more than 200 malware. The evaluation of these procedures demonstrates that they outperform in completeness and soundness those currently available in top-rated commercial malware detectors.

1.3 Organization of the dissertation

The dissertation is subdivided into eight chapters. Chapter 2 present a brief overview of the terminology and malware basics. Chapter 3 compares the work presented in the dissertation with the related work done by others. Chapter 4 describes FluXOR, our system to detect and monitor fast-flux service networks. Chapter 5 describes Phan, our hybrid analysis framework for detecting web application vulnerabilities. Chapter 6 presents a testing methodology to assess the
quality of the remediation procedures used by malware detectors to revert the
effect of an infection from a compromised system. Chapter 7 try to settle prob-
lems presented in the previous chapter proposing a framework for the automated
generation of remediation procedure. The dissertation ends by giving concluding
remarks in Chapter 8, discussing results and possible improvement to the ideas
presented in previous chapters and describing some opportunities for future work
that leverage the contributions of the dissertation.
Terminology and malware basics

In this chapter, we present a brief overview of bots and botnets to provide a first glance at the malicious networks we discuss later in this dissertation. We also introduce some of the most common web application attacks used by attackers and bots themselves to infect vulnerable hosts.

2.1 Botnets

Bot is a term derived from the word robot and refers to a software application that runs automated tasks in an autonomous manner. A computer system that can be remotely controlled by an attacker is commonly called a bot or zombie. A botnet, also known as zombie network, is a network of hosts infected with a particular malware sample, a malicious program that allows criminals to remotely manage the infected systems without the users’ knowledge. In the last ten years the concept of botnet have evolved: starting from small group of a dozen hosts controlled from a single C&C (command and control) center into more sophisticated and distributed system involving thousands or even millions of hosts with decentralized or P2P (Peer to Peer) control. Nowadays, botnets have become a big source of income for entire groups of cybercriminals: the risks of managing botnets are minimal and the economics are still favorable to the botmasters, the controlling entities, so we expect the problem will continue to grow.

2.1.1 Creation

To create a new zombie network there are some simple steps to perform. First a host must be infected with a special malicious program. Obfuscation and lots of encryption techniques as packing or code emulation, are typically used by
these programs in order to protect them from detection by malware detector and antivirus tools. Criminals, with no programming skills, who wants to start a new business can likewise control a botnet: they can rent an existing botnet. There are plenty of “bot for sale” offers on the Internet, especially in web forums. Otherwise, skilled criminals often choose another dangerous option: steal an existing botnet.

To continuously increase their gains and let botnets become a booming industry, criminals need always new infected hosts, and this is typically done by sending spam email, posting messages on web forums and social network sites, or via drive-by downloads.

2.1.2 Infection strategies

In recent years, botnet infections have incredibly increased and today zombie hosts are one of the most critical problems in the Internet. Also the FBI recently announced an operation called Operation Bot Roast, revealing that over 1 million hosts have been compromised in the United States and are now controlled by botnets. In those counting process they report that are included enterprise workstations as well as home desktops. With the increase of malware, spyware and rootkits samples and the use of Web 2.0 applications such as Twitter, Facebook and MySpace, the possibility of an host of being infected has increased exponentially. Typical infection vectors in fact relay on various social engineering techniques, especially when dealing with spam emails or posted messages on web forums and social network sites in order to deceive potential victims and let malware infect their systems.

Unfortunately, there are a number of existing exploitation strategies for installing malware on a user’s computer. There are common technique such as remotely exploiting vulnerable network services and lots of new schemes.

For example, a recent tactic, used by criminals, consists in offering an interesting video to view, which requires downloading a special codec. Of course, the user wants to be able to watch the video and, after downloading, launches the malicious file downloaded. Probably the user immediately does not notice any changes in his system, nor he suspects of the infection, but at the same time the host will be infected. From this point on, his computer is a botnet member, that, as a servant, can perform any action the botnet owner commands.

An equally potent alternative is to simply lure web users to connect to malicious web servers that subsequently deliver exploits targeting vulnerabilities of web browsers or of their plugins as in the case of drive-by-download attacks. Drive-by-downloads start when the exploits are executed and the browsers connect to a botnet relate malware distribution site to retrieve malware executables.
The number of these drive-by-download attacks has tripled in the last year and they are beginning to affect government web sites as well as small business organizations. These undesired downloads are possible because little pieces of code called “exploits” use vulnerabilities to stealthily execute code instructions not only to download other programs, but also to execute malicious programs without the user’s knowledge. If these attacks are successful, the user will not even suspect that there is something wrong with his computer. Alternatively, the malware itself can include self-replication functionality, like older viruses or worms.

All these malware distribution methods are becoming particularly dangerous: typically malicious hackers manage to infect lots of popular web pages with a potent exploit cocktail that targets a variety of vulnerable browsers to secretly install a malware sample on visitor machines.

One of the fundamental problems with web-based attacks is that those web sites can serve as primary infection points where even legitimate web sites can infect the internal network. Further, every day new web sites are created and search engines can make countless numbers of new web sites available in real time; many conventional web-filtering solutions adopted are not able, in time, to discern malicious from legitimate websites.

Another known modus operandi used by attackers to infect websites consist into injecting malicious iframes in vulnerable pages by exploiting SQL injection vulnerabilities. Once injected in each web page, the iframe silently load a javascript piece of code that silently runs while victims are visiting the infected website. Exploiting a vulnerability, this javascript code could both install a malware into the visitor’s system or try to steal sensitive information trough classical cross-site-scripting (XSS) attack.

Malware infected hosts, being now part of the botnet, can be actively used to infect other websites: attackers to this end install on new bots their SQL injection attack or spamming tools.

## 2.2 Web applications attacks

Attacks on web applications are on a constant change. Attackers are targeting web applications with different goals and every vulnerability has its own consequence. In this section we report some of most critical web application security flaws commonly used by attackers in relation to botnets activities. We introduce attacks like cross-site-scripting (XSS) and Cross-site request forgery (CSRF) that target the applications’ user, and others attacks like SQL injections (SQLIA) that primarily target the web application itself.
2.2.1 SQL injection

SQL injection techniques exploiting applications have been around for over 10 years now, but recently with web applications have seen a dramatic increase in both number of attacks and the complexity and damage caused by them.

A typical SQL injection attack consists of insertion of unwanted input data coming from untrusted sources in a SQL query executed by the underlying DBMS. Web applications in fact may use user-supplied input to create custom SQL statements for dynamic web page requests. When the application fails to properly sanitize user-supplied input, an attacker could alter the construction of SQL statements and control the execution of the queries. SQL injection is today the number one exploit used by hackers and criminals to steal information and deface websites.

Nowadays every kind of information on the web are logged into databases: shopping, banking or other kind of data. In most cases, sensitive information are stored in a relational database along with other user details and related information. The proliferation of databases expressly created to support this growth of information and websites with sensitive content has lead to many insecure sites, hastily published with few checks to the underlying security requirements.

A successful SQL injection attack can lead the attackers to be able to read sensitive data from the database, alter stored data, execute advanced administration operations on the database, sometimes recover the content of the file system and in the worst cases issue commands to the underlying operating system. The impact of this attack can allow attackers to gain total control of the database or the entire system.

SQL injection attacks have become widespread as criminals involved in botnet increasingly target legitimate websites injecting Iframes that redirect victims to other malicious servers. Those servers silently attack the visitor’s PC, often trying multiple exploits, and if at least one works, download additional code to the machine to add it to the group of already infected systems (bots). The botmasters behind recent botnets, as for example Asprox, used SQL injections attacks in order to inject their fast-flux malicious domains in vulnerable sites to increase the availability and survivability of their malicious infrastructure.

2.2.2 Cross-Site-Scripting

First cross-site-scripting vulnerabilities date back to 1996 during the early days of rich web pages on the Internet. Cross-site-scripting (XSS) is still today one of the most popular access for attacks against the internal network. Researchers have recently presented and XSS attack wherein, while the victim was watching a malicious website, they could actually scan internal network, fingerprint all the
web enabled devices and send attacks or commands to those devices. Nowadays, XSS remains the most popular attack against the masses, because it can be very easy to find the vulnerability and launch the attack, while the consequences can be devastating.

XSS attacks are a type of injection attack in which malicious scripts are injected into the otherwise benign and trusted web pages. Classical XSS occurs when an attacker uses a web application, typically a web page, to send malicious code to a victim, generally in the form of a client side script.

The victim’s browser has no way to know if the script he will execute is trusted, and thus it will execute the script. Because it comes from a trusted source, the same domain of the page, the malicious script can access any cookies, session tokens, or any other sensitive information saved by the browser and that can be used within that site. These scripts can also access to the Document Object Model (DOM) structure of the page, being able to rewrite the content of the current HTML page.

Basically XSS is an input validation issue that lets a malicious user inject and run scripting code in victim browser within the trusted context of compromised site. Injected scripting code typically sends victim authentication cookies to a malicious site so they can be used as authentication tokens to impersonate the victim. In a fair amount of cases, especially in older web applications, stolen cookies can easily lead to accounts hijacking. This happens when the cookie is used as the only authentication token, holding verification information only on the client side and nothing is tracked and saved on the server side.

Mitigation and active defense strategies against XSS fall both to content and web application developers, and to browser designers. Users could disable scripting execution, but most of today “rich” web application must be executed enabling scripting functionality. Several best practices have been proposed to content and application developers, but sanitization problems still persist. Web applications must still be deeply tested and source code reviewed before released.
Related work

The problem of mitigating malicious activities is a well-known research topic in computer security as well as the challenge of remediating the effects of an infection from a compromised system. This chapter presents the work done by other authors related to the topics presented in this dissertation.

3.1 Fast-flux networks detection

The problem of detecting botnet has been so far approached by the research community mainly from two different perspectives, from the prospective of the bot, by studying its code and its behaviors, and from the prospective of the network, by studying the traffic generated by these bots. The approach proposed in this dissertation studies the phenomenon from the prospective of a victim of the scams perpetrated through these botnets.

Fast-flux analysis. The first analysis and characterization of fast-flux service networks was presented by the HoneyNet project[95]. The report analyzed the two types of networks seen so far (i.e., single-flux and double-flux service networks) and analyzed the behavior of a malware with the capabilities of a fast-flux agent. After that a number of approaches for detecting fast-flux domain names have been studied and developed. Thorsten et al. [41] presented an empirical study of fast-flux service networks and developed a metric that try to exploit the principles of fast-flux service networks to derive a mechanism for detecting new fast-flux domains in an automated way. Their work differs from our in the number of features used to characterize fast-flux domains and in the classification algorithms used. Our method is based on several features that let our system to be more accurate and able to correctly classify malicious fast-flux domains. Our
system is also completely unsupervised: we do not need to develop and validate a metric with which FFSNs can be effectively detected. Although the work of Rajab et al. [81] differ from our contributions in term of techniques and goals, the point of view from which botnets are observed is similar to ours. We detect and monitor fast-flux service networks by performing active probes: simple DNS and WHOIS queries to collect information about the set of resolved IP addresses. Similarly, in their work, Rajab et al. tracked botnets by infiltrating in IRC channels and by measuring the cache-hit rate of the DNS servers queried by the bots to contact their control center. Recently Konte et al.[54] examined the role of fast-flux in some hosting online scams observed at a private spam trap and performed content-based scam campaign clustering. Nazario et al.[67] also investigated the behaviors of botnets behind fast-flux, Shin et al. the malicious infrastructure behind Asprox botnet, one of the most recent botnet using fast-flux as counter-countermeasures to defend itself from take-down and detection attempts. Perdisci et al.[77] in their ACSAC ’09 paper propose a novel, passive approach to detect and track malicious fast-flux service networks. Contrary to previous works that mainly studied domains advertised through email spam, they extract suspicious domain names from recursive DNS traffic collected at two large ISP networks. Castelluccia et al.[12] in their last work handled the problem of geolocalize fast-flux servers, that is, determine the physical location of the fast-flux networks motherships. Their framework is based on network measurements and by experimentations they showed that fast-flux servers can be localized with similar mean distance errors than non-hidden servers, i.e. approximately 100 km.

We have been also in touch with the ICANN Security and Stability Advisory Committee involved in fighting fast-flux operations at the registrar level. They recently completed a study[86] which describes the fast-flux technique to establish best practices to mitigate this phenomenon.

**Network-based techniques.** The analysis of the network traffic generated by compromised machines transformed into bots and the traffic generated by bot “management” open several opportunities for understanding the phenomenon and for detection. Rishi, by monitoring the network traffic for unusual IRC communications like connection to uncommon servers and ports and use of suspicious nicknames, detects machines infected with bots [36]. Karasaridis et al. developed a transport and application layer traffic analyzer to detect IRC-based bots on wide-scale [51]. Cooke et al. studied the effectiveness of detecting botnets by directly monitoring IRC communications and other command and control activities. Unfortunately their work demonstrated that a more comprehensive approach, based on the correlation of data coming from multiple sources, is required to precisely detect botnets. BotHunter correlates alerts coming from different types of
sensors to identify the communication sequences that occur during the infection process (i.e., target scanning, infection exploit, binary egg download, and out-bound scanning) [37]. Dagon et al. used DNS redirection to detect machines part of specific botnets and to understand how time and geographical location affect the spread dynamics [23] of these botnets. The problem of understanding how challenging is to estimate the size of botnets were addressed by Rajab et al. [82]. A similar problem was subsequently addressed by Dagon et al. [22]. They proposed several metrics to measure the utility of botnets for various activities and presented a taxonomy of botnets based on these metrics and on the topological structure of the networks. Recently, Wurzinger et al. [101] presented a system able to detect bots, independent of any prior information about the control channels or propagation vectors, and without requiring multiple infections for correlation. They automatically generate and use botnet detection models, based on network traffic traces recorded from bot instances.

Host-based techniques. Many researches tried to study botnets by analysing how bots behave and how they are implemented. These bots can be analyzed using dynamic, static, or a hybrid dynamic and static analysis. BotSwat characterizes and detect the typical behaviors of bots using dynamic taint analysis [90]. Barford et al. statically analyzed the codebase of four of the most common IRC bots to understand their propagation methods, the mechanism used for their remote control, the delivery and the obfuscation mechanisms used [75]. Other specific bots have been thoroughly analyzed to understand the new techniques used and the best method to block them [14, 25, 80].

3.2 Mitigation of Web application attacks

Existing solutions for the automatic detection of security vulnerabilities in web applications can be classified into two broad categories: static and dynamic approaches.

Static approaches. Many different approaches have been proposed for statically detecting security vulnerabilities in web applications, here we report a short selection of some recent works that are closely related to our contributions. For example, Huang et al. proposed WebSSARI [45], a lattice-based static analysis algorithm for the intra-procedural analysis of PHP programs. WebSSARI is derived from type systems and typestate, and it does not track the value of string variables; this can lead to a high false positives rate. In [50, 49], Jovanovic et al. present Pixy, a static analysis tool that performs flow-sensitive, interprocedu-
ral and context-sensitive data-flow analysis on PHP applications. Pixy is quite efficient and precise, with a low false positives rate. Finally, in [97] the author propose a fully automated static technique for detecting SQLI vulnerabilities in PHP programs. Their approach consists in approximating possible queries that the application could submit to the database using context free grammars; then, they track how input coming from the user can influence these grammars. However, both these approaches do not support several features of the PHP language, most notably classes and dynamically generated code. In Phan, the framework we developed, we address these limitations with our on-line analysis engine.

Dynamic approaches. Probably, the first approach that employed dynamic techniques for the taint analysis of applications is Perl taint-mode [78]: the interpreter prevents the use of user-supplied data that has not been explicitly sanitized. The works presented in [79, 69] are very close to our on-line engine. Both solutions protect PHP application against injection attacks using taint analysis, with an average run-time overhead of $\sim 10\%$. Unfortunately, both these works propose a fully dynamic analysis solution; we believe a hybrid approach like the one discussed in this dissertation can further reduce their run-time overhead. Moreover, the taint propagation performed by CSSE [79] is too coarse-grained, as it is not able to propagate taint meta-information when character-level string operations are performed; Phan offers a more fine-grained taint tracking solution.

3.3 Malware effects: analysis and remediation strategies

In this section we briefly review the work done by the research community on malware detection and analysis. We also present some recent results that focus on execution of untrusted applications without any risk for the system and on the problem of testing malware detectors.

Malware detection and analysis. The traditional approach for the detection of malicious code is based on signature matching of various complexity [93]. A signature can be a sequence of bytes that identifies pieces of data or code of the malicious program, but even very complex algorithms that test whether a particular program satisfies certain properties. The advantage of using sophisticated detection methods is that signatures become more generic and thus a single signature can be used to detect multiple variants derived from the same family. On the other hand, from the remediation point of view, excessively generic signatures do not allow to distinguish variants. If single variants cannot be told apart, the
remediation procedure cannot take variant-specific behaviors into account and cannot perform a complete cleanup.

Purely signature-based approaches have demonstrated their weaknesses when packed, polymorphic and metamorphic malware appeared. The research community started to move toward behavior-based solutions. Behavior-based detection [16, 60] and analysis [7, 99, 65, 106] approaches do not focus on the syntactic structure of the analyzed program, but try to consider its semantics. Because these solutions work by observing a concrete execution of the malicious sample, they could provide much more accurate remediation procedures. Recently Kolbitsch et al. [17] proposed and effective and efficient malware detection method that can be used at the end host replacing or complementary to traditional antivirus software. This method is based on fine-grained models obtained by executing the malware program in a controlled environment, monitoring and observing its interactions with the OS resources; detection is done by matching extracted behavior models against the runtime behavior of unknown programs.

Execution of untrusted applications  In [44], Hsu et al. present a framework to automatically remove a malicious program from the system and also to repair any damage it could have done. The safe state of the system is restored by using the logs of the execution and by reverting each logged operation. An alternative approach is proposed by Liang et al. [56]. Untrusted programs are executed in a sandbox and the changes made to the “virtual” system are committed to the real one at the end of the execution, only if the program can be considered innocuous.

In the operating systems and self-healing communities, a number of different works investigate the problem of automatically reverting the modifications made by an unwanted program. As an example, in [71] the authors present Speculator, a modified Linux kernel that allows speculative execution of user-space processes. Speculator avoids blocking user processes during slow I/O operations (such as remote I/O operations): the system predicts the operation’s result, checkpoints the process and allows it to continue; later, if the prediction is found to be incorrect, the process is reverted to the checkpointed state.

Evaluation of state-of-the-art malware detectors  The need for automatic testing methodologies targeting anti-malware products has been clearly stated by the Anti-Malware Testing Standards Organization (AMTSO) [4]. However, little research work focuses on the evaluation of malware detection and remediation solutions. One of the few examples is represented by [15]; in this paper, Christodorescu et al. present a technique for generating test-cases to stress malware detectors. They use program obfuscation techniques to evaluate the resilience of malware detectors to various transformations of the malicious code.
Recently Oberheide et al., the authors of CloudAV[73] (in-cloud antivirus system), proposed the creation of an online automated service, called PolyPack[48], that uses an array of packers, software tools for evading the detection capabilities of classical signature-based antivirus engines, and top rated antivirus engines as a feedback mechanism to select the packer that will result in the optimal evasion of the antivirus engines. The goal of our contribution instead is to estimate the completeness of remediation procedures. For this reason, the testing infrastructure described in Chapter 6 could complement all presented works, in order to produce more comprehensive testing methodologies.

Other researchers highlighted the importance cleaning infected systems and the importance of testing such functionality [11, 64]. Motivated by the same convictions, the 6th Chapter contributes to address this problem by proposing a fully automated testing methodology and an extensive evaluation of several state-of-the-art commercial products.

Automated signature generation. The generic models from which we start the construction of remediation procedures can be viewed as generic signatures for the effects of the malicious program on system resources. For this reason, the approach proposed in Chapter 7 is also related with the topic of automated signature generation. Different approaches have been developed for the automated generation of attack signatures [68, 55]. As an example, Nemean [105] is a system for the automated generation of intrusion signatures from a network packet trace. Nemean starts by abstracting single network packets are abstracted into sessions, and then similar sessions are grouped together in a single cluster. Finally sessions clusters are abstracted into generic attack signatures, using automata learning techniques. This analysis methodology is very similar to ours, but the context where it is applied is completely different.
4.1 Introduction

One of the most dangerous phenomenon we are observing today on the Internet is the unprecedented spreading of malware, a program written with malicious intents. Today, the main motivation behind malware writing and their use is the easy financial gain. Smart miscreants write malware and sell them in the wealthy underground market to other miscreants [32]. These malicious programs are installed on machines all around the world, without any permission of the users, and transform these machines into bots, i.e., hosts completely under to control of the attackers. Bots are then used to steal computational resources and confidential information, to relay spam email messages, to mount distributed denial of service (DDoS) and other attacks, to host phishing websites, and for other kinds of scams. To maximize the profit from these activities, multiple infected machines are grouped together in a botnet (a network of bots) and used simultaneously to achieve the same purpose [94]. With a single command, miscreants can control hundreds or even thousands of bots [82]. The botnet problem is so extensive nowadays that it has made headlines several times [33, 58].

The most well known botnets are those related with the Warezov and the Storm worms and recently Waledac, Asprox, Conficker and Torpig [20, 80, 88, 91]. These botnets are infamous for the huge amount of spam emails they have been generating or for the massive SQL injections attacks performed, both containing links to malicious web servers hosting various frauds as well as malicious web pages able to infect the machines of the visitors with malware. Of particular interest is the technique used by those botnets to masquerade the identity of the malicious web servers in order to maximize the availability of the service. If these web servers are difficult to identify, they are difficult to shutdown, and they can hit more and more victims. This technique, known as fast-flux service network, is
very simple and consists in associating the canonical hostname of a malicious web server (e.g., www.factvillage.com) with multiple IP addresses corresponding to the addresses of a subset of the bots of the botnet. Each victims’ request to visit the web server will thus reach one of the bots and the bot will proxy the request to the real server, making impossible to discover the identity of the malicious web server without having full control of one of these bots. The association between the hostname of the web server and the IP addresses of the bots acting as front-end proxies is updated very frequently such that newly compromised machines can immediately take part in the game and dead bots are excluded without affecting the availability of the service [95].

The impact that botnets using fast-flux service networks have on the Internet community is tremendous [35, 88]. Although the average lifetime of domains used for malicious purposes, including the domains associated with fast-flux service networks, is very short, the lifetime of botnets using those domains is much longer. As the identity of the hosts associated with those domains is well protected and the bots that are part of the networks are difficult to track, botnets are difficult to eradicate. Authorities put a lot of efforts to take down the domains registered for malicious purposes, but these efforts are worthless because the bots are not isolated. Before the domain is suspended, a new one is registered and associated with the same set of bots, to replace the old one. Consequently, miscreants can continue their malicious activity through their botnets without interruption. Fighting miscreants that control botnets is everyday harder: recently they started using a new hiding strategy, called domain flux, in order to protect their command and control servers. With domain flux, each bot periodically generates a list of domain names to contact; only the first domain that reply is identified as a valid C&C server until the next domain generation round. With this improvement the attacker only needs to register one of these domains to keep connection with all the bot, and for defenders it’s very hard to predict and take down all generated malicious domains. Both Torpig and Conficker relies on domain flux not only to hide their C&C servers, but also to generate domain names used in the spreading phase (through drive-by-download attacks).

Two main strategies are adopted for fighting against botnets. The first one aims at individuating hosts belonging to a botnet by passively analyzing the network traffic, a fundamental task for eradicating the botnet, but unfortunately, that requires the access to a significant network segment [22, 36, 37, 51, 23, 83, 19].

The second one aims at individuating scam domains and shutting them down. Unfortunately, botnets adopting fast-flux service networks are immune to both of these countermeasure. First, the controlling elements of the network are accessed using a set of bots as proxies, and the components of such a set are changed very frequently; in this way, the core components of the network are masqueraded
by an additional layer of indirection. Second, any scam site is associated with multiple fully-qualified domain names (FQDNs), which we call fast-flux domains. Consequently, the site is immune to domains closure: if one or more domain names are shutdown, the availability of the botnet is not compromised, as miscreants can always register new domains and divert malicious agents to them.

Observing the behavior of fast-flux botnets, we individuated some distinguishing features that can be exploited for building a fast-flux botnet detector. To understand the idea behind our approach, consider a victim, i.e., an host $h$ of a botnet; although the visibility $h$ has on the botnet is generally quite limited, this is not true in the case of a fast-flux botnet. In such a case, $h$ is likely to access the scam site associated to the botnet through different bots for each visit (recall that in fast-flux botnets the canonical hostname of the scam site is resolved into the IP address of one of the bots). After a large number of visits (as those done by a recidivous victim), $h$ will have discovered the IP addresses of a significant portion of the active bots of the botnet. Thus monitoring many victims will provide us a lot of information regarding the botnet components. On the basis of such a principle we built our detector named FluXOR. Given a suspicious domain name, FluXOR, by behaving like a recidivous victim, tries to detect if the domain conceals a fast-flux service network and, if this is the case, it continuously monitors the domain name to find out all the associated IP addresses. The ultimate intent is to identify all the compromised machines that are part of the botnet associated with the fast-flux service network. FluXOR’s detection strategy is driven by the combined analysis of nine distinguishing features describing some properties of (i) the domain the suspicious hostname belongs to, (ii) the degree of availability of the potential fast-flux service network, and (iii) the heterogeneity of the potential hosts of the network.

Furthermore, as any scam site may be associated with multiple fully-qualified domain names, we developed and implemented inside FluXOR a strategy that, given a set of fast-flux domains, groups together those which take part to the same scam fraud, thus individuating current scam campaigns. Such a strategy is based on the supposition, proved by the empirical observation, that malicious FQDNs that belong to the same botnet host very similar scam pages, both in terms of the products that are advertised and the syntactic structure of the web pages. The similarity between these fraudulent pages is computed using document resemblance techniques and is made more efficient through the employment of probabilistic counting algorithms.

We have been using FluXOR since the beginning of January 2008 to monitor potential fast-flux service networks whose hostnames were collected from spam emails. In our experiments we have been able to infer existing relationships between fast-flux botnets and spam campaign. In the first month the sys-
tem correctly classified all the analyzed hostnames (4961) and 7.8% of them (378) turned out to be associated with fast-flux service networks, involving 31998 distinct compromised ip addresses located all around the world. Two month later malicious domains analyzed were 9988, corresponding to 162855 different ip addresses collected during time. We developed a web front-end to access real-time results gathered and analyzed by FluXOR that is available on-line at \url{http://fluxor.laser.dico.unimi.it}.

To summarize, this chapter makes the following contributions:

- identification of the features that, combined together, allow to precisely detect whether or not a suspicious hostname conceals a fast-flux service network (Section 4.4 and 4.5).

- Implementation of a strategy to monitor a fast-flux service network and to detect the majority of the bots that are in the network (Section 4.7).

- An efficient method for grouping together fast-flux domains serving the same scam site. This allows for the identification of all malicious bots serving the network, so that they can be subsequently isolated and cleaned. In this way, scam botnets using fast-flux networks can be effectively eradicated.

- Empirical analysis of the fast-flux service network phenomenon (Section 7.5).

### 4.2 Problem description and solution overview

A fast-flux service network is a network of malicious hosts, mostly compromised, that is used to carry out malicious activities, for example to deliver malware to users, to distribute illegal materials or to steal users’ credentials [95]. The service network is identified by one or more fully qualified domain names (FQDNs) that are resolved to multiple (hundreds or even thousands) different IP addresses, belonging to unaware compromised hosts, the fast-flux agents (or bots). The fundamental characteristic of a fast-flux service network is high availability, which is provided by continuously updating the pool of agents serving the network. Newly compromised hosts are inserted into the network, inactive or unreliable hosts are removed, and victims are always redirected to the active and most reliable agents. The key is a combination of a very short time-to-live (TTL) of the DNS resource records that associate the canonical name of the service network with the set of IP addresses of the agents and a round-robin selection of these records [62, 63]. In the common setup, the agents do not carry out the malicious activities, but they simply redirect received requests to the fast-flux mother-ship,
Figure 4.1: An example of the fast-flux service network used by our sample malicious web server www.factvillage.com, the entities involved and the communication between these entities (nodes in blue and red denote hosts under the control of the miscreants and shaded agents denote those that are not currently serving the network).

The controlling element of the network, whose identity must be kept secret. With this setup, it is not possible to identify the mother-ship without having complete control of one of the agents.

Imagine that the fully qualified domain name www.factvillage.com conceals a fast-flux service network composed of hundreds of agents and that it is used to attract users, with the promise of very cheap drugs, and to infect their machines with malware. Figure 4.1 shows how our sample malicious contents provider leverages the fast-flux service network to serve the victims. A victim, wishing to visit the on-line drugstore, queries a name server (usually a non-authoritative name server which recursively queries the authoritative one) to resolve the hostname of the website. The name server returns the addresses of a subset of the agents currently active in the network, and the victim connects to one of them. The agent then proxies the victim’s requests to the mother-ship, which in turn delivers the malicious contents. In background, the mother-ship, or another entity controlled by the miscreants, continuously monitors the status of the agents and updates the resource records of the authoritative name server of the domain (in the example the authoritative name server is ns0.uthvfybz.com), to distribute the network across the reliable agents. The short time-to-live associated to the DNS resource records prevents non-authoritative name servers to cache for too
long the records that define the subset of agents currently serving the network. When the cache expires the name server contacts again the authoritative name server for the domain and gets the new list of agents serving the network. These agents are selected from the set of all active agents in a round-robin fashion[9], to balance their load.

**Web scams using multiple fast-flux domains.** In a fast-flux service network, the shutdown of the mothership is made very difficult by the fact that its identity is protected by the agents. Furthermore, the service network can increase the availability by using multiple motherships. If authorities isolated and analyzed some agents, finding out the identity of a controlling element and shutting it down, miscreants could still perpetrate their activities with the remaining agents and motherships. The Achille’s heel of fast-flux service networks seems to be their domain name. Unfortunately this weakness is just apparent. Even though the domains were shutdown, miscreants would continue to control all the agents of the network. Indeed, they just have to register a new domain name for the service network to continue their malicious activities exactly as before. To perpetrate these activities without the minimal interruption, criminals periodically register multiple new domain names and advertise them (e.g., through spam emails) before older ones are shutdown. Figure 4.2 shows an example of two fast-flux service networks used concurrently for the same fraud.
4.3 FluXOR: an outline

Our goals are, given a fully qualified domain name, to precisely tell if it conceals a fast-flux service network and, if it does, to identify all the agents that are part of the network as well as to individuate and group together the different malicious FQDNs which are part of the same fast-flux botnet. The prompt identification and isolation of all the agents is important because if the service network seems shutdown, but the agents remain under the control of miscreants, a new service network of the same extent can be created by simply registering a new domain and reusing the same agents. Moreover, these agents can be used for other malicious purposes (e.g., they can be used as DDoS zombies, to steal personal information from the hosts they are running on, and to act as spam bots).

The key idea behind FluXOR (the system we have developed to accomplish these goals) is that a fast-flux service network has multiple distinguishing features that are not typically found in benign fully qualified domain names. Intuitively, some of the most characteristic features are (i) the time-to-live of DNS resources records, (ii) the large number of IP addresses into which the canonical hostname is resolved, and (iii) the heterogeneous set of organizations that own these addresses. Clearly, these features taken singularly are not enough to distinguish between benign and malicious hostnames. As an example let us compare our sample malicious hostname **www.factvillage.com** with the benign hostname **database.clamav.net**. The latter is a typical example of how DNS resources records with very small time-to-live and round-robin can be used to distribute the load across multiple mirrors (in this case the mirrors are used to distribute updates for the database of signatures of the ClamAV anti-virus [53]). Moreover, as mirrors are hosted by universities and companies, the hosts running a mirror belong to different networks, owned by different organizations, and are distributed around the world. Despite hosts like **database.clamav.net** have most of the characteristics of a fast-flux service network, FluXOR, by monitoring the suspicious hostname for a small period of time and by combining the extracted features using a naive Bayesian classifier [46], can precisely distinguish between hostnames that are associated with fast-flux service networks and those that are not. It is worth nothing that the chosen approach works well also when some of the selected features are not available.

When a fast-flux service network is detected, FluXOR continuously monitors the service network, behaving like a victim and periodically querying various DNS servers to resolve the canonical name of the network for the purpose of enumerating the IP addresses of the compromised hosts that, even for a small period of time, are used as agents.

A fast-flux service network, like the one described in this section, is known
in the literature as a single-flux network. More complex setups are possible, an example is a double-flux network [95]. FluXOR handles indifferently any kind of fast-flux service network, but in the results presented we do not distinguish between the various types.

In order to identify and group together those networks that are part of the same fast flux botnet we clusterize fast-flux networks into categories that group together similar scam sites. Ideally, two scam sites should be considered similar if they advertise the same products. In our case we relax the problem of computing this “semantic resemblance” by approximating it with the simpler notion of “syntactic resemblance” between the HTML pages hosted on the malicious web sites (see also [5, 96, 108]). Even if quite simplistic such an approximation works pretty well in practice: scam sites that are part of the same scam botnet are actually almost identical, as they usually differ only for minor syntactic details.

For the remaining of the chapter, for conciseness, we will refer to the FQDNs associated with a fast-flux service network as malicious and to all the others as benign, although what in the chapter is considered benign could be an hostname created for other malicious purposes but not associated with a fast-flux service network. Moreover, we will refer to any hostname whose maliciousness has not been established yet as suspicious.

4.4 Characterizing fast-flux service networks

The features used by FluXOR to distinguish between benign and malicious hostnames are summarized in Table 4.1 and discussed in detail in the remaining of the section. The features are grouped in three categories: (i) features characterizing the domain name to which the suspicious hostname belongs to, (ii) features characterizing the degree of the availability of the network that is potentially associated with the suspicious hostname, and (iii) features characterizing the heterogeneity of the potential agents of the network. Some of the features might appear similar initially, but, as shown later, each of them tells us something important about the suspicious hostname, especially because some features might not be always available and because there is no well known convention about how some of them are attributed.

4.4.1 Features characterizing the domain name

Domain age (F_1). Benign domains are usually characterized by a relatively long age. Domains used for malicious purposes instead are typically active only for short periods of time. As soon as they are identified, they are deactivated by the authority in charge of the corresponding top-level domain. Thus, miscreants
Table 4.1: Summary of the features used to distinguish between benign and malicious hostnames, grouped by category.

<table>
<thead>
<tr>
<th>Category</th>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain name</td>
<td>F₁</td>
<td>Domain age</td>
</tr>
<tr>
<td></td>
<td>F₂</td>
<td>Domain registrar</td>
</tr>
<tr>
<td>Availability of the network</td>
<td>F₃</td>
<td>Number of distinct DNS records of type “A”</td>
</tr>
<tr>
<td></td>
<td>F₄</td>
<td>Time-to-live of DNS resource records</td>
</tr>
<tr>
<td>Heterogeneity of the agents</td>
<td>F₅</td>
<td>Number of distinct networks</td>
</tr>
<tr>
<td></td>
<td>F₆</td>
<td>Number of distinct autonomous systems</td>
</tr>
<tr>
<td></td>
<td>F₇</td>
<td>Number of distinct resolved qualified domain names</td>
</tr>
<tr>
<td></td>
<td>F₈</td>
<td>Number of distinct assigned network names</td>
</tr>
<tr>
<td></td>
<td>F₉</td>
<td>Number of distinct organizations</td>
</tr>
</tbody>
</table>

have to register new domains and start to use them right away, to successfully achieve their malicious purposes. The average age of a benign domain is much older than the average age of malicious domain. Indeed, during our experiments, we have estimated that the average age of malicious hostnames is less than five weeks.

**Domain registrar** (F₂). We empirically observed that most of the domains used to implement fast-flux service networks are registered through a limited number of registrars, typically located in countries with a lax legislation against cyber-crime. Our hypothesis is that these registrars perform almost no check when domains are registered. Miscreants can easily complete the registration process using false identities and paying with stolen credit card numbers, making impossible, for the authorities, to identify the person who has effectively registered a domain. On the other hand, the set of registrars used to register benign domains is more heterogeneous and is not likely to overlap with the set of registrars used by miscreants.

### 4.4.2 Features characterizing the degree of availability of the network

**Number of distinct DNS “A” records** (F₃). Fast-flux service networks are generally composed by a large number of agents. The authoritative name server for the malicious domain, when queried, returns the set of active agents (i.e., the subset of agents currently serving the network) by returning multiple DNS “A” records, each one containing the IP address of a specific agent. These resource records are periodically updated by the fast-flux mother-ship to put in
the network newly compromised agents and to remove the faulty ones. Thus, after a reasonable long span of time, the number of distinct DNS records of type “A” (i.e., agents IP addresses) that had or have been associated with a malicious FQDN is rather large. The higher the number of distinct DNS records of type “A” associated to the same FQDN, the larger the number of potential agents, and the higher the probability that the FQDN conceals a fast-flux service network.

Time-to-live of DNS resource records ($F_4$). The fundamental characteristic of fast-flux service networks is the high frequency at which the set of active agents is updated. Most of the agents are end-user machines and consequently it is reasonable to expect that they will appear on-line and disappear very frequently. Thus, to guarantee the high availability of the service offered through the fast-flux network, the set of active agents has to be updated as soon as one of them changes its state. Moreover, the update must be promptly propagated across the Internet, down to the victims. To achieve this goal, the authoritative name server for the malicious domain associates a very short time-to-live to the DNS resource records of the domain. That forces non-authoritative name servers, used by the victims, to flush their cache and to query the authoritative name server very frequently, that in turn returns a different set of active agents every time. The higher the time-to-live associated to the various DNS resource records of a domain, the lower the probability that the domain is malicious. Unfortunately the converse is not always true. Several authoritative name servers for benign domain names associate very short time-to-live to their records for various purposes.

4.4.3 Features characterizing the heterogeneity of the agents

Number of distinct networks ($F_5$). Fast-flux agents are usually randomly compromised hosts scattered all around the globe. Thus, a malicious FQDN is resolved to many different IP addresses belonging to hosts that very likely belong to different networks. On the other hand, when a benign FQDN encompasses multiple hosts, for load-balancing purposes, these hosts often belong to the same network because they are owned by the same company and physically very close to each other. The higher the number of distinct networks associated to the same FQDN, the more scattered the hosts are, and the more likely these hosts have been compromised and have been used as fast-flux agents. As an example compare the networks associated with the benign FQDN hp.com with those associated with the malicious FQDN www.factvillage.com, reported respectively in Table 4.2(a) and Table 4.2(c). The IP addresses associated with the former all belong to the same network (15.0.0.0/8), while the addresses associated with the latter belongs
Table 4.2: Comparison of the host specific features (F₅ to F₉) characterizing two benign and one malicious FQDNs (the entries in bold are those common to multiple IP addresses).

(a) hp.com (benign)

<table>
<thead>
<tr>
<th>IP address</th>
<th>F₅</th>
<th>F₆</th>
<th>F₇</th>
<th>F₈</th>
<th>F₉</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.216.110.140</td>
<td>15.0.0.0/8</td>
<td>AS9218</td>
<td>polyserve.com</td>
<td>HP-INTERNET</td>
<td>Hewlett-Packard</td>
</tr>
<tr>
<td>15.192.45.22</td>
<td>15.0.0.0/8</td>
<td>AS9218</td>
<td>polyserve.com</td>
<td>HP-INTERNET</td>
<td>Hewlett-Packard</td>
</tr>
<tr>
<td>15.200.30.24</td>
<td>15.0.0.0/8</td>
<td>AS9218</td>
<td>polyserve.com</td>
<td>HP-INTERNET</td>
<td>Hewlett-Packard</td>
</tr>
</tbody>
</table>

(b) www.avast.com (benign)

<table>
<thead>
<tr>
<th>IP address</th>
<th>F₅</th>
<th>F₆</th>
<th>F₇</th>
<th>F₈</th>
<th>F₉</th>
</tr>
</thead>
<tbody>
<tr>
<td>67.228.112.196</td>
<td>AS36351</td>
<td>avast.com</td>
<td>SOFTLAYER-4-5</td>
<td>SoftLayer Tech.</td>
<td></td>
</tr>
<tr>
<td>216.12.205.130</td>
<td>AS36420</td>
<td>avast.com</td>
<td>EVRY-BLK-4</td>
<td>Everyone Internet</td>
<td></td>
</tr>
<tr>
<td>74.86.245.119</td>
<td>AS36351</td>
<td>avast.com</td>
<td>SOFTLAYER-4-4</td>
<td>SoftLayer Tech.</td>
<td></td>
</tr>
</tbody>
</table>

(c) www.factvillage.com (malicious)

<table>
<thead>
<tr>
<th>IP address</th>
<th>F₅</th>
<th>F₆</th>
<th>F₇</th>
<th>F₈</th>
<th>F₉</th>
</tr>
</thead>
<tbody>
<tr>
<td>61.18.66.7</td>
<td>61.18.0.0/16</td>
<td>AS99908</td>
<td>hkcable.com.hk</td>
<td>HKCABLE-HK</td>
<td>HK Cable TV</td>
</tr>
<tr>
<td>218.47.195.7</td>
<td>218.47.0.0/16</td>
<td>AS4713</td>
<td>ap.plala.or.jp</td>
<td>PLALA</td>
<td>Plala Net. Inc.</td>
</tr>
<tr>
<td>81.173.151.10</td>
<td>81.173.151.0/24</td>
<td>AS8422</td>
<td>netcologne.de</td>
<td>NC-DIAL-IN-POOL</td>
<td>NetCologne</td>
</tr>
</tbody>
</table>

Chapter 4. FluXOR

Number of distinct autonomous systems (F₆). An autonomous system (AS) is a connected group of one or more IP prefixes run by one or more network operators with a single and clearly defined routing policy [39]. Thus, distinct networks, but physically very close, might be connected to the Internet through the same AS. As with the previous feature, the majority of benign FQDNs are mapped to hosts located in a circumscribed geographical area and are all part of the same autonomous system. On the other hand, as the agents of a fast-flux network are scattered across all the countries, they typically belong to distinct autonomous systems. As an example let us compare the autonomous systems associated with the benign FQDN www.avast.com, with those associated to www.factvillage.com (Tables 4.2(b) and 4.2(c) respectively). In the first case we have three distinct networks but only two autonomous systems. In the second case, each host, as located in a different country, is part of a different AS.

Number of distinct resolved qualified domain names (F₇). Even if a FQDN is associated with multiple hosts scattered around the globe and part
of distinct networks and autonomous systems, the hosts might still be owned by the same company or organization and thus they can share the same qualified domain name. As an example let us compare the benign FQDNs of Tables 4.2(a) and 4.2(b) with the malicious www.factvillage.com of Table 4.2(c). In the first two cases both hostnames are resolved into multiple IP addresses, but these addresses are in turn resolved into canonical hostnames belonging to the same domain (i.e., polyserve.com and avast.com respectively). The example of www.avast.com clearly indicates that all the IP addresses found are legitimate. Unfortunately, that is not completely evident in the case of hp.com because the domain name (polyserve.com) does not match the domain name of the suspicious FQDN under analysis. Nevertheless, all the IP addresses found are part of the same domain, which is not common for malicious FQDNs. Indeed, fast-flux agents are compromised hosts belonging to distinct organizations, and the canonical hostnames associated with their IP addresses are solely under the control of the respective owners of the networks and the attacker cannot control in any way these information. In the case of www.factvillage.com, each of the three IP addresses found, probably used by dial-up hosts, is resolved into a hostname with a distinct qualified domain, corresponding to that used by the ISP providing the service.

**Number of distinct assigned network names (F₈).** The network name is the name assigned to a network by the registration authority. Multiple network addresses can be logically grouped under the same network name. This is often the case when the different network addresses are owned by the same company or organization. Like the other three previous features, the number of distinct network names is an indication of the degree of scattering of the hosts associated with the suspicious FQDN.

**Number of distinct organizations (F₉).** Each network is assigned to an organization, but as with network names, same organization can own multiple networks with one or multiple names. As an example let us consider the benign domain avast.com analyzed in Table 4.2(b). Each network is assigned a distinct network name, but two of these networks belong to the same organization (i.e., SoftLayer Technologies Inc.). Clearly, fast-flux agents randomly distributed around the world share a limited number of organizations.
Table 4.3: Comparison of three sample benign and malicious FQDNs using the selected features ($F_1$ is measured in weeks) and comparison of the features of the average benign and malicious FQDNs (computed from a set of about 75 benign and 215 malicious hostnames monitored for about three hours).

### 4.5 Combining the features for detection

FluXOR initially monitors suspicious hostnames for a short period of time, after which the selected features are analyzed to determine whether the domain is malicious or not. The number of domains is incredibly growing. Indeed, it has been estimated that several hundreds of thousands of generic second-level domains (e.g., .com, .org, .net) are registered daily [26]. Consequently, the number of suspicious hostnames to monitor can be very large, it is essential that a precise classification can be accomplished in the shortest period of time, to reduce the workload of the system, but also to promptly intervene to mitigate the damage fast-flux service networks and their bots can cause to the Internet community.

Table 4.3 shows a comparison of the features of three FQDNs, associated with as many distinct fast-flux service networks, with those of three benign hostnames. Note that the features reported in the table were extracted after only three hours of monitoring. From a quick glance at the numbers in the table it should be clear that each of the selected features effectively tells us something important about the maliciousness of a hostname. Although it is easy to spot by hand benign and malicious hostnames, the numbers in the table show a high variability in the most intuitive features (e.g., $F_3$ and $F_4$). For all the analyzed hostnames reported in the table it was possible to extract all the selected features. In the general case some of these features might be missing, but nevertheless the system must be able to correctly discern between malicious and benign hostnames. Furthermore, hosts associated with malicious hostnames tend to be rather scattered, but the degree of the scattering and the number of fast-flux agents might depend on the amount of time the fast-flux service network has been active. If hosts are
compromised and turned into agents using a self-propagating malware (e.g., that identifies targets using weak random scanning), it is reasonable to believe that, in the early stage, the agents are rather localized and limited in number. Our goal is to be able to detect if a hostname is malicious as soon as possible, even when the number of agents involved is very small.

For these reasons the detector tries to achieve the best accuracy by combining the selected features using a naïve Bayesian classifier [46]. Given the features of a suspicious hostname, the classifier returns the class (i.e., benign or malicious) to which the hostname is most likely to belong to. The classifier was trained with a set of malicious and benign FQDNs that we manually classified, with the help of data obtained after a week of monitoring. The set of malicious hostnames was composed of hostnames found in spam emails. The set of benign hostnames was composed of hostnames found in spam and non-spam emails. Furthermore, the latter set was extended, to make it more heterogeneous, by adding the address of some randomly selected websites we recently visited. The assumption that the features are completely independent, made by this type of classifier, might appear to simplistic (e.g., features like $F_5$, $F_6$, $F_8$, and $F_9$ could be correlated). Nevertheless, this approach turned out to have very good performance in many real-world situations and the work of Zhang has shown that the efficacy of naïve Bayesian classifiers has some theoretical foundations [107]. In our context, as discussed later in Section 7.5, this approach gives very accurate results (for this reason we decided not to evaluate other classifiers). Our hypothesis is that, in practise, no real correlation between the alleged correlated features ($F_5$, $F_6$, $F_8$, and $F_9$) exists because no convention regulates how ISPs should partition their address space. For example the network associated with a single autonomous system ($F_6$) could be divided into sub-networks and multiple sub-networks ($F_5$) can be assigned to the same organization ($F_9$).

### 4.6 Clusterization of fast-flux service networks

As any scam site may be associated with multiple fully-qualified domain names, we need a strategy that, given a set of fast-flux domains, is able to group together those which take part to the same scam fraud, thus individuating current scam campaigns. In this section we deeply discuss the methods employed to estimate the resemblance between two scam pages, and how this metric can then be used to group together similar fast-flux service networks. In FluXOR we chose to relax the problem of computing this “semantic resemblance” by approximating it with the simpler notion of “syntactic resemblance” between the HTML pages hosted on the malicious web sites. In this section we present a method to quantify the syntactic resemblance between pairs of HTML documents.
4.6.1 Clusterization

To establish the similarity between different HTML documents, we rely on Broder’s notion of document resemblance [10], the more appropriate for our context. Informally, to each document \( d \) is associated a multiset of tokens \( S(d) \). For example, tokens can be single letters, words or lines. Then, given two documents \( d_1, d_2 \), their resemblance is defined as:

\[
sim(d_1, d_2) = \frac{|S(d_1) \cap S(d_2)|}{|S(d_1) \cup S(d_2)|}
\]  

From the above definition, \( \sim(d_1, d_2) \in [0, 1] \). When the resemblance is close to 0, it is likely that the documents are very different, while when the resemblance index is close to 1, the documents are “roughly the same”. This approach has the obvious drawback that the pairwise comparison of \( t \) documents has cost \( O\left(\sum_{i=1}^{t} |d_i|^2\right) \), where \( |d| \) is the size of document \( d \). Such a complexity is often unacceptable, especially when, as in our situation, the number of clustered documents keeps increasing over time, and some subsets of the documents have to be periodically re-clustered. For this reason, as suggested in [29], we employ probabilistic counting techniques to make the computation of document resemblance significantly more efficient, at the cost of an often negligible error. In particular, with probabilistic counting, the pairwise comparison of \( t \) documents becomes \( O\left(\sum_{i=1}^{t} |d_i|\right) + O(t^2) \) in complexity. Thus, we replace the resemblance function of equation 4.1 with the following probabilistic function:

\[
sim'(d_1, d_2) \approx \frac{||W(d_1) \oplus W(d_2)|| - ||W(d_1)|| - ||W(d_2)||}{||W(d_1) \oplus W(d_2)||}
\]  

where \( W(d) = R(S(d)) \), \( R(M) \) is a fingerprint of the multiset \( M \) that only depends on the set of distinct values in \( M \) [27], \( \oplus \) denotes the maximum component-wise, and \( ||M|| \) is a probabilistic estimation of the cardinality of \( M \) (in our case, \( R(M) \) consists of \( m \) byte values and results in an accuracy of \( \approx 1/\sqrt{m} \)). Our current approach uses the HyperLogLog [30] cardinality estimation algorithm to efficiently calculate the similarity of two web pages.

Besides its efficiency, this method has the undisputed advantage that, in order to calculate the resemblance between two documents, it is not needed to store the whole documents, but it suffices to just keep a short fingerprint of each page. In our current implementation, the algorithm has been tuned for an accuracy of \( \approx 2.3\% \) (when \( m = 2048 \)), requiring roughly two kilobytes for each document fingerprint, instead of \( \approx 20 \) kilobytes per document when storing the whole HTML page.
Input: $d$, an HTML document to be clustered, and $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ a set of existing clusters of HTML documents, where $\forall i \in \{1, \ldots, n\}, C_i = \{c_1, c_2, \ldots, c_w\}$ is a set of documents’ fingerprints.

1. $d' = \text{preprocess}(d)$
2. $y = \text{fingerprint}(S(d'))$
3. $C_{\max} = \bot$
4. $l_{\max} = \bot$
5. \textbf{foreach} $C \in \mathcal{C}$ \textbf{do}
   6. \hspace{1em} $x = \text{first}(C)$
   7. \hspace{1em} $l = \text{sim}'(x, y)$
   8. \hspace{1em} \hspace{2em} \textbf{if} ($l_{\max} = \bot$) $\lor$ ($l > l_{\max}$) \textbf{then}
   9. \hspace{4em} $C_{\max} = C$
  10. \hspace{4em} $l_{\max} = l$
11. \hspace{1em} \hspace{2em} \textbf{if} ($l_{\max} \neq \bot$) $\land$ ($l_{\max} > \tau$) \textbf{then}
12. \hspace{4em} $C_{\max} = C_{\max} \cup y$
13. \hspace{1em} \hspace{2em} \textbf{else}
14. \hspace{4em} $\mathcal{C} = \mathcal{C} \cup \{y\}$

**Figure 4.3:** Pseudo-code for the clusterization algorithm. The function \text{preprocess}($d$) applies the three preprocessing steps to the document, the function \text{fingerprint}(M) returns the fingerprint of the multiset $M$, and the function \text{first}(C) returns the first document fingerprint that has been inserted into $C$.

### 4.6.2 Clusterization algorithm

When a new document $d$ has to be clustered, for each existing cluster $C$, we calculate the resemblance between $d$ and $C$, and we keep the value of the maximum resemblance index. If the maximum index is greater than a given threshold $\tau$, we add the document to the cluster with that index. Otherwise, $d$ is inserted into a new cluster.

The algorithm reported in Figure 4.3 formalizes these notions. The algorithm receives as input a document $d$ to be clustered and a set of existing clusters $\mathcal{C}$. The preprocessing performed at line 1 allows us to neglect, during the computation of document resemblance, some minor syntactical details proper of the web environment. Currently, our preprocessing phase consists of the following steps. First, a new-line character is added at the end of each HTML tag of the input
document. Second, HTTP GET parameters are removed\(^1\). Third, to avoid case sensitivity, each line is converted to lowercase.

For our purpose, we define the multiset of tokens \( S(d) \) as the multiset of lines (i.e., sequences of characters terminated by a new-line character) extracted from document \( d \). As the preprocessing step inserts a new-line character after the end of each HTML tag, exceedingly long lines are broken (of course, provided that they contain at least one HTML tag).

To compute the resemblance index (line 7) the whole document it is not needed. We can store in each category just the fingerprint of an HTML page, thus reducing space requirements and avoiding the computational effort connected to recalculating the fingerprint of a document multiple times.

The computation of the resemblance between a document \( d \) and a cluster \( C \) is a rather tricky question, as it could drastically affect the results of the clusterization process. As a first approximation, we could define the resemblance between \( d \) and \( C \) as the resemblance between \( d \) and a representative member of \( C \). If we elected as representative the document with maximum resemblance with \( d \), then we would incorrectly group in the same cluster a whole set of documents whose pairwise similarity is greater than \( \tau \). For example, imagine a set of versions of a paper: consecutive versions are often “roughly the same”, but non-consecutive ones can be quite different; choosing as cluster representative the document with maximum resemblance would lead to group all revisions in the same cluster. Alternatively, we could act conservatively, by choosing as representative the document with minimum resemblance with \( d \). We experimentally observed that this solution often leads to the creation of many spurious clusters. A better approach would consists in defining the resemblance between \( d \) and \( C \) as the mean value of the resemblance between \( d \) and each member of \( C \). Unfortunately, to cluster a new document, all these solutions are \( O(w) \) in complexity, where \( w \) is the number of documents that have already been classified.

Our approach leverages the observation that the web pages hosted on fast-flux domains that belongs to the same scam botnet differ only for minor details, many of which have already been removed through the preprocessing phase discussed above. For these reasons, the algorithm in Figure 4.3 approximates the resemblance between \( d \) and \( C \) with the resemblance between \( d \) and the first member that has been inserted in \( C \) (line 6). This reduces the complexity for the clusterization of a new document to \( O(n) \), where \( n < w \) is the number of existing clusters. Although very simple, this method turned out to work incredibly well in practice.

\(^1\)The rationale is that very often these parameters consist in randomly generated session identifiers that could jeopardize the computation of the resemblance index.
Chapter 4. FluXOR

Figure 4.4: Typical deployment of the system. Multiple collectors and monitors can be used to distribute the workload and to uniformly blend the system in the victims.

4.6.3 Periodical re-clusterization

We observed that often fast-flux botnets change the hosted scam site. We suppose that malware authors rent their botnets to spammers or other miscreants for rather brief periods of time[66]. For this reason, existing clusters need to be periodically refreshed. Every day, for each existing cluster, we randomly choose a representative malicious FQDN and verify whether the scam homepage hosted on that domain has actually changed or not. To this end, we calculate the similarity index between the version of the scam homepage that has been previously used to cluster this malicious FQDN and a newly downloaded version of the page. If the similarity index is less that $\tau$, then we delete the whole cluster and recluster its members again.

4.7 Architecture and implementation of the system

The architecture of FluXOR includes four components and each one accomplishes a very specific task (see Figure 4.4): (i) one or more collectors of suspicious hostnames, (ii) one of more monitors of suspicious and malicious hostnames, (iii) a detector of fast-flux service networks, (iii) and a clusterization module.

FluXOR is entirely developed in Python and consist of more than 3000 LOC, without including the code of the web interface used to display the results of the analysis.
Chapter 4. FluXOR

4.7.1 Collector

The collector harvests from various sources hostnames that could be associated with fast-flux service networks. Examples of sources are unsolicited emails, instant messages and post in public web forums and blogs. The current implementation of FluXOR only supports harvesting of suspicious hostnames from emails. In the future this component will be extended to support other sources, for example using web crawlers and honeypots. Newly collected hostnames are flagged as suspicious and are considered as such and monitored until the detector classifies them.

4.7.2 Monitor

The monitor is responsible for monitoring suspicious and malicious hostnames. Benign FQDNs, instead, do not need to be monitored. The distinguishing features used by FluXOR to detect fast-flux service networks are extracted from data obtained by querying two different sources: (i) non-authoritative name servers and (ii) WHOIS servers. Once a malicious hostname is detected, instead, it is sufficient to perform a subset of the queries used to monitor suspicious hostnames, that is, those used to extract features describing the heterogeneity of the agents. For statistical and analysis purposes other information about the agents are also collected (e.g., the country in which the hosts are located and their geographical location). A description of the queries performed follows.

Features characterizing the domain name ($F_1$ and $F_2$). Given a FQDN like www.factvillage.com, the age of the domain and the registrar in charge for the domain are determined through WHOIS queries on the name of the second-level domain (e.g. factvillage.com). Although the query is conceptually trivial, it presents a serious challenge from the practical point of view. The WHOIS protocol does not define the format in which replies to queries have to be formatted and registries are free to choose the format they like more [24]. Moreover, some registration authorities omit to publish part of the information needed by our analysis. Today the entire IPV4 address space is assigned to 10 different registries. Things are more and more complicated for top-level domains because each domain is assigned to a different registry. Currently we are using a custom WHOIS client that is able to parse the format used by the most common registration authorities. To deal with the registries not currently supported by our client, we rely on a commercial service, that extracts WHOIS information and convert them in XML and offers a free limited number of queries per day. In the future we will extend our client to make the system completely independent from third parties.
Features characterizing the degree of availability of the network (F₃ and F₄). The natural approach to enumerate all the resource records of type “A” associated with a particular FQDN (i.e., the IP addresses of the potential fast-flux agents) and the time-to-live of the various records would be to query directly the authoritative name server for the suspicious domain. Although at each query we would always obtain “fresh” records and we would have the highest chance to see previously unseen records (i.e., in the ideal case records are rotated at each query and always have the highest time-to-live), the malicious authoritative name server could easily correlate the high number of queries with a system like FluXOR and consequently fool the analysis by returning fake resource records. The solution currently adopted by FluXOR is to collect the information by issuing recursive queries through multiple public non-authoritative name servers, such that FluXOR queries are blended in the victims’ queries. To estimate the maximum time-to-live of the resource records, to maximize the number of agents seen, and to minimize the network traffic, non-authoritative name servers are queried immediately after the cached records have expired.

Features characterizing the heterogeneity of the agents (F₅ to F₉). The remaining features are specific to the IP addresses into which a suspicious FQDN is resolved to. The number of distinct networks (F₅) associated with the same FQDN is computed by enumerating the distinct networks associated with the IP addresses of the potential fast-flux agents. This information can be obtained through a WHOIS query, one for each IP address, directed the respective registry. Similarly, the number of distinct autonomous systems associated with the same FQDN (F₆), is obtained by querying the databases of the regional registries for the AS to which each IP address belongs to. The number of distinct domain names associated with the IP addresses of the potential fast-flux agents (F₇) are obtained by querying name servers for pointer (PTR) resource records associated with each IP address (this kind of query is commonly known as “reverse lookup”). The hostnames obtained are subsequently split to extract the domain name. The network name and the organization owning the network (F₈ and F₉) are obtained through WHOIS queries. Unfortunately some of the information from which we extract the features of interest are not always available. An example are PTR records associated with the IP addresses of the potential agents.

4.7.3 Detector

The detector of malicious hostnames feed the set of collected features of the suspicious hostname to the naïve Bayesian classifier for the classification. The classifier is built on top of Weka [100], using the classification algorithm called
“NaiveBayesSimple”, which models numeric attributes by a normal distribution. Given the features of a suspicious hostname the classifier returns the class (i.e., benign or malicious) to which the hostname is most likely to belong to. The classifier was trained with a set of malicious and benign FQDNs that we manually classified, with the help of data obtained after a week of monitoring. Furthermore, the set of benign FQDNs used for training were extended, to make it more heterogeneous, by adding the address of some randomly selected websites we recently visited. The assumption that the features are completely independent, made by this type of classifier, might appear to simplistic. Nevertheless, this approach turned out to have very good performance in many real-world situations and the work of Zhang has shown that the efficacy of naïve Bayesian classifiers has some theoretical foundations [107]. In our context, as discussed later in Section 7.5, this approach gives very accurate results.

4.7.4 Clusterizer

The clusterization module periodically retrieves from FluXOR’s database the existing clusters and the malicious FQDNs that have not been clustered yet. Then, for each malicious FQDN that has to be clustered, this component is responsible to download the scam homepage, apply the clusterization algorithm and finally store the results back into FluXOR’s database.

4.8 Experimental results

We started running FluXOR in the beginning of January 2008, but we completed the integration of the clusterization module only at the end of February 2008. At the beginning the monitor and the detector were located on the same machine, an AMD Athlon XP 1.8GHz with 384Mb of RAM, running GNU/Linux and using MySQL for the persistent storage. After the presentation of our research results in Paris, Cisco IronPort, a leader industry involved in stopping emerging Internet threats, gave us powerful new hardware letting us continue our research.

The detector has initially been trained with three different data-sets, containing features extracted after one, two, and three hours of monitoring respectively. The three training sets were composed by 50 benign and 75 malicious FQDNs manually analyzed and classified. The collector was located on the mail server of our laboratory and processed all the spam emails forwarded by the mail server of our department. Malicious FQDNs were all extracted from spam emails, while benign hostnames were extracted from emails (both spam and non-spam) and from the history of our browsers.
<table>
<thead>
<tr>
<th>Description</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processed spam email messages</td>
<td>44804</td>
</tr>
<tr>
<td>Extracted URLs</td>
<td>15281</td>
</tr>
<tr>
<td>Active FQDNs (whose hostname could be resolved)</td>
<td>4961</td>
</tr>
<tr>
<td>Fast-flux service networks</td>
<td>387</td>
</tr>
<tr>
<td>Fast-flux agents</td>
<td>31998</td>
</tr>
</tbody>
</table>

Table 4.4: Summary of preliminary results (first 3 months) obtained using FluXOR to monitor the suspicious hostnames found in spam emails. Note that the number of agents is the number of distinct IP addresses. Dial-up hosts using dynamically assigned addresses might use multiple addresses and multiple hosts might share some addresses.

Table 4.4 summarizes the most important numbers of our first month experiments at the time of FluXOR presentation in Paris: the volume of spam email messages processed, the number of URLs extracted, the number of FQDNs active at the time the emails were received, the number of fast-flux service networks detected and the number of distinct fast-flux agents. About 7.8% of the active FQDNs turned out to conceal fast-flux service networks served by 31998 distinct fast-flux agents.

We evaluated the detection accuracy automatically, before training the classifier, and manually by comparing the output of the detector with our belief. Although during the manual analysis we found some corner case benign and malicious hostnames, the detector always classified the suspicious hostnames correctly: no false-negatives have been found because we correctly classified every bad domain as bad and zero false-positives because no good domains were classified as bad. We also tried to correlate the data collected in the different ways to understand the underground economy behind the botnet phenomenon.

### 4.8.1 Detection accuracy

We evaluated the accuracy of our detection strategy following two different strategies: (i) an automatic cross-validation with the three training data-sets and (ii) a manual analysis of a random subset of the active FQDNs extracted from the emails.

Part of our training data-set was used to estimate the accuracy of the model using cross-validation, with 5 and 10 folds [52]. No hostname was misclassified. The manual analysis was performed by comparing the response of the detector with our belief about the maliciousness of the hostnames. Hostnames whose maliciousness was difficult to attest were monitored for a day. The detector was
invoked three times on each sample, the first time with the features extracted after one hour of monitoring and the corresponding model, the second and the third time with the features extracted after two and after three hours of monitoring, and the corresponding model, respectively. Note that the amount of active hostnames processed were rather large and impossible to analyze manually in its entirety. Thus, we pruned the set using a filter to identify all the hostnames that were undoubtedly benign (i.e., those, after three hours of monitoring, associated with only two or less IP addresses and classified as benign). The manual analysis confirmed the correctness of our classifier, no hostname was misclassified.

During the manual analysis of the accuracy of the detector we came across some peculiar benign hostnames that had some of the characteristic of malicious hostnames. Two examples of these hostnames are imageshack.us and database.clamav.net. These hostnames are associated with very small time-to-live and are resolved in multiple IP addresses, 129 and 21, respectively. All the 129 distinct IP addresses associated with the first hostname belong to the same network. That makes us believe that the hosts are hosted in a server farm somewhere and that load-balancing is implemented using DNS round-robin. On the other hand, the IP addresses associated with database.clamav.net (see the discussion in Section 7.3) are located in 12 distinct networks, because mirrors are voluntarily hosted by companies and universities. Both hostnames belong to domains registered several years ago through registrars that are not commonly used by miscreants. In both cases FluXOR correctly classified the hostnames, even when the detection was performed using the features collected during one hour of monitoring only. Other examples of correctly classified benign hostnames that share some of the features of hostnames used for fast-flux service networks are pool.ntp.org and en.wikipedia.org.nyud.net (Wikipedia mirrored through Coral Content Distribution Network). It’s important to remind that nowadays some companies are specialized in answering the requests of a high demand of stability and availability of server infrastructure and offer to host a service in their content delivery network (CDN). These providers offer a large variety of services as for example reverse proxying an existing web server; all these services have in common that the load is distributed to several servers located in data centers throughout the world, whose IP addresses are distributed via DNS. Simply evaluating requests source IP address, load can be routed to the nearest unloaded server. Since there are many huge companies (like Microsoft, Apple, CNN, NBC, Adobe, Symantec...) that use these kind of CDN to host their web sites, and
some CDN have many features in common with FFSN, we noticed that some of other proposed methods to detect Fast-Flux domains can lead to false positives while FluXOR seems always able to correctly classify analyzed domains.

We also identified several very young (or not very active) fast-flux service networks for which, after an hour of monitoring, we only saw from three to five distinct agents. After three hours of monitoring the size of the network was still very small and reached only seven or eight agents. Despite the small number of agents, the hostnames were always classified as malicious, even when detection was performed using the data collected in an hour of monitoring. Not completely convinced of the response of the detector, we continued to monitor the hostnames. After several days the service networks encompassed hundred of hosts.

Three observations are worth mentioning. First, the detector is surprisingly precise. Second, in less than three hours we can precisely tell if a FQDN is malicious or not. Third, the current status of the fast-flux service network might not reflect the status of the network in the future (e.g., a hostname can be used for any kind of purpose at the beginning and then associated with a fast-flux service network in the future). The detector can only classify the current status of the hostname and, in order to detect a change of the status, the hostname must be monitored and classified again.

4.8.2 Empirical analysis of the fast-flux service networks phenomenon

As beforehand said in the first month we collected suspicious hostnames from a single source only and the number of hostnames collected was rather small, but the number of detected fast-flux service networks and the number of their agents was unexpectedly very large. We kept on analyzing new domains over the time and in the following paragraphs we briefly summarize some results we believe are interesting. Real-time and complete results of the analysis can be found on-line at http://fluxor.laser.dico.unimi.it.

Figure 4.5 shows the number of fast-flux agents, belonging to six distinct networks, detected during our preliminary analysis. The number of agents detected depends on many factors. For example the time-to-live of the DNS resource records, the number of records returned at each query, and the frequency at which the set of active agents is updated. The case of ibank-halifax.com is very impressive. In less than a day we detected about 3000 agents. The turnaround of agents in the average fast-flux service network is much smaller. By our preliminary analysis the average number of new agents detected daily was about 122.

We visited some of the websites served thorough the detected fast-flux ser-
Figure 4.5: Number of fast-flux agents, serving some representative fast-flux service networks, detected during the time.

vice networks and found out that several FQDNs were associated with the same website. The networks were probably pointing to the same mother-ship. Our hypothesis was that, to improve the availability of the system, miscreants registered multiple domains. If a domain was shutdown, victims could still be served through the other domains. Thus, it is more difficult for the authorities to eradicate the scam. Besides the common website, this hypothesis was further corroborated by the fact that multiple fast-flux service networks were served by the same set of agents. Figure 4.5 shows that the number of agents detected during the time for the FQDNs wherefell.com and cheaptmundo.com grows symmetrically. We also observed that the two domains shared the same authoritative name servers and also about 81% of the agents. We believe it is reasonable to assume that all the fast-flux networks pointing to the same website, and thus used for the same fraud, are served by agents belonging to the same scam botnet.

Table 4.5 summarizes some of the most important numbers of our subsequent experiments: the volume of spam email messages processed, the number of URLs extracted, the number of FQDNs active at the time the emails were received, the number of fast-flux service networks detected, the number of distinct fast-flux agents, and the number of fast-flux botnets resulting from our clusterization strategy. Note that the number of fast-flux agents we report is the number of distinct IP addresses we observed. Dial-up hosts using dynamically assigned
Table 4.5: Summary of the results obtained using FluXOR for three months.

<table>
<thead>
<tr>
<th>Description</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processed email messages</td>
<td>144952</td>
</tr>
<tr>
<td>Extracted URLs</td>
<td>34466</td>
</tr>
<tr>
<td>Active FQDNs (whose hostname can be resolved)</td>
<td>26236</td>
</tr>
<tr>
<td>Fast-flux service networks</td>
<td>9988</td>
</tr>
<tr>
<td>Fast-flux agents</td>
<td>162855</td>
</tr>
<tr>
<td>Fast-flux botnets</td>
<td>25</td>
</tr>
</tbody>
</table>

addresses might use multiple addresses and multiple hosts might share some addresses. The presence of agents with dynamic IP addresses makes the problem of counting the number of botnet members significantly harder [104, 108]. Currently, we do not handle this issue, but we plan to address it in future work.

Figure 4.6a shows the cumulative distribution of fast-flux service networks. Even if malicious FQDNs get continuously closed by authorities, the number of active fast-flux service networks seems to grow almost linearly: every day, we discover on average about 147 new fast-flux service networks. In total we saw more than 30 different Top Level Domains (TLDs) used in fast-flux service networks and in Table 4.6 we report the top 10 TLDs with the cumulative number of malicious domain names tracked by Fluxor in the first three months of experiments. We must notice that as top registrars become aware of Fast-Flux technique and increasingly take measures to fight its use, as criminals buy new domains from more, and often almost unheard, global top level domains (TLDs) than ever.

There is also another factor to consider. For a long time criminals behind fast-flux and classical scam sites have been able to take advantage of a time limit policy named Add Grace Period which allows them registering and then deleting a new domain at no cost, as long as the cancellation was within the first five days. This practice is commonly known as domain tasting. Repeated use of domain tasting results in domain kiting where domains remain registered for considerably longer and without ever being paid for.

In order to fight domain tasting and domain kiting, in June 2008, the Internet Corporation for Assigned Names and Numbers (ICANN) implemented some measures to address these kind of problems. Domain registrars could de-register only ten percent of their total registrations for free, and US $0.20 would be levied for each domain cancellation over and above the ten percent limit, increasing this penalty to US $6.75 as of July 2009. After these changes, in August 2009, ICANN reported a 99.7% decrease in such deletions between June % 2008 and April 2009.

On the other side, the growth of the cumulative distribution of malicious agents (Figure 4.6b) seems to be less uniform than the previous graph, but the
Figure 4.6: Cumulative distribution of (a) fast-flux service networks and (b) malicious agents.
rate of newly discovered malicious agents is impressive: the average number of new agents discovered daily is 2395.

In Figure 4.8 is shown the geographical distribution of the detected agents. Their heterogeneous geographical distribution, that ranges from US, Russia, China and Europe testifies that fast flux service networks are truly distributed and the scale of the problem is world-wide.

In Figure 4.7 are reported the number of malicious and benign domains collected during about 2 years of analysis. Unfortunately, for hardware problems, FluXOR didn’t run constantly during these two years, but it’s interesting pointing out that we have been likewise able to notice the most famous fast-flux botnet instance. For example, from the graph, we can identify at the end of December 2008 the increase of malicious domains related to the famous Conficker. Conficker, also known as Downup, Downadup and Kido, is a computer worm that targeted the Microsoft Windows OS, and in one of its variants was able to download a previously identified email-worm called Waledac with password stealing capabilities, still in the wild at the time of writing. According to data from the Shadowserver Foundation [87] and the Conficker Working Group [18], on 29 December 2009, IP addresses showing signs of Conficker infections peaked at 6.5 million.

4.8.3 Correlations among different fast-flux service networks

Fast-flux botnets. Globally, the clusterization module identified 25 clusters, that correspond to as many different fast-flux botnets. To validate these results, we manually inspected the scam sites hosted by some randomly chosen malicious
Figure 4.7: FluXOR results: malicious and benign domains analyzed in our experiments.

Figure 4.8: Geographical distributions of the detected fast-flux agents.
Chapter 4. FluXOR

Figure 4.9: One of the most famous fake pharmacy scam site using fast-flux technique.

FQDNs of each botnet. The manual inspection confirmed that the scam sites appeared very similar and they were actually advertising the same products, thus they can be reasonably considered as closely-related.

Table 4.7a reports the biggest fast-flux service networks clusters, obtained through the application of the approach discussed in Section 4.6.2. For each fast-flux botnet, we report its name (manually extracted from one of the scam sites of the botnet), and the number of different fast-flux service networks (i.e., the number of malicious FQDNs in that cluster). It is worth noting that roughly 40% of the fast-flux service networks detected by FluXOR belongs to the “European Pharmacy” botnet, thus granting to this botnet a very high availability. A picture of the infamous “European Pharmacy” site is reported in Figure 4.9.

It is interesting that, despite the large number of malicious FQDNs detected by FluXOR, these domains belongs to a rather limited number of botnets. This is a clear evidence that, even if the fast-flux phenomenon is still widespread, it can probably be ascribed to a limited number of miscreants.

Fast-flux botnets size. Table 4.7b shows fast-flux botnets that include the largest number of malicious agents. Except for “European Pharmacy”, it is interesting to note that these botnets involve a rather limited number of distinct malicious FQDNs. This makes such botnets apparently easy to shutdown. Despite that, many of them are rather long-living. As an example, “Halifax Online
Table 4.7: Fast-flux botnets (a) and fast-flux botnets with the largest number of agents (b). These results have been obtained by tuning the algorithm in Figure 4.3 with a similarity threshold $\tau = 0.8$, as we empirically observed that such a value leads to very good results.
Banking” is the second largest fast-flux botnet we have seen during our experiments; even if it includes just a single malicious FQDN (ibank-halifax.com), this botnet has been active from February, 11th 2008 until March, 20th 2008. This is a clear symptom of the problems involved with taking legal actions against domains registered in countries with a lax cyber-crime legislation.

**Fast-flux agents dispersion.** Table 4.8 shows the dispersion of malicious agents among different fast-flux botnets. About 11609 (11.04%) hosts are included in more than 5 different botnets (calculated only from the set of active fast-flux service networks at the time of writing). Globally, 49463 (47.02%) malicious agents are spread over more than one botnet. We believe this is connected to several factors. First, the dispersion of the same agent across different botnets could be due to the fact that different malware samples spread themselves using the same vulnerabilities. Furthermore, this can probably also be a symptom of the existence of tight relationships between malware authors. Finally, it is also worth considering that the same IP address could appear in multiple botnets because different malicious agents with dial-up Internet connections dynamically acquire the same address during the time.

**4.8.4 Correlation between fast-flux botnets and spam**

In our experiments, we tried to infer the existing relationships between spam and fast-flux botnets, in order to gather detailed statistics about the volume of spam related with each botnet. Tracking specific spam back to a particular botnet is somewhat challenging, but, by putting together in the right way all the information collected, it can easily be accomplished. For this purpose, we modified FluXOR’s collectors to keep track of how many times a suspicious FQDN has appeared in a spam email message. Subsequently, when a new malicious hostname is clustered, the number of emails connected to that cluster is increased by the emails containing the newly classified domain name.
Since mid of January 2008, during the subsequent three months, we analyzed 144952 spam email messages, from which we extracted 29393 distinct suspicious FQDNs (26236 active during clusterization), and 9988 of them turned out to be malicious. Table 4.9 reports some statistics about fast-flux botnets we estimate to have generated the greater volume of spam traffic. It is quite impressive than more than 18% of spam messages received by the mail server of our department were actually connected with the fast-flux phenomenon.

From our results, “European Pharmacy” is also the most active botnet in the spam-flow: the concept of utilizing a pharmacy site as a scam is not new, as there are plenty of them around that all resolve into the same financial endpoints, but by changing the content of spam messages roughly every few minutes, this botnet is able to evade many content-based spam filters.

<table>
<thead>
<tr>
<th>Botnet name</th>
<th># spam emails</th>
<th>% of all spam</th>
</tr>
</thead>
<tbody>
<tr>
<td>European Pharmacy</td>
<td>12056</td>
<td>8.32%</td>
</tr>
<tr>
<td>SwissWatchesDirect</td>
<td>3330</td>
<td>2.30%</td>
</tr>
<tr>
<td>RXNET</td>
<td>2558</td>
<td>1.76%</td>
</tr>
<tr>
<td>MaxHerbal</td>
<td>1897</td>
<td>1.31%</td>
</tr>
<tr>
<td>Other FFSNs</td>
<td>6395</td>
<td>4.41%</td>
</tr>
<tr>
<td>Total</td>
<td>144952</td>
<td>18.10%</td>
</tr>
</tbody>
</table>

Table 4.9: Relation between botnets and spam.

<table>
<thead>
<tr>
<th>Botnet name</th>
<th>Domain</th>
<th>Obs</th>
<th>Eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>European Pharmacy</td>
<td>mhvzd.maincoat.com</td>
<td>23</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>bvnct.inthe.com</td>
<td>20</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>whole botnet</td>
<td>68</td>
<td>248</td>
</tr>
<tr>
<td>Best Adult Sites</td>
<td><a href="http://www.truesexfilms.cn">www.truesexfilms.cn</a></td>
<td>13</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.videohome.cn">www.videohome.cn</a></td>
<td>18</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>whole botnet</td>
<td>52</td>
<td>95</td>
</tr>
<tr>
<td>SwissWatchesDirect</td>
<td>rgd.wdchange.com</td>
<td>13</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>rsy.dlineven.com</td>
<td>9</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>whole botnet</td>
<td>40</td>
<td>63</td>
</tr>
</tbody>
</table>

Table 4.10: Comparison of the lifetime of malicious FQDNs and fast-flux botnets. Column “Obs” reports the observed lifetime, while column “Eff” shows the effective lifetime, both in days.
4.8.5 Fast-flux domains lifetime

We define the observed lifetime of a fast-flux service network as the time interval between the first and the last time we have seen the domain associated with the service network (i.e., we performed a successful DNS query for the FQDN). We also define the effective lifetime of the service network as the time interval between the creation of the domain and the last time the domain was seen active. Similarly, we define the observed lifetime of a fast-flux botnet as the time interval between the first and last time we have seen one of the service networks of the botnet. The effective lifetime is instead the time interval between the creation of the oldest service network in the botnet and the last time we have seen any of its service networks.

In Table 4.10 we report in days the observed and effective lifetime for three different fast-flux scam botnets; for each of them, we show the lifetimes of two randomly chosen representative domain names. These data significantly differ from other analyzes of the lifetime of domains used for traditional web scams (i.e., non fast-flux) [5, 108], for which the average lifetime is around a week. The average lifetime of fast-flux domains tends to be a bit longer. As the table clearly shows, the use of multiple fast-flux service networks drastically increases the lifetime of the scam websites. Indeed, the majority of the scams we have been tracing since mid January are, three months later, still online.

These results significantly differ from other analyzes of web sites hosting scams lifetimes [5, 108], that place the average lifetime of traditional (i.e., non fast-flux) these sites around a week. Furthermore, as a fast-flux botnet often includes several hundreds of malicious domain, it is easy to see the tremendous impact of fast-flux techniques on the overall lifetime of botnets. On the other hand, maybe some of the domains we observed were not born as scam sites, so the effective lifetime may be actually greater than the effective malicious lifetime.

4.9 Mitigation

The information gathered through FluXOR can be used to actively defend benign hosts against the fast-flux threat. The list of malicious domains and IP addresses can be used in dns servers as URI DNSBL (DNS-based Blackhole List also known as Real-time Blackhole List or RBL), a list of domain names found in the “clickable” links contained in the body of spams, but generally not found inside legitimate messages, most often used to publish the addresses of computers or networks linked to malicious activity. With this strategy is possible to block the DNS resolution of all detected fast-flux domains.

Our analysis has also shown that these agents sometimes are active spammers.
We believe it would be very interesting and profitable to continuously correlate fast-flux agents and spammers. If such correlations existed, the data collected by our infrastructure could be used for spam prevention, to generate blacklists of active spammers. Such blacklists could extend the common blacklists used by the majority of mail servers to block incoming spam emails (e.g., spamhaus.org).

To this end, in one of our dns server, we configured a private version of DNSBL denying incoming connects from spam sources, where the contacting remote machine is identified by its IP address, which is checked against FluXOR database on the fly, and denying the resolution of fast-flux domains.

4.10 Discussion

Since January 2008 using FluXOR architecture we have been able to collect lot of data that provided us a rich picture of fast-flux botnets and their use in today’s Internet underground economy.

The apparent weakness of these service networks is that they can be blocked by shutting down the domain names registered for this purpose. Unfortunately, miscreants advertise their malicious web sites using multiple service networks simultaneously, such that the closure of some of the domain names does not affect the overall availability of their business. Furthermore we noticed that a large pool of domain names that are associated with fast-flux have been registered but, for some time before their active use in scam, remain dormant. Thus, the only effective way to eradicate this threat is to identify and clean all compromised machines serving the fast-flux service networks involved with the same scam and which are very likely to be part of the same botnet.

We notice also that a variety of web-based attacks are perpetrated using fast-flux techniques, including fraud and impersonation (often through SQL injection and Cross-Site-Scripting attacks) as well as malware infections.

Most of clusters we identified were used to increase the availability of scam or phishing sites. Criminals such as those of Rock Phish group are still using fast-flux techniques to protect and strengthen their network of compromised hosts. The number of today up-and-running Canadian or European Pharmacy scam sites are the proof of the fully functional method adopted by these miscreants, but also that proposed methods are not enough to fight the problem or suffers from problems related to scaling up the system or to the possible presence of false positive.

As a final remark it should be noted that starting from the mid of August 2009 we observed a significant decrease in the number of newly registered fast-flux domains, probably related to the measures implemented by the ICANN to
fight domain tasting and domain kiting or to the work law enforcement agencies are doing, trying to locate the criminal elements in cyberspace.

In the next Chapter we will focus on one of the most used botnet spreading vector, namely web applications attacks, proposing a mitigation framework. Our technique could represents a promising approach tring to automatically mitigate web injection attacks, also on legacy systems.
Increasingly, web applications handle sensitive data and interface with critical back-end components, but are often written by poorly experienced programmers with low security skills. As in other kind of software applications, also in web applications the presence of vulnerabilities can become a security problem. The majority of vulnerabilities that affect web applications can be in fact ascribed to the lack of proper validation of user’s input, before it is used as argument of an output function. Several program analysis techniques were proposed to automatically spot these vulnerabilities. One particularly effective is dynamic taint analysis, but unfortunately, this approach introduces a significant run-time penalty that in the worst case degrades the application response time, leading the application to be unusable. For web-based applications, even a short-term performance problem can cause lost revenue when customers cannot place orders or they become frustrated with slow response.

In this chapter, we present a hybrid analysis framework that blends together the strengths of static and dynamic approaches for the detection of vulnerabilities in web applications: a static analysis, performed just once, is used to reduce the run-time overhead of the dynamic monitoring phase. We designed and implemented a tool, called Phan, that is able to statically analyze PHP bytecode searching for dangerous code statements; then, only these statements are monitored during the dynamic analysis phase. Presented technique represents a promising approach to automatically mitigate injection attacks on web application with low run-time penalty and motivate further work in this direction.
5.1 Introduction

A web application is an application developed by adopting the web paradigm. Computation is performed via a client-server model, where the client is a web browser, the server is a web server augmented by some extension modules which enable the execution of server side code, and the communication between client and server relies on the HTTP protocol.

Today web applications are employed in a wide variety of different contexts: common examples include web mails, forums, blogs, social networking websites, online stores, and so on. Typical web pages contain both text and HTML markup that is often generated on demand generated by the server and interpreted by the client browser. Servers that generate static pages have full control over how the client will interpret the pages sent by the server, in case of dynamic pages however, they do not know how their output will be interpreted by the client. The heart of this issue is that if untrusted content can be introduced into a dynamic page, neither the server nor the client has enough information to recognize that this has happened and take protective actions.

Unfortunately, even if the insecurity of these applications is a well-known problem, according to a recent analysis [92], roughly 60% of the software vulnerabilities annually reported are specific to web applications. These vulnerabilities, which are typically remotely exploitable in nature, constitute the fastest growing and largest segment of the overall vulnerability count. In 2008, remotely exploitable vulnerabilities represented 90.2% of all vulnerabilities, up from about 85% in 2005. In Figure 5.1 is reported the cumulative count of web application vulnerabilities over time. Indeed, all the antivirus industries show in their seasonal reports the web as a major vehicle for cybercriminals looking to infect computers around the world. Many well-known and reputable sites have fallen victim to these kinds of attacks, pointing-out the need for all organizations, both large and small, to properly defend their websites. Sounds silly but the majority of these vulnerabilities can be ascribed to the same root cause: the lack of proper validation of user’s input.

The most common web application attacks that exploit such vulnerabilities are cross-site-scripting (XSS [13]) and SQL injection (SQLI [38]). In the first case, an attacker supplies an input that contains malicious Javascript code, that is later sent to an unaware client without a proper sanitization. Because of an implicit trust relationship between servers and clients, the malicious Javascript code will be interpreted by the client’s browser, thus leading to possible privacy violations (e.g., cookies stealing). Similarly, in a SQLI attack user-supplied data is used for building a SQL query string that is sent to a DBMS, without being properly validated against the presence of special characters (e.g., quotes or other
These kind of web threats are becoming more pervasive and cybercriminals belonging to underground industry are progressively driven by higher profit. Thousands of malicious web pages are discovered on a daily basis as cybercriminals hijack or infect legitimate web sites with malicious code, or redirects unsuspecting users to third-party sites that contain malware.

In both of presented attacks, the root cause of the vulnerability is that the application programmer fails or completely ignores to correctly validate the user-supplied input.

To prevent these security problems, web languages offer native sanitization primitives that a developer can use to validate input data. For example, PHP provides the `mysql_real_escape_string()` function that can be used to escape SQL special characters inserted in a given string. However, in order to avoid introducing security vulnerabilities in their applications, programmers must be aware of these security problems and properly sanitize each user-supplied input before any possible use as the argument of an output function. Unfortunately, nowadays web applications are often written by developers with low programming and security skills, that sometimes ignore programming good practice. Moreover, most applications are produced by assembling scripts coming from different developers, and it is not always feasible to review all the code base. As web applications are getting more and more complex, it is becoming quite difficult to be able to assert any elementary property about their code.

Several solutions have been proposed that aim at finding automatically secu-
Chapter 5. Web attacks mitigation with Phan

rity vulnerabilities in web applications [21]. Existing solutions can typically be classified into static and dynamic approaches. Static analyzers [50, 49, 102, 45] consider the source code without actually executing it; their strength is that they can reason over all possible program paths, but they are often overly conservative, since they normally report properties weaker than the ones that actually hold in a specific execution. On the other side, dynamic approaches [79, 34] focus on an actual execution of the target application; they consider only a limited number of program paths (i.e., those that have been covered in the observed executions), but they can provide more accurate results. Unfortunately, dynamic tools introduce a significant run-time overhead in the application being analyzed.

In this chapter we present a system for analyzing web applications based on a hybrid approach. Our solution blends together the strengths of static and dynamic approaches [28]. It has been implemented in an experimental prototype code named Phan (PHP Hybrid Analyzer). Phan currently targets PHP applications, but can be easily extended to other environments. First, Phan statically analyzes the target application, searching for dangerous statements; afterwards, only those statements that have been found to be dangerous will be monitored dynamically, thus reducing typical run-time overhead. It is worth noting that Phan does not work on PHP source code, but directly at the bytecode level. In fact, PHP applications are first compiled into a low-level and poorly documented bytecode language that is then interpreted by Zend, the PHP underlying virtual machine. Even if dynamic approaches targeting Zend bytecode already exist [79, 34, 69], this is, to the best of our knowledge, the first time static techniques are directly applied to Zend bytecode.

The contributions of this chapter can be summarized as follows:

- We present a hybrid program analysis framework for detecting input-driven security vulnerabilities in web applications. Our solution relies on a static preprocessing technique to reduce the run-time overhead of the subsequent dynamic analyzes.

- We describe how the aforementioned analysis can be applied to PHP programs. The application of program analysis methods to an interpreted, object-oriented and dynamic-typed scripting language like PHP presents several interesting and far than easy problems.

- We describe how we instrumented the Zend Framework in order to implement a prototype of Phan, our experimental tool that performs static and dynamic analysis of PHP applications, at the Zend bytecode level.
Chapter 5. Web attacks mitigation with Phan

5.2 Hybrid analysis of web applications

The goal of our approach is to monitor the execution of a given web application and to intercept any exploitation attempt, while trying to minimize the run-time overhead introduced by the on-line analysis. Our idea is that a preliminary static analysis of the application can reduce the amount of code that needs to be monitored dynamically. In particular, we are interested in protecting the web application against injection attacks, i.e., those attacks that exploit the improper validation of user-supplied input data, before it is used as argument of an output function.

Our approach is made up of two distinct logic steps: (i) a static analysis of the application, that identifies dangerous statements and (ii) run-time monitoring of the identified statements.

Initially, we generate a static model of the whole application. The application code is translated into an intermediate form. The rationale behind the choice of our intermediate representation language was to reduce the number of instruction and expression classes, in order to ease the subsequent analyzes, while still being able to precisely capture the semantics of the application code. Then, for each program function, we build its control flow graph (CFG) [3]. Such CFGs are connected together, thus obtaining an interprocedural CFG (iCFG), that is analyzed to individuate all possible code paths from a user input source (e.g., \$_GET, \$_POST and \$_COOKIE arrays in PHP) to a sensitive sink. By sensitive sink we mean any function that could lead to security problems when executed over un-sanitized user-supplied data (e.g., mysql_query() and echo(), in PHP). The mysql_query() function sends a unique query to the currently active database on the server while echo(), that is not actually a function, it’s a language construct that outputs one or more strings given as parameters, it’s the typical way in which the HTML code of the page are echoed.

Finally, from all statements appearing in these paths, we extract only those that might affect the input arguments of a sensitive sink. We do this by calculating, for each sensitive sink, the backward slice [98] over its input arguments. All these statements are marked as “dangerous” and during the subsequent dynamic phase, only dangerous statements need to be monitored.

It should be noted that, currently, our analysis is not able to accurately handle all reference statements that contain arrays or array elements: when our analysis is not able to determine which array element has been used or defined, we overly consider all the array as used or defined in order to preserve the soundness of the taint propagation phase. This limitation applies also to well-known hard-to-analyze constructs such as variable variables (e.g., \$($variable) ) an to the whole alias analysis phase.
As reported, the resulting static model is overly conservative, and due to limited support for aliasing and class constructs our analysis could be unsound (i.e., it may generate false negatives): currently, we address these limitations by extending the number of program statements to be monitored dynamically. In other words, we try to preserve soundness at the cost of greater run-time overhead.

Dangerous statements will be used to perform an efficient dynamic taint analysis, since most of the statements have been filtered out by the static preprocessing. Data that originate or derive from an untrusted source are marked as tainted: we start by marking input data as tainted, and then we dynamically keep track of how the tainted attribute propagates to other data. Initially, only a minimal set of program variables are considered to be tainted (e.g., all PHP global arrays that contain user-supplied data). As the execution goes on, other program variables become tainted. When we detect that tainted data containing malicious characters has reached a sensitive sink, we can choose either to block the execution or to sanitize tainted data before allowing the application to continue.

It is worth pointing out that the tainted values that we are able to detect cannot be sanitized as soon as they are read from input sources; in fact, at this point, we are not sure whether untrusted variables will eventually be sanitized by the application, nor if they will ever reach a sensitive sink. For these reasons, the preemptive sanitization of tainted variables could alter the original semantics of the target application.

On-line monitoring can be very effective, but, in order to minimize the run-time penalty, it should be constrained to a limited number of dangerous statements. However, the identification of these code paths requires a priori knowledge of the structure of the program, that, without the initial off-line static analysis phase, would be almost incomplete.

In Figure 5.2 we report a simple PHP script\(^1\) that checks whether the user has supplied a product ID as a GET parameter of the HTTP request (line 7); in this case, a SQL query is built to extract the information about the specified product from the underlying MySQL database (line 2). This script contains a SQLI vulnerability: since the user input is not properly sanitized, an attacker could manipulate the query submitted to the DBMS (line 4). This example will be used in the following to illustrate our approach.

\(^1\)Phan deals with Zend bytecode, however, for the sake of clarity, the example is shown in its source code form.
function get_product($id) {
    $q = "SELECT ... WHERE id=$id";
    mysql_connect(...);
    $res = mysql_query($q);
    }

if(isset($_GET['product_id'])) {
    $a = $_GET['product_id'];
    get_product($a);
} else {
    $msg = 'Invalid request';
    echo $msg;
}

Figure 5.2: Sample PHP code fragment with a SQLI vulnerability.

5.3 Phan: a hybrid analyzer for PHP applications

In this section we describe how the high-level solution introduced in Section 7.3 can be applied to PHP code. To this end, we present Phan, our hybrid analysis framework for PHP applications. Phan does not require any modification to the source code of the target application, nor any interaction with web application developers. All the static and dynamic analyses performed by Phan are carried out directly on Zend bytecode. We adopted this strategy in order to avoid the intricacies of parsing PHP: the bytecode has \( \sim 150 \) opcodes, and it is pretty stable among PHP releases.

Phan is organized into the two following main components:

1. an off-line analysis engine, that translates Zend bytecode into an intermediate form, constructs a control flow graph for each program function, merges together the CFGs into a single interprocedural CFG, and finally identifies dangerous program statements;

2. an on-line monitoring engine, that performs dynamic taint tracking on Zend bytecode, and reacts properly when tainted user-supplied data reaches a sensitive sink.

Each of these components is described into more details in the following sections.
5.3.1 Off-line analysis

The goal of this phase is to provide a conservative view of the whole application, that will be used to drive the on-line analysis to a limited number of program statements. It is worth pointing out that the off-line analysis has to be done just once for each application script, and has not to be repeated every time a user requires the execution of a PHP script. In this section, we describe the steps involved in the off-line analysis of a single application script. The same steps have to be carried out for each script in the target application.

**Translation into intermediate representation.** First of all we had to instrument Zend, in order to intercept the compilation of PHP scripts. In this way, we are able to obtain a bytecode representation of each application script. To ease the analysis, bytecode instructions are translated into a simple intermediate representation (IR). Our IR resembles a RISC-like assembly language, including just 5 instruction types (Assignment, Call, Ret, Jump and Nop) and 4 expression types (Constant, Variable, Array, and CompoundExpression; the last one is used to model unary, binary and ternary expressions). For each intermediate instruction, we also compute the set of used and defined variables [2]. The translation of Zend bytecode into IR language has required a significant engineering effort: each opcode supported by the Zend virtual machine has to be precisely modeled using our RISC-like language, in order to capture the exact semantics of the application being analyzed.

Today most of web applications, coded in scripting languages such as PHP, organize their code into different source files; only at run-time needed files are included following given file inclusion rules.

In other languages, as for example C, include statements only contain static file names (static literals) and thus, the consolidation can be easily done. In PHP, however, include statements can be composed of arbitrary expressions requiring more sophisticated resolution techniques than the application of a simple preprocessor such as the one used for C programs: this solution applied to PHP scripts would leave a significant number of includes unresolved leading all subsequent analysis to false positive. In PHP scripts indeed, names of files to be included can be constructed at run-time, can be built by recursion; including function can return values and also conditional inclusions are permitted. Therefore, to resolve this kind of include statement, information about the value of these expressions should be computed. These include resolutions, essential for the strength of subsequent analysis phases, could be automatically performed by literal analysis [49, 103].

Currently our framework supports only file inclusions performed through
static file names (static literals): if an application script includes additional modules through static names, each of them is recursively compiled and translated into our IR language in an automated way. Dynamic inclusions or run-time file name contructions must be manually resolved. In this way, we aim to obtain a complete and self-contained (except for PHP native functions) view of the PHP script.

In order to enhance the whole analysis and resolve nasty inclusions an iterative two-phase algorithm for fast and precise file inclusion resolution could be integrated in our framework as described by Jovanovic et al. in [49].

The example in Figure 5.2, when compiled by Zend, includes 24 opcodes, and is then translated into 33 intermediate instructions.

**CFG construction.** We briefly recall that a control flow graph (CFG) is a directed graph \( G = (B, E) \), where \( B \) is a set of nodes and \( E \subseteq B \times B \) is a set of edges [3]. In our context, CFG nodes represent basic blocks, i.e., sequences of intermediate instructions with a single entry point and a single exit point. Each graph edge \((b_i, b_j) \in E\) indicates that the execution can flow from basic block \( b_i \) to \( b_j \); we say that \( b_j \) is a successor of \( b_i \), and \( b_i \) is a predecessor of \( b_j \).

Let \( S = \{P_1, P_2, \ldots, P_n\} \) be a PHP application script, where each \( P_i \) is a program procedure, and \( P_1 \) is the "main" procedure of \( S \) (i.e., \( P_1 \) is the first code sequence that gets executed when \( S \) is invoked). We build the CFG of each procedure \( P_i \in S \) using standard techniques [2]. We inspect each CFG searching for indirect control transfer instructions. Indirect control transfers are handled using constraint propagation and reaching definition analysis [70]. We have now a set of CFGs \( C = \{C_{P_1}, C_{P_2}, \ldots, C_{P_n}\} \), where \( C_{P_i} \) is the control flow graph for program procedure \( P_i \). Then, in order to generate an interprocedural CFG (iCFG) of \( S \), we combine together the CFGs in \( C \) in the following way. For each instruction \( i \in S \), if \( i \) is a call to a user-defined function \( P_i \), let \( b_i \) be the basic block \( i \) belongs to, and let \( b_j \) be the successor of \( b_i \); then, we remove from the iCFG the control flow edge \((b_i, b_j)\) and we add two edges \((b_i, b_{entry}), (b_{exit}, b_j)\), where \( b_{entry} \) and \( b_{exit} \) are the entry and exit points of \( C_{P_i} \), respectively. Similarly, if \( i \) is an inclusion statement (i.e., ```include()```, ```include_once()```, ```require()```, or ```require_once()``` ) that includes the PHP script \( S' \), then we replace the control flow edge \((b_i, b_j)\) with two edges \((b_i, b'_{entry}), (b'_{exit}, b_j)\), where \( b'_{entry} \) is the entry point of \( C_{P_i} \) and \( b'_{exit} \) is its exit point.

In Figure 5.3 we show the interprocedural CFG of the example described in Figure 5.2. To build the iCFG, we merged together the CFG of the "main" procedure with the CFG of ```get_product()``` : the basic block containing the function call to ```get_product()``` is connected with the entry point of the called procedure.
Figure 5.3: Interprocedural control flow graph for the PHP code fragment presented in Figure 5.2.
Identification of dangerous statements. The off-line analysis terminates with the identification of dangerous code statements. We analyze the iCFG and we identify all input sources and sensitive sinks. Input sources correspond to those PHP superglobal variables\(^2\) that allow an application developer to read user-supplied data (e.g., \$_GET, \$_POST, \$_COOKIE and \$_REQUEST). In order to prevent second-order injection attacks, we should also consider the output arguments of functions that read data from a database or the filesystem as sensitive data sources. However, not all data coming from these sources was originally supplied by the user. For this reason, we excluded this feature from our current implementation, as we have not investigated yet how to handle the false positives that could arise from this design choice.

Sensitive sinks correspond to those functions that might send malicious data back to the user (e.g., \texttt{echo}() and \texttt{print}()) or to the underlying DBMS (e.g., \texttt{mysql_query}()). Through the application of standard data-flow analyses on the iCFG, we are able to ignore those sink function calls that are guaranteed to receive as input only constant arguments. Then, we use a graph traversal algorithm to identify all possible code paths from an input source to a sensitive sink. Dangerous statements are identified by extracting from these paths those statements that might affect an input argument of a sensitive sink. Let \(i\) denote a sensitive sink, and let \(W\) be the set of program variables that represent the input arguments of \(i\). We identify dangerous statements by computing over the iCFG the backward slice for the slicing criterion \((i, W)\). We first compute the set \(H = \bigcup_{w \in W} \text{srd}(w, i)\), where \(\text{srd}(w, i)\) represents the static reaching definitions for variable \(w\) at program point \(i\). Then, as described in [43], we can reduce our problem to the computation of the set \(L\) of program statements that are reachable in the data dependence graph of the analyzed program, starting from a statement in \(H\). Dangerous statements are all those statements included in \(L\) that also appear in a code path that connects an input source with a sensitive sink.

In the example reported in Figure 5.2 the only input source is represented by the \$_GET array, used at lines 7 and 8. We have two different sensitive functions: \texttt{mysql_query}() (line 4) and \texttt{echo}() (line 12); however, constant propagation analysis reveals that the only input argument of \texttt{echo}() is a constant value, thus this function is not considered as a sensitive sink. In Figure 5.3 we depict with a solid border the basic blocks that appear in a code path that connects an input source with a sensitive sink. The backward slice for the slicing criterion \((4, \{\$q\})\) includes only source lines \(\{8, 9, 2, 4\}\); these are the dangerous statements whose corresponding Zend opcodes will be monitored in the on-line phase.

\(^2\)PHP superglobals are built-in global variables that are always available in all scopes.
5.3.2 On-line analysis

During the on-line analysis phase we perform a dynamic taint analysis on Zend bytecode. Initially, only the input sources are marked as tainted. We modified the Zend virtual machine to guarantee the correct propagation of taint information during program execution; only dangerous code statements are dynamically monitored. When a function corresponding to a sensitive sink is invoked over tainted malicious characters, we can choose either to abort the execution or to sanitize the input before allowing the function to continue.

Taint meta-information. We modified the Zend virtual machine to keep track of the taint meta-information connected to string variables. Zend associates to each variable $x$ a zval structure, updated during the execution to reflect the current value of $x$. We augmented the zval structure by including taint meta-information. In particular a list of (index, labels) pairs is associated to each string variable, where index denotes a specific string character, while labels is a bit vector that specifies which taint labels are associated to that element. Taint labels allow us to precisely track which input sources affect a tainted program variable. In our architecture, taint meta-information is protected from unauthorized modifications by the isolation provided by the Zend virtual machine: as long as an attacker cannot tamper the virtual machine, taint meta-information cannot be altered.

Propagation of taint meta-information. To ensure the correct propagation of taint meta-information, we had to modify the implementation of string-related functions inside the Zend virtual machine. We also instrumented Zend’s internal functions that manipulate zval operands, propagating taint information from source operands to destination.

Phan is able to perform fine-grained tracking of tainted meta-information: as taint propagation is performed with character-level granularity, we can precisely handle also program statements that directly manipulate strings as character arrays.

Detection of injection attacks. When program execution reaches a sensitive sink, we check whether the sink function is going to be invoked over tainted input arguments. To each sink function, we associate an “oracle” procedure that determines if a particular tainted string exploits the vulnerability associated to that specific sink. In order to detect exploitation attempts, our current implementation of the oracle functions leverages well-known attack techniques. As an example, the oracle associated to the mysql_query() procedure performs limited
syntactical analysis of the SQL query that is going to be sent to the database, searching for tainted characters in unsafe positions (i.e., we search for tainted characters that could alter the original semantics of the query statement).

5.4 Implementation details

We have implemented Phan in an experimental prototype that extends PHP 5.2.6. The off-line analysis module consists of \( \sim 6000 \) lines of Python code and \( \sim 1500 \) lines of C code for interfacing with the Zend virtual machine. The on-line engine consists of \( \sim 1000 \) lines of C code.

The off-line analyzer has been realized as a PHP extension module that hooks Zend’s compilation routine. After Zend has successfully compiled a source file, the extension sends its bytecode representation to the Python module, that translates it into the intermediate language and performs the analysis described in Section 5.3.1. The final outcome is the set of dangerous opcode statements that have to be monitored at run-time. Our current implementation is still not complete, as we currently supports 93 out of 150 Zend opcodes.

For performance reasons, the on-line analyzer is entirely written in C. We had to install a limited number of hooks inside the Zend virtual machine, but the majority of the taint propagation code is encapsulated into a self-contained module. By limiting the number of modifications to Zend’s source code, we tried to minimize the burden of work required for porting the on-line engine to different versions of PHP.

5.5 Experimental results

Evaluation. Table 5.1 presents some experiments we accomplished over a set of open-source PHP applications. For each application, we report the vulnerability type, a public reference to the vulnerability description, the total number of Zend opcodes (i.e., Zend bytecode statements) in the monitored application script, the total number of Zend opcodes that appear along code paths that connect sources to sinks, and the number of dangerous opcodes. In the last column, we report the percentage of dangerous opcode with respect to path opcodes. As path opcodes represent a lower bound to the number of opcodes monitored by a fully dynamic approach, this percentage is a good approximation of the performance gain coming from a hybrid analysis solution. Moreover, we believe the improvements sketched out in the previous paragraph might further decrease the number of dangerous opcodes, and thus reduce the run-time overhead of the dynamic phase.
Chapter 5. Web attacks mitigation with Phan

<table>
<thead>
<tr>
<th>Application</th>
<th>Type</th>
<th>Opcodes</th>
<th>Path opcodes</th>
<th>Dangerous opcodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clean CMS 1.5</td>
<td>SQLI</td>
<td>221</td>
<td>104</td>
<td>56 (53.85%)</td>
</tr>
<tr>
<td>Goople CMS 1.8.2</td>
<td>SQLI</td>
<td>62</td>
<td>58</td>
<td>17 (29.31%)</td>
</tr>
<tr>
<td>MyForum 1.3</td>
<td>SQLI</td>
<td>1102</td>
<td>651</td>
<td>141 (21.66%)</td>
</tr>
<tr>
<td>Pizzis CMS 1.5.1</td>
<td>SQLI</td>
<td>91</td>
<td>38</td>
<td>11 (28.95%)</td>
</tr>
<tr>
<td>W2B phpGreetCards</td>
<td>XSS</td>
<td>1078</td>
<td>814</td>
<td>221 (27.15%)</td>
</tr>
<tr>
<td>WordPress</td>
<td>XSS</td>
<td>612</td>
<td>26</td>
<td>10 (38.46%)</td>
</tr>
</tbody>
</table>

Table 5.1: Evaluation.

Here we report the list of CVE or Bugtraq identifiers for publicly known information security vulnerabilities tested during our experiments: CVE-2008-5290, Bugtraq ID 33135, Bugtraq ID 31926, Bugtraq ID 33173, Bugtraq ID 33001, CVE-2008-5278.

Limitations and future work. The current version of Phan has some limitations, that we briefly summarize in this paragraph together with possible directions for future work.

The off-line engine can be significantly improved by integrating a strong include statements resolver and a static taint analysis module [49], that could further reduce the number of false positives and the number of program statements to be monitored dynamically. Moreover, the current static analysis engine has limited support for aliasing and class constructs. In the current implementation, we address these limitations by dynamically monitoring all those code regions that we are not able to analyze statically. Finally, Phan assumes the output of a sanitization routine to be untainted, without even considering that the sanitization process implemented by the application developer could be incorrect or incomplete. When a programmer uses a custom sanitization routine, our tool does not report warnings or errors and does not report false positives results but only increase the number of program statements to be monitored dynamically. Currently a boring manual activity to find all the custom sanitization routines is required but, once those have been identified and examined, if safe, can be added to the list of known sanitization routines and our framework will no longer be overly conservative over them.

In order to automate the entire process, we could use the approach described in [6] to verify the correctness of the input sanitization process.
5.6 Discussion

Even though the vulnerabilities that lead to SQL injections and XSS are well understood, these kind of attacks continues to be a problem due to a lack of effective techniques for detecting and preventing them. Improved coding practices could prevent SQL Injections or XSS but in practice attackers continue to find new exploits to circumvent the input checks implemented by programmers. Moreover, in most of legacy systems, the human effort needed to find and reimplement all of the vulnerable sections of such systems often makes this approach impractical or too expensive in realistic settings.

In this chapter, we presented an approach for detecting injection vulnerabilities in web applications through hybrid analysis techniques. Our proposal blends together the strengths of static and dynamic approaches: the preliminary static analysis phase helps reducing the run-time overhead connected with dynamic monitoring. We described the design and implementation of Phan, a hybrid analyzer for PHP applications that works directly at the Zend bytecode level.

The experimental results presented indicate that the improvement with respect to a taint analysis entirely dynamic is significant.

Those results are also encouraging: our technique, that is completely transparent to the applications, was able to correctly identify all the attacks that we performed on the tested applications. We think that presented technique represents a promising approach to mitigate injection attacks also on legacy systems and motivate further work in this direction. Thus, we plan to further increase the accuracy of our analysis in order to evaluate our solution in extended examples.

Unfortunately sometimes malicious activities mitigation or detection could fail and post-infection remediation remains the only solution. In the next two Chapters we will focus on various aspects of remediation capabilities of current malware detectors.
6.1 Introduction

One of the biggest problems the Internet community has to face today is the widespread diffusion of malware, malicious programs written with the explicit intent to damage users and to use compromised systems for various types of frauds. The second half of 2007 witnessed a drastic increase (about 135%) of the number of threats related to malware [92]. This can be ascribed to a number of different root causes, but the main reason is probably the easy financial gain that malware authors obtain by selling their creations in the underground market [32]. Besides the rapid spread of malware, we are observing a parallel advance in the techniques for protecting end-users against malicious code. In order to face the growing complexity in the techniques employed by malware writers to evade detection, traditional signature-based anti-malware solutions are now being supported by behavioral, semantics-aware, approaches [16, 60], that mainstream commercial products are starting to include [72, 85, 76].

To defend against malicious programs, users typically rely on malware detectors, which try to detect and prevent threats before the system is damaged. Unfortunately, in many cases detection and prevention are not possible. Imagine for example a user that is not running a malware detector or a user that is running a malware detector but who gets infected before the appropriate detection signature is released. In such a situation, post-infection remediation remains the only solution to get rid of a malware and of the damages it may have caused to the system, other than reinstalling the entire system. However, the experience has taught us that sometimes automatic remediation procedures could cause more problems than they would solve [89, 40].

As any kind of software application, malware detectors require thorough test-
Users do not only need a stable application, but also a product capable of detecting threats with low false-negative and false-positive rates, and capable of remediating their system from a damage caused by a malicious program that was not detected in time. For these reasons, the testing and the evaluation of a malware detector require particular attentions, to the point that the leading industries and researchers in the field have recently defined common guidelines to test this particular class of software [4]. Although these guidelines describe what should be evaluated, they do not describe any precise methodology to do that.

In this chapter we address the problem of evaluating the remediation capabilities of a malware detector and we propose a fully automated testing methodology to evaluate this characteristic. The proposed methodology is dynamic. We run a malicious program in a victim system and we monitor the execution to detect what modifications are made to the environment. Subsequently, we trigger the remediation procedure of the tested malware detector to clean up the victim system. Finally, we analyze the state of the environment to verify which of the modifications previously caused by the malicious program have been successfully reverted. We have implemented the proposed methodology in a prototype and evaluated six of the most rated malware detectors on the marked. Our evaluation testifies the effectiveness of our tests and shows that the remediation procedures of the tested detectors suffers incompleteness. For example, we have empirically observed that only about 80% of the untrusted executables dropped by malicious programs on infected systems are properly removed by malware detectors.

To summarize, the chapter makes the following contributions:

- a fully automated testing methodology to evaluate the completeness of remediation procedures in commercial malware detectors;
- a prototype implementation of our testing methodology;
- an empirical evaluation of six malware detectors currently available on the market, with about 100 malware samples each.

### 6.2 The importance of remediation

To comprehend why remediation is a key issue in defeating malware, let us consider a sample malicious program. Figure 6.1 shows a fragment of an execution trace of the sample malware, reporting the most important modifications to the system performed by the application. The malicious program replicates itself into a new executable (c:\windows\pq.exe), creates a registry key to configure the system to start the new executable automatically at boot, and tampers the...
configuration of the resolver (writing into `c:\windows\system32\drivers\etc\hosts`) to hijack network traffic, directed to `www.google.com` and `www.citi.com`, to a malicious web site. Moreover, consider a user whose system gets infected by this malware and whose system, at the time of infection, was not properly protected (e.g., the infection took place before a signature for detecting the malware was released). Only after a while, when the appropriate signature becomes available, the malware detector will detect the presence of the malware on the system and remediate the damages.

What the user expects from the detector is that it is able to remediate completely the system. That is, the malware detector has to revert all the modifications made to the system by the malicious program. In the case of the example, that means that the original malicious executable (`malware.exe`), the executable created (`c:\windows\poq.exe`), and the registry key (`\HKLM\Software\Microsoft\Windows\CurrentVersion\Run\v`) have to be removed from the system. Similarly the process started has to be killed, and the malicious entries added to the configuration of the resolver removed (`c:\windows\system32\drivers\etc\hosts`).

If the remediation procedure is not complete, the system can be left in an unsafe state. Imagine for example that the malware detector reverts all the actions performed by the malicious program, but that it is not able to restore the proper configuration of the resolver (i.e., to remove the malicious entries added to the file `c:\windows\system32\drivers\etc\hosts`). Even though all the malicious executables dropped by the malware are removed from the system, the security of the user is still compromised because part of the network traffic is hijacked to a malicious web site. This site can be used to steal sensitive information or to deliver new malware to the user.

Of course one should claim that it is not really necessary to remove all effects
of a malware execution to make the system safe again. For example, when a malware adds some entries into the Windows registry, these entries might not matter once the malware process have been killed and the executable have been removed, but there are some well known counterexamples.

We can report the case of ImageFileExecutionOptions registry entry, that allows to perform a redirection whenever an executable with a specific name is launched: this is commonly used for running an application under the debugger. Imagine for example a malware sample that configure itself into the ImageFileExecutionOptions entry in order to be lauched as a debugger for an application $A$ or for the whole Windows GUI shell (explorer.exe). When these executables will be launched, the operating system will first launch the malware, as it were a debugger, and the path to application $A$ or to the explorer.exe as parameters. If the malware detector will erase the malware executable without deleting the related ImageFileExecutionOptions registry keys, the application $A$ or explorer.exe (the whole Windows system) won’t be able to start again leading in a crash.

### 6.3 Testing methodology

This section defines the ideal remediation procedure (Section 6.3.1) and presents the testing methodology we have developed to verify whether the remediation procedures available in a malware detector resemble the ideal one or not (Section 6.3.2).

#### 6.3.1 The ideal remediation procedure

For the purpose of defining the ideal remediation procedure, we can think the execution of a malicious program as characterized only by interactions with the environment, where each interaction corresponds to the invocation of a particular OS routine (or system call). Let $S = \langle s_0, s_1, \ldots, s_n \rangle$ be the execution trace of our malware sample. The system calls in $S$ can be classified in two classes: those that modify the state of the environment and those that do not. For example, to replicate itself into a system folder, a malicious program has to create a file and to copy its code into the file. Similarly, to install itself at boot, the program has to create a particular registry key. Both activities involve a modification of the state of the environment. On the other hand, a program that reads and parses the content of a file does not alter the state of the environment. For our purpose, it is sufficient to consider only a subset of all the system calls executed by the malicious program, including only the ones that modify the state of the
local environment: $S' = \{s_j \in S : s_j \text{ contributes to modify the state of the local system}\}$.

To achieve a particular high-level goal, the malicious program has to execute multiple system calls. As an example, to replicate itself, the program has to create a file and then to write its payload into the file (typically in multiple passes). Nevertheless, for remediating a system from a malware infection, it is not important to know which system calls the malicious program executed to modify the system, but instead what modifications were made to the local system by the program. For this reason, we can abstract the sequence of system calls $S'$ executed by the malicious program to infect the system through a set of high-level system state transitions $T$. Each transition $t \in T$ represents the effect on the local system produced by the execution of a sequence of related system calls. Let us consider again our sample malicious behavior of Figure 6.1 and the corresponding system call trace shown in Figure 7.3, where each high-level behavior is associated with the sequence of system calls executed by the malware and that produces a particular state transition. In the figure, irrelevant system calls (i.e., the system calls that do not modify the state of the system) are reported in gray. As an example, to create a file on the file system (which consists in a copy
of the malicious program) the following system calls are executed: \texttt{NtCreateFile}, \texttt{NtWriteFile}, and \texttt{NtClose}. The high-level state transition associated with this sequence of system calls is the creation of a new file on the system.

The set of high-level system state transitions $T$ can be divided in multiple classes, each of which represents a state transition involving a particular class of OS resource. For example, for a Microsoft Windows system we have $T = F \cup R \cup P \cup S$ where: $F$ represents the state transitions involving files, $R$ the state transitions involving registry keys, $P$ those involving processes, and $S$ those involving system services. This separation is important because each class of state transition requires a specific mechanism for remediation. It is worth pointing out that, in our context, we are interested only in the state transitions that modify the local system, as no remediation could be accomplished for transitions that affect remote hosts. Furthermore, we do not consider state transitions caused by other benign processes that might be running in the test environment.

A remediation procedure $\mathcal{P}$ is \textit{complete} if it is able to revert all the effects (i.e., the high-level state transitions) of the execution of the malware: $\forall t \in T, t$ is reverted by $\mathcal{P}$. The ideal remediation procedure is the one that is complete. Reverting a particular state-transition means to bring the state of the system back to that preceding the transition. Practically speaking, if a malicious program creates a file we expect the malware detector to remove the file; if the malicious program reconfigures the resolver, we expect the malware detector to adjust the configuration of the resolver.

### 6.3.2 Testing the completeness of a remediation procedure

**Testing scenarios.**

The following paragraphs present two real-world scenarios that resemble the one we use to perform the testing of a malware detector. The first scenario involves a system protected by a conventional malware detector, while the second one involves a system protected by a behavior-based detector.

**Scenario 1 – Conventional malware detector.** A user’s system gets infected by a malicious program because the conventional (signature-based) malware detector running on the system is not able to promptly detect and to prevent the infection (e.g., because the appropriate signature has not been published yet). Only later, the malware detector detects the presence of the malicious program on the system and cleans the system to get rid of the threat.
Scenario 2 – Behavior-based malware detector. A user is running a behavior-based malware detector on his system. The system is infected by a malicious program but the detector does not detect it until any malicious activity is observed. For example, consider malicious program that creates some files on the system and then tries to infect a running process. As the initial activity is legitimate, the malicious program is blocked only when it tries to infect other processes (or after the infection has taken place). The malware detector, after having detected the malicious behavior, repairs the system to rollback all the potentially dangerous activities performed before the detection.

Overview of the testing methodology.

Our goal is to measure remediation capabilities of the detector in any of the aforementioned scenarios. To accomplish this goal, we select a set of sample malware and we use each of these programs to infect a test system, we let to the detector to remediate the damages caused to the system by each infection, and finally we check the state of the system to see if the detector was able to revert the state to that prior to the infection. In other words, by infecting our test system with a malicious program we identify the set of system state transitions which are direct consequences of the infection and then we use these information to measure the completeness of the remediation procedure.

\[
\begin{align*}
(P_1) \text{ Execute & trace malware} & \quad \mathcal{S} = \langle s_0, s_1, \ldots, s_n \rangle \\
(P_2) \text{ Freeze malicious processes} & \quad (P_3) \text{ Abstract high-level behaviours} \\
& \quad \mathcal{S}' = \{ s \in \mathcal{S} : s \text{ modifies local system} \} \\
& \quad \mathcal{T} = \{ t : t \text{ abstracts a set of syscalls} \} \\
(P_4) \text{ Discard intangible transitions} & \quad \mathcal{T}' = \{ t \in \mathcal{T} : t \text{ is still valid} \} \\
(P_5) \text{ Trigger remediation} & \quad (P_6) \text{ Check for reverted transitions} \\
& \quad \mathcal{R} = \{ t \in \mathcal{T}' : t \text{ has been reverted} \}
\end{align*}
\]

Figure 6.3: Overview of our testing methodology. In gray we report the outcome of each phase.

A generalization of our testing methodology is outlined in Figure 6.3 and is summarized in the following paragraphs.
(P1) – Execute and trace the malicious sample. We select a malicious program which is detected by the malware detector under testing, and we run it in the test system. To simulate the scenario involving a conventional malware detector it is sufficient to disable the detector temporarily. On the other hand, to simulate the scenario involving a behavior-based detector the malicious program is run with the detector enabled. The execution is stopped when a timeout is reached or when the behavior-based malware detector detects a malicious behavior. As the execution of the malicious program is monitored by an external monitor, at the end of the execution we obtain $S$, the complete trace of the system calls invoked by the program during the execution.

(P2) – Freeze malicious processes. We freeze the state of the malicious program to prevent it from further altering the state of the system. Subsequent steps of the analysis will refer to that state.

(P3) – Abstract high-level behaviors. We analyze the recorded execution trace $S$ to extract $S'$, by excluding all the system calls that do not alter the state of the system (e.g., those used to open a file in read-only mode, or to read a registry key). Then, we analyze the resulting trace to infer the high-level behaviors of the program and the corresponding set $T$ of high-level system state transitions.

It is worth noting that we analyze only the behavior of the malicious process, and its children, and we do not consider high-level state-transitions associated with other processes running concurrently on the system. Thus, some of the high-level state transitions we analyze could conflict with those associated with other processes. To mitigate this problem without increasing the complexity of the analysis, we trace the malicious program in highly passive environments, with a minimal number of potentially conflicting processes and with no user interaction at all.

(P4) – Discard intangible transitions. Not all the observed high-level program behaviors lead to tangible system state transitions. As an example imagine our sample malicious programs that deletes the original executable after it has replicated. It is important to preemptively detect intangible state transitions because otherwise one might think that the transitions is reverted by the remediation procedure. For this reason, we identify such transitions and filter them out. The next phases of the testing will target only tangible transitions: $T' = \{ t \in T : t \text{ is tangible on the test system} \}$. 
(P₅) – **Trigger remediation.** Having collected all the information necessary to test the completeness of the remediation procedure, we can now trigger the malware detector to remediate the infection and to cleanup the system. In the case of a conventional detector we have to launch a full-system scan, which includes the scanning of all files and running processes. In the case of a behavior-based detector we have to authorize the detector to quarantine the malicious program; recall the behavior-based detector has been active since the beginning of the execution of the malicious program and it has already blocked the execution of the program.

(P₆) – **Check for reverted transitions.** Once the malware detector has completed the remediation, we have to check whether each of the high-level state transitions \( t \in T' \) has been properly reverted. Practically speaking, that means that we have to compare the state of the system prior to the infection with the state after the infection and the remediation, to detect any mismatch that can be ascribed to the malicious program. It is worth pointing out that we cannot expect the conventional malware detector to revert state transitions that caused data loss. On the other hand, it is legitimate to expect that from the behavior-based malware detector, as it has observed the whole execution of the malicious program since the beginning. At the end of this phase, we obtain a set \( R \subseteq T' \) of abstract transitions that have been reverted by the malware detector. If the remediation procedure is complete, then \( R = T' \); instead, if \( R \subset T' \), then every transition \( t \in T' \setminus R \) testifies the incompleteness of the remediation procedure for the malicious program used for the testing. It is worth noting that \( R \) could also include some state transitions that are not in \( T \). This happens when the malware detector incorrectly attributes a spurious action to the malicious program [89]. However, as our analysis is driven by the observed behaviors, we do not handle this situation.

### 6.4 Implementation

We have developed a prototype that implements the testing methodology discussed in the previous section, specific for testing malware detectors for Microsoft Windows. In this section, we discuss the technical details regarding the implementation of our testing infrastructure. The methodology described previously can be used to test the completeness of remediation procedures of both conventional and behavior-based malware detectors. In the following, we describe in detail only the implementation specific for the testing of conventional detectors. Nevertheless, the implementation for behavior-based detectors only differs in the
fact the detector is active when the malicious program is executed and traced.

![Figure 6.4: Architecture of the testing infrastructure](image)

Figure 6.4 depicts our testing infrastructure. The main components of our architecture are the victim test system, where the malware sample and the detector are located, and the analysis environment, where execution traces are analyzed. The malicious sample is uploaded into the test machine and its execution is monitored. Syscall traces are subsequently analyzed in the analysis environment, and further abstracted into high-level state transitions that are then verified. Finally, the malware detector is allowed to scan the whole system, and then the state of the system is checked to detect the set of transitions that have been reverted.

### 6.4.1 Tracing the malware sample

The malware sample is executed and traced in the test system (steps 1–3 in Figure 6.4). For the tracing we rely on our homemade system call tracer, code-named WUSSTrace [59], a user-space system call tracer for Windows. WUSSTrace parses the majority of the arguments of system calls, thus allowing a subsequent fine-grained analysis of the behavior of the program. Each intercepted system call is logged into an easy-to-parse XML trace, together with its input and output arguments. If the monitored process creates other processes or threads, these are monitored recursively. We are aware that user-space tracing can be easily circumvented by a nasty malware and that safer solutions exist (e.g., hooking from kernel space or through virtual machine introspection). However, we made this decision only to ease the development of our prototype.
We set a timeout on the execution of the malicious program and the other processes it creates. If a monitored process does not terminate spontaneously before the timeout expires, we freeze the process, by suspending the execution of all its threads. By freezing the malicious process instead of terminating it, we allow the malware detector to operate in a “best-case scenario”, where it can apply in-memory scanning techniques to analyze the memory image of the processes and to apply all the available heuristics.

6.4.2 Analysis of the system call trace

In order to analyze the system calls issued by the monitored malware sample, we developed a trace analysis tool that off-line performs the abstractions needed to infer the high-level program behaviors and the corresponding system state transitions. In our current implementation, we focus on the identification of the files, registry keys, processes, and system services that have been created or tampered by the malicious sample or by any of its child processes. For this reason, starting from a trace \( S \), we obtain (steps 4 and 5 in Figure 6.4) the set of system calls that modify the state of the environment (\( S' \)) by including only those syscalls that lead to the system state transitions of interest: file-system modifications (e.g., NtCreateFile, NtOpenFile, NtWriteFile), modifications of registry keys (e.g., NtCreateKey, NtSetValueKey), process creation or infection (e.g., NtCreateProcess, NtOpenProcess), etc.

To abstract \( S' \) into high-level behaviors and the corresponding set of state transitions \( T \), we need to correlate together the system calls that contribute to the same high-level behavior (step 6). In order to identify the syscalls responsible for a particular behavior (i.e., those that operate on the same resource) we employ standard data-flow analysis techniques [70]. The data-flow analysis is not fine-grained, as we do not log every single machine instruction executed by the monitored processes. Thus, dependency relationships between system calls are identified through handles (i.e., Windows resources identifiers): if system call \( s_2 \) uses handle \( h \) and the system call \( s_1 \) is the (dynamic) reaching definition for \( h \), then we can assume that \( s_1 \) and \( s_2 \) operate on the same resource. As an example, when we find in the execution trace a NtSetValueKey(\( r \), "v", "\p\oq.exe") system call we need to determine the name of the key that is being written; for this purpose, we compute the dynamic backward slice for the key handle \( r \) and we analyze the arguments of the system call that originally defined it [1]. Similarly, in order to compute the name of the files that are actually modified by the malware, we calculate the dynamic reaching definition for the file handle \( f \) used by the system call NtWriteFile(\( f \), "..."); this reaching definition will correspond to a NtCreateFile or NtOpenFile, and through the analysis of its
input arguments we can infer the name of the file being written.

### 6.4.3 Filtering of intangible high-level transitions

Having built the set $\mathcal{T}$ of high-level state transitions that represent the modification of the system caused by the malicious program, it is important to ensure that each transition $t \in \mathcal{T}$ is valid (i.e., it represents an actual modification of the state of the environment, that is tangible after the malicious program has terminated or it has been frozen). Indeed, any spurious state transition must be discarded, as it could negatively affect the accuracy of the evaluation of the remediation procedure.

As an example consider again our sample malicious program, whose high-level behavior is summarized in Figure 6.1. The program replicates its payload and then deletes the original executable. When the execution of the program in the test system terminates (or is frozen) the executable no longer exists on the system. If we do not test whether the file still exists on the system prior to the invocation of the malware detector we might erroneously praise the malware detector for something it has not done. On the other hand we want to be sure that system state-transition, even if not annihilated by the malicious program itself, are effectively tangible. The assumption that each write access to a resource of the system produces a modification of the system state might be too broad. For example, several malware often overwrite registry keys with the actual content of the keys; thus, despite the keys are overwritten, the system state does not mutate (this is probably a side effect caused by the use of some high-level libraries). A similar situation might occur with memory mapped files, because these files are written without invoking system calls and thus we have to conservatively assume that a file mapped with write permission is eventually modified.

We identify intangible state transitions by querying directly the test system in the exact same way we query it to detect if a transitions has been reverted by the malware detector (steps 7 and 8). Only tangible transitions $\mathcal{T}' \subseteq \mathcal{T}$ are targeted by the remaining phases of the testing. We detect registry keys or files that are effectively modified by comparing their actual content with their content preceding the infection. To do that we maintain a database of hashes of the content of all files and registry keys of the test-system before the infection. We discard all the behaviors that preserve the content of these resources. Similarly we also discard all the behaviors that involve the creation of files, registry keys, and processes that cannot be found on the test system at the end of the execution of the malicious program. Further details about how the test system is queried are given in the next paragraphs.
6.4.4 Execution and evaluation of the remediation procedure

At this point it is possible to trigger the malware detector to analyze the system and clean it from the infection. We invoke it to perform a full-scan of the file system, of the registry, and of the image of running processes (step 9 in Figure 6.4). We also enable all the heuristics supported to improve the detection and remediation rate. When the detector terminates the analysis of the system, we verify which of the state transitions associated with the execution of the malicious program have been reverted (step 10 and 11). Recall that the system state transitions $T'$ can be divided in multiple classes according to the type of resource affected by a transition. That is, $T' = F \cup R \cup P \cup S$, where $F$, $R$, $P$, and $S$ are the classes of transitions involving respectively files, registry keys, processes, and system services. Each class of transitions requires a particular procedure to verify whether the transition has been reverted or not. A transition $t \in T'$ is considered to be reverted by the malware detector when one of the following conditions is satisfied:

- if $t \in F$, the file subject of the transition is deleted or modified by the malware detector;
- if $t \in R$, the registry key subject of the transition is removed or modified by the malware detector;
- if $t \in P$, the process spawned by the malicious program is terminated;
- if $t \in S$, the system service created by the malicious program is disabled.

Note that we want to check if the modifications made by the remediation procedure to a resource manipulated by the malicious program successfully restore the initial state of the resource.

To test the aforementioned conditions, we leverage a small helper program we run in the test system, that allows us to query the state of a particular resource. For example, if we have observed the malicious program to create a registry key, we query the helper to check whether the key still exists on the system and, if so, to retrieve its contents and perform the appropriate comparisons: we expect the content after restoration is the same as the initial content (obtained by taking a snapshot of the initial, clean system).

6.5 Experimental results

This section presents the results of the testing of six of the top-rated commercial malware detectors. The goal of our experimental evaluation was to prove the
effectiveness of the proposed testing methodology and not to compare the tested malware detectors to tell which was the best and which was the worst. The experiments witnessed the effectiveness of our testing methodology. Indeed, they highlighted that none of the tested malware detector has complete remediation procedures. Furthermore, the experiments showed that the type and percentage of system state transitions reverted varies substantially among detectors.

### 6.5.1 Experimental setup

We tested the following malware detectors: Avast Professional 4.8, Kaspersky Anti-virus 2009, McAfee VirusScan Enterprise 8.5.0, Nod32 Anti-virus 3.0, Panda Anti-virus 9.0.5 and Sophos Anti-virus 7.6. We selected the malware detectors that facilitated the most the batch analysis, that is, those invokable directly from the command line and with the ability to cleanup the system automatically. We assumed that the detection capabilities of the command line version (with the proper arguments) and the GUI version corresponded. The virus definitions of each product were last updated on 15 January 2009. To discourage any direct comparison among the malware detectors, they were tested using different sets of about 100 malware samples, chosen randomly from a corpus composed by several thousand samples collected in the last quarter of 2008. All the samples tested were detected by the six detectors.

We performed the evaluation of our testing methodology using as test systems multiple VirtualBox virtual machines, each one running a different malware detector. To prevent other processes to alter the state of the system resources affected by the malicious programs used for the testing, we stripped down the virtual environments used for the analysis: we stopped all unnecessary services and processes and we did not interact at all with the environments. We traced the execution of the selected malicious program for five minutes and we performed all the steps of the analysis without restarting the test system. After each test, we restored the original clean state of the virtual machine.

### 6.5.2 Evaluation of state-of-the-art malware detectors

Figure 7.8 presents the overall results of our experiments. The names of the malware detectors have been anonymised to discourage comparisons. For each malware detector, we report the average percentage of system state transitions that were reverted. The average is computed on the total number of malware used to test each detector. The transitions are separated in two groups, according to their security impact on the system: primary and ancillary. Primary transitions are those that have a high impact on the system, while ancillary transitions have
Figure 6.5: Average percentage of system state transitions reverted by each malware detector.

A minor impact. A user should expect all primary transitions to be reverted by the malware detector, while he could tolerate if some ancillary transitions were not reverted. The partitioning of transitions in the two groups has a certain degree of subjectivity. We divided each of the transitions classes $F$, $R$, $P$, and $S$ in primary and ancillary as follows:

- $F$ (files): we consider as primary transitions all those that involve executable files (e.g., `.exe`, `.dll`, `.bat`, `.pif`, `.scr`), while as ancillary those involving the remaining types of files.

- $R$ (registry keys): we consider primary transitions those that involve registry keys that can be used to start programs automatically and ancillary all the remaining ones.

- $P$ (processes): we consider primary the transitions that create processes where the executed files match any of the files dropped on the system by the malicious program; the remaining processes started by the malware but executing programs already present on the system are instead considered ancillary.

- $S$ (services): for simplicity we treat system services as normal processes.

The graph in Figure 7.8 clearly shows the effectiveness of our testing methodology at evaluating the completeness of remediation procedures. None of the tested malware detectors turned out to be complete, even if only primary transitions
are taken into account: about 75% of the total primary transitions and only 4% of the total ancillary transitions were reverted.

A more detailed overview of the average distribution of primary and ancillary system state transitions reverted, for each transition class, is reported in Figure 6.6 (product names have been anonymised). While all malware detectors reverted the majority of primary transitions involving files, some of them (e.g., Vendor C and Vendor F) did not revert transitions involving registry keys at all. Other detectors instead (e.g., Vendor A and Vendor B) did not seem to terminate malicious processes, although we did not check the state of the system after a reboot.

We did not test interactively whether the system continued to work properly after infection and remediation. Indeed, there could exist situations in which an incomplete or improper remediation might render the system unusable. For example, imagine a malicious program that creates a registry key pointing to an executable, and that the existence of the key mandates the existence of the file (e.g., in Windows XP, the Image File Execution Options registry key). If the executable were removed, but the key were not, the system would stop working. We plan to address this problem in the future.

6.6 Discussion

Today, malware detectors are essential components for preserving the security of networks and computer systems. They allow to detect and prevent malicious
software and, when malware cannot be stopped from infecting a system, they allow to recover from the infection. In this chapter we presented an automated testing methodology to assess the completeness of remediation procedures used by malware detector to clean up compromised systems. We used this methodology to test six of the most rated malware detectors on the market and found out that the dangerous effects of an infection are seldom completely removed. In the next chapter we present our study on automatic technique for generating more complete remediation procedures.
7.1 Introduction

To defend against malware and others malicious programs, users typically rely on classical anti-malware software, which try to detect and prevent threats before systems can be damaged. Unfortunately, in many cases detection and prevention are not effective or possible. Signatures for detection are typically available only some time after a malicious program has gone in the wild, and therefore there exists a window of time in which even the most hardened systems are exposed to the threat. Furthermore, many users do not update the signatures database frequently enough, or buy an anti-malware software only after their system has been compromised. In such situations it is not sufficient to detect and remove the malicious program: it is also essential to remediate the system from all the secondary damages caused by the program. Ideally, infected systems should be reinstalled from scratch. This approach is costly and time expensive, and consequently users prefer a less drastic alternative: to rely on anti-malware software to revert the effects of the infection. Unfortunately, anti-malware products are not very good at this task. In the previous chapter we presented the results of our work demonstrating that often commercial anti-malware products, even the top-rated ones, suffer incompleteness (i.e., they remediate only partially the effects of infections) [74]. It is very easy to imagine how unsafe and unstable can be systems remediated only partially.

In this chapter we present a system for automatically generating malware remediation procedures, procedures that can be executed on infected systems to revert the effects of an infection. Ideally, a remediation procedure should be able to handle all possible effects of the execution of the malicious program, but should be as good as the ones that are tailored on the effects of a specific
execution (observed at run-time) [44]. Given a malicious program, our system automatically generates a set of procedures that aim at remediate all and only the effects of the execution of the malicious program in any possibly infected system, eventually, also, when the malicious program exhibits non-deterministic behaviors. The remediation procedures we want to generate are able to remove from any infected system all the resources (e.g., files and registry keys) created during the infection, but also to recover the original state of the resources that have been altered by the malware sample. We achieve this goal by learning the behaviors the malicious program manifests in different environments and by identifying the effect each behavior has on the system of the victim. For each behavior, we generate a remediation procedure that analyzes an infected system to identify potentially all, and only, the resources affected by the particular behavior, and that restores their state (e.g., deletes created resource and restores the ones modified). We are aware that our system is currently not complete and we cannot learn all the behaviour the malicious program can manifest only by changing the environments, but our framework could be further enriched with well known solutions proposed in literature in the recent years.

Our system is composed by three components. The first component analyzes dynamically multiple executions of the malicious program and infers the high-level behaviors that occur in each execution. More precisely, this component executes the malicious program multiple times in multiple heterogeneous execution environments (i.e., environments with different configurations), to exercise the largest class of behaviors. For each execution, the component records the high-level behaviors carried out by the program. The second component infers all the possible behaviors the malicious program can manifest. It performs clustering of the high-level behaviors detected with dynamic analysis; a cluster groups multiple occurrences of the same high-level behavior observed in multiple executions. Subsequently, the second component processes each cluster to construct a generalized model of the high-level behavior associated with the cluster. In other words, the generalized model represents all the possible occurrences of the behaviors in a cluster, including the occurrences not observed during the dynamic analysis. The third component analyzes each generalized high-level behavior constructed to detect the set of system resources the malicious program could affect, and to detect the possible effects of the execution of the program on these resources. Finally, the third component of the system generates a procedure that analyzes the system of a victim of the malicious program to identify if any of the resources the malicious program could possibly affect is present on the system, to look for evidence that support the hypothesis that such resources have been effectively tampered by the malicious program, and to remediate each of them.
To evaluate the proposed idea, we have developed a prototype implementation. Using this prototype, we analyzed more than 200 malicious programs (belonging to about 50 different families) and generated automatically the procedures to remediate the effects of each program. We infected with each malicious program 25 test environments, and then used the procedures generated by our system to remediate the infected environments. The results of the evaluation witness the effectiveness of our contribution. Our remediation procedures outperform those currently available in top-rated commercial anti-malware software: our procedures remediated 98% of the total effects of the infections while the most effective commercial anti-malware software remediated only 82% of the effects. The evaluation also demonstrates that the remediation procedures generated by our system identify very precisely the system resources affected by malicious programs. Only one of the remediation procedures involved system resources that should not.

To summarize, the chapter makes the following contributions:

- a system for the automatic generation of malware remediation procedures that can be executed on an infected system to revert the effects of the execution of a malicious program;
- a working prototype of the above mentioned system;
- an evaluation of the proposed solution over more than 200 malicious programs.

The chapter is organized as follows. Section 7.2 motivates our contribution and introduces a sample malware used thorough the chapter to present our work. Section 7.3 presents an overview of our approach for generating remediation procedures. Section 7.4 discusses the details of our approach. Section 7.5 presents the results of our experimental evaluation. Section 7.6 discusses some of the open challenges.

### 7.2 Motivations and goals

Consider the malicious program whose pseudo-code is shown in Figure 7.1 and that operates as follows. The program replicates its payload into a new file (e.g., c:\windows\poqwz.exe), creates a registry value to start the newly created executable at boot (\...\CurrentVersion\Run\vq), tampers the configuration of the DNS resolver (by modifying the file c:\windows\system32\drivers\etc\hosts) and infects a benign library (c:\windows\user32.dll). Finally, the malware deletes itself.
Chapter 7. Generating remediation procedures

There are two approaches to cleanup a system infected by a malicious program like the one just described: to reinstall the entire system or to remediate it (i.e., to remove all the effects of the infection). Given the cost of reinstallation, remediation is the ideal solution. Practically speaking, in the case of our sample program, to remediate an infected system means to: (1) delete the file containing a copy of the malicious payload, (2) delete the registry key created to start the malware at boot, (3) disinfect the infected DLL, and (4) restore the original configuration of the resolver. Unfortunately, even such a simple malicious program poses several challenges for remediation. The hardest challenges to solve are connected with the non-determinism in the behavior of the program and the alteration of existing system resources. Indeed, our malware sample creates files and registry keys with semi-random names (Figure 7.1, lines 2, 3, 6, and 13) and modifies existing files (lines 17 and 22). Moreover, although the problem is not evident through our sample malicious program, certain behaviors could be triggered only in particular situations.
To understand the importance of remediation, imagine an anti-malware software that reverts all the actions performed by the malicious program of Figure 7.1, but that fails to restore the configuration of the resolver (i.e., to remove the malicious entries added to the file `c:\windows\system32\drivers\etc\hosts`). Even though all the malicious executables dropped by the malware are removed from the system, the security of the user is still compromised because part of the network traffic continues to be hijacked to a malicious website, with easy to imagine consequences.

Our goal is to generate automatically an optimal remediation procedure. A remediation procedure is a procedure that analyzes a system infected by a particular malware sample to detect all the effects of the infection, removes all these effects, and prevents unpleasant situations like the one described in the previous paragraph. Note that our intent is to generate a sufficiently generic remediation procedure, that can be applied to any system infected with a specific malicious program. The optimal remediation procedure is sound and complete. The optimal procedure must not affect the system resources (i.e., files, registry keys, services, processes) that have not been altered by the malicious program (soundness). Moreover, it is also essential that the remediation procedure removes all the effects of the execution of the malicious program, and that restores a safe state of the system (completeness), even in the presence of a certain degree of non-determinism in the behavior of the program.
Chapter 7. Generating remediation procedures

7.3 Overview of the system

The architecture of the system we have developed for generating automatically remediation procedures is shown in Figure 7.2. The system is composed by three main components: (1) the high-level behaviors extraction component, (2) the behaviors generalization component, and (3) the remediation procedures generation component. The input to our system is a malicious program and the output is a specific remediation procedure for repairing the system from the effects of the execution of the input program. Currently our system supports only user-space malware, although the support for kernel-level can be introduced with few efforts. In the remaining of the section we describe briefly the three components. Further details are given in Section 7.4.

7.3.1 High-level behaviors extraction

The high-level behaviors extraction component (component 1 in Figure 7.2) is responsible for analyzing a malicious program, for detecting the high-level operations (or behaviors) the program performs (e.g., file system activity, network activity), and for selecting those that alter the state of the system and that should be reverted in order to restore the state the system had before the infection. Since malware typically use anti-analysis techniques (e.g., packing and other forms of code obfuscation), we adopt dynamic analysis, which is more resilient against various forms of code obfuscation. We monitor the execution of the malicious program in a special execution environment, that allows us to intercept all the system calls executed by the program, and we leverage successful research to reconstruct meaningful high-level operations from a low-level stream of system calls [61].

The behavior of a program is typically affected by the environment in which the program is executed. Furthermore, in certain cases, even in the same execution environment we can observe different behaviors. That is exactly the case of our sample malware of Figure 7.1. Since one of the most important requirements of a remediation procedure is completeness, it is fundamental to observe the largest class of behaviors a malicious program can manifest. For this reason, we analyze each malicious program multiple times, using different execution environments. The sequence of system calls executed in each environment is recorded \((S^1, \ldots, S^4)\), and abstracted separately into the corresponding sequence of high-level behaviors \((B^1, \ldots, B^4)\).

Table 7.1 reports the high-level behaviors we consider. Note that we are interested only in behaviors that modify the local state of the system. Each high-level behavior reported in the table is associated with a set of arguments...
### Table 7.1: High-level behaviors considered for remediation

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileCreation</td>
<td>File name and content</td>
<td>Creation of a new file</td>
</tr>
<tr>
<td>RegistryCreation</td>
<td>Key name and content</td>
<td>Creation of a new registry value</td>
</tr>
<tr>
<td>DropAndAutostart</td>
<td>File name and content, Key name and content</td>
<td>Creation of a new file and of a registry value containing its name (to execute the file automatically at every boot)</td>
</tr>
<tr>
<td>DropAndExecute</td>
<td>File and process name</td>
<td>Creation and execution of a new executable</td>
</tr>
<tr>
<td>FileInfection</td>
<td>File name and content, List of preserved regions</td>
<td>Infection of an existing file</td>
</tr>
<tr>
<td>RegistryInfection</td>
<td>Key name and content</td>
<td>Replacement of an existing registry value</td>
</tr>
<tr>
<td>FileDeletion</td>
<td>File name</td>
<td>Deletion of an existing file</td>
</tr>
<tr>
<td>RegistryDeletion</td>
<td>Key name</td>
<td>Deletion of an existing registry value</td>
</tr>
</tbody>
</table>

describing the resources affected by the behavior and the actions performed on the resources. Behaviors are grouped in three classes. The first class represents behaviors involving the creation of new system resources (e.g., the creation of new files and new registry keys). The second class represents behaviors involving the infection, or alteration, of existing system resources. Finally, the third class represents behaviors involving the deletion of existing system resources. The high-level behaviors in the table represents the activities that any user-space malicious program performs. The techniques we are presenting can be used also with other behaviors, for example involving persistent mutexes and named pipes.

#### 7.3.2 Behaviors generalization

To partially contrast the incompleteness of dynamic analysis we monitor multiple executions of the same malicious program, in heterogeneous execution environments, and for each execution observed we infer the set of high-level behaviors characterizing it. The next step is to identify when multiple behaviors, from different executions, are actually instances of the same malicious activity. The component for behaviors generalization (component 2 in Figure 7.2) processes the output of the first component, consisting in a set of high-level operations (or behaviors) performed by the malicious program in the various executions.
monitored, to construct new and generic models of these behaviors. To handle non-determinism in the behavior of the malicious program, this component generates models that tolerate differences in the sequences of system calls executed by the program to carry out the various high-level operations, including similar sequences of system calls not seen in the executions analyzed by the first component.

The generalization is a two phases process. The high-level behaviors observed in the various executions are clustered to group behaviors representing the same malicious activity (e.g., file activity representing the replication of the malicious program, and registry activity representing the infection of the system register). In Figure 7.2, clusters are indicated with $C_1$, $C_2$, and $C_3$. Subsequently, each cluster is transformed in a new, but generic model of the associated high-level behavior, that synthesizes all the possible variations of the behavior that could be produced by the malware sample under analysis.

### 7.3.3 Remediation procedures generation

The third component of our architecture (component 3 in Figure 7.2) is responsible for generating remediation procedures. A remediation procedure is a program (or procedure) that analyzes a system infected by a particular malware sample, detects the effects of the infection and removes them. For each of the high-level generalized behaviors produced in output by the second component of our architecture (see Table 7.1), we generate a specific procedure. The union of these procedures represents the full remediation procedure we generate for a particular malicious program.

A remediation procedure for a behavior involving the creation of a new system resource removes from the system a resource (or a set of resources). A remediation procedure for a behavior involving the infection or the deletion of an existing resource tries to restore the original state of the resource, or at least to revert it to a known safe state (e.g., the default). All classes of remediation procedures apply the requested steps to remediate a compromised system only if the claim that the resource being remediated has been affected by the malicious program is supported by tangible evidence. That is, before removing or recovering a resource, the remediation procedure verifies whether all the constraints dictated by the model describing the high-level malicious behavior associated with the remediation procedure are satisfied. More precisely, the procedure verifies that: (i) the names of the resources whose creation (or modification) is attributed to the malicious program match the names of the resources described by the generic model, (ii) the content of these resources also match the model, and (iii) optional dependencies with other resources related to the malicious activity are satisfied.
Chapter 7. Generating remediation procedures

\begin{verbatim}
\begin{verbatim}
s_1 \text{NtCreateFile("c:\windows\poqwz.exe")} \rightarrow f
s_2 \text{NtWriteFile(f, \"...malicious code...\")}
\end{verbatim}
\begin{verbatim}
s_3 \text{NtWriteFile(f, \"...other malicious code...\")}
s_4 \text{NtClose(f)}
\end{verbatim}
\begin{verbatim}
... 
\end{verbatim}
\begin{verbatim}
s_{11} \text{NtOpenKey("...\Windows\Current\Version\Run")} \rightarrow r
s_{12} \text{NtQueryValueKey(r, \"vq\")} \rightarrow \text{FAILURE}
\end{verbatim}
\begin{verbatim}
s_{13} \text{NtSetValueKey(r, \"vq\", \"poqwz.exe\")}
s_{14} \text{NtClose(r)}
\end{verbatim}
\begin{verbatim}
... 
\end{verbatim}
\begin{verbatim}
s_{21} \text{NtOpenFile("c:\windows\system32\user32.dll")} \rightarrow g
s_{22} \text{NtClose(g)}
\end{verbatim}
\begin{verbatim}
... 
\end{verbatim}
\begin{verbatim}
s_{31} \text{NtOpenFile("c:\windows\system32\drivers\etc\hosts")} \rightarrow h
s_{32} \text{NtReadFile(h, 1024)} \rightarrow \text{"# Copyright (c)..."}
\end{verbatim}
\begin{verbatim}
s_{33} \text{NtWriteFile(h, \"67.42.10.3 www.google.com...\")}
\end{verbatim}
\begin{verbatim}
s_{34} \text{NtWriteFile(h, \"67.42.10.3 www.citibank.com...\")}
s_{35} \text{NtClose(h)}
\end{verbatim}
\begin{verbatim}
... 
\end{verbatim}
\begin{verbatim}
s_{41} \text{NtDeleteFile("c:\malware.exe")}
\end{verbatim}
\end{verbatim}

Figure 7.3: System call trace of our sample malicious program \textit{(malware.exe)}

7.4 Architecture details

This section presents in detail the three main components of our architecture, introduced in the previous section.

7.4.1 High-level behaviors extraction

We execute a malicious program into a virtual environment, and monitor and log all the system calls performed by the application, with the respective arguments. Our virtual environment is based on QEMU [8], a whole system emulator, which has been extended to perform the required monitoring and logging. Figure 7.3 reports a system call trace obtained by monitoring an execution of our sample malicious program with the aforementioned environment.

Given a system call trace $S = \langle s_1, s_2, \ldots, s_n \rangle$, the next step consists in analyzing the trace to infer the corresponding high-level execution trace $B = \langle b_1, b_2, \ldots, b_k \rangle$, consisting in the high-level behaviors performed by the malicious program during the monitored execution. To infer high-level behaviors from a stream of system calls we use multilayer behavior specifications, as proposed in [61], where complex high-level behaviors are described using hierarchical models of incremental
specificity.

We have constructed models to identify the high-level behaviors reported in Table 7.1. These models, or behavior graphs, are multilayer, in the sense that the nodes of the highest level graphs are new behavior graphs whose nodes are in turn other behavior graphs, and so on. The top layer models the high-level actions of the behaviors, and the underlying layers provide incrementally more details about how each action can be carried out. Lowest-layer graphs represent the system calls a program can execute to perform a particular action (e.g., to open a file). Edges between nodes of the same layer represent dependencies. For example, system calls operating on the same resource handle are considered dependent and edges between these system calls are used to represent data-dependencies on the handle. Similarly, higher-level operations on the same file are also marked as dependent using edges. The advantage of using such a layered description of the behaviors is twofold. First, the graphs are very easy to construct and to manage; for example, lowest-layer behavior graphs can be updated without having to update higher-level ones. Second, the graphs can be easily constructed to account for all the possible ways a program can carry out a particular action. Thus, we can construct models of program behaviors that are resilient against malicious attempts to obfuscate the behaviors (e.g., using system calls reordering, nop system calls, or multi-threading). Further details about behavior graphs can be found in [61].

To infer the high-level execution trace $B = \langle b_1, b_2, \ldots, b_k \rangle$ performed by the malicious program under analysis, we process the system call trace $S = \langle s_1, s_2, \ldots, s_n \rangle$ obtained by monitoring the program in our virtual environment and we look for sequences of system calls that match those of the behavior graphs that model the behaviors of interest. For each high-level behavior found, we produce in output two different representations: a concrete graph of the behavior, and a summary of the behavior. The concrete graph of the behavior, $b_i$, is an instance of the graph used to model the behavior. The leaf nodes of the graph are replaced with the system calls (with their arguments) found in the trace that match the nodes of the model; nodes not matched are simply omitted. The summary of the behavior instead describes the key information about the behavior inferred: the type of the behavior and the most relevant arguments (e.g., the names of the system resources involved).

Figure 7.4 shows two of the high-level behaviors extracted from the system call trace of Figure 7.3. The figure shows both the concrete graphs and the summary of the behaviors extracted. The first four system calls in the trace ($s_1$ to $s_4$) are executed by the sample malicious program to replicate its payload into a new file. We associate these calls with the behavior FileCreation. Similarly, we associate the subsequent system calls ($s_{11}$, $s_{13}$, and $s_{14}$) with the
Concrete graphs of the behaviors

Summary of the behaviors
DropAndAutostart("c:\...\poqwz.exe", data, "...\Run", "vq", "poqwz.exe")
FileCreation("...\poqwz.exe", data)
FileDeletion("c:\malware.exe")
RegistryCreation("...\Run", "vq", "poqwz.exe")

Figure 7.4: High-level behaviors extracted from the system call trace of Figure 7.3

behavior RegistryCreation and the last system call ($s_{41}$) with the behavior FileDeletion. Since the behaviors FileCreation and RegistryCreation are related (i.e., the name of the executable, excluding the path, is the same), we are able to infer the higher-level behavior DropAndAutostart, that represents the fact that the malware replicates and configures the system to execute the malicious payload at boot.

7.4.2 Behaviors clustering

Given a set of high-level execution traces $\{B_1, B_2, \ldots, B_m\}$, obtained by monitoring the same malicious program in multiple execution environments, the next step is to analyze the traces to infer if multiple behaviors from different traces are actually instances of the same malicious activity. The intent is to understand all the possible ways the various malicious activities can manifest.

As an example, Figure 7.5 shows two high-level execution traces of our sample malicious program, $B_1 = \langle b_1^1, b_1^2, \ldots \rangle$ and $B_2 = \langle b_2^1, b_2^2, \ldots \rangle$. Note that we denote with $b_i^j$ the $j^{th}$ behavior observed in the $i^{th}$ execution trace. Our goal is to group the various high-level behaviors of the two traces together, when they represent the same malicious activity (e.g., the infection of the resolver). In our example, we want to group behaviors $b_1^1$ and $b_2^1$ since they correspond to the same activity:
the malicious program installs a hook in the Windows registry to let the operating system start the dropped executable when the machine is restarted. Similarly we want to group the behavior $b_1^2$ with $b_2^2$, and $b_1^3$ with $b_2^3$. Unfortunately, our sample program creates files and registry keys with partially random names, so some of the high-level behaviors of the two execution traces might appear unrelated. In order to generate sufficiently generic remediation procedures, it is essential to detect that the aforementioned behaviors are actually related and to tolerate the non-determinism in the behavior of the malicious program.

Our approach to group the high-level behaviors observed in multiple execution traces consists in comparing the graphs associated to each behavior and in grouping together similar graphs. There is an important advantage in comparing the graphs of the behaviors instead of their summaries: the graphs describe how the malicious program carried out each activity (i.e., the sequence of system calls involved), while the summaries just describe the effect of each activity. Moreover, non-determinism in a malicious program typically affects the summary of the behavior (e.g., the name of the files or registry values created), but does not affect the low-level operations used to achieve the behavior (e.g., the sequence of system calls executed to create a file). Therefore, our approach to group behaviors allows to perform more thorough comparisons, is less sensitive to non-determinism, and
Chapter 7. Generating remediation procedures

gives more and more precise results.

Clusters generation

The problem of grouping high-level behaviors observed in multiple execution traces can be reduced to a clustering problem. Given $m$ high-level execution traces $\{B^1, B^2, \ldots, B^m\}$, our goal is to find a set of clusters $\{C_1, C_2, \ldots, C_k\}$, where each cluster includes the behaviors that are instance of the same activity. As an example, starting from the two sample execution traces depicted in Figure 7.5, we would like to identify the cluster $C_1 = \{b_1^1, b_1^2\}$ that includes the two DropAndAutostart behaviors. In our context, clusters must be disjoint ($C_i \cap C_j = \emptyset$), each behavior must belong to exactly one cluster ($\bigcup_{i=1}^m C_i = \bigcup_{i=1}^m B^i$), and trigger-dependent high-level activities might be missing in some execution traces ($|C_i| \leq n$).

Our clusters generation method is a slight variation of the $k$-means algorithm [57]. In particular, let $|B^i|$ be the length of the $i^{th}$ high-level execution trace, we aim at partitioning the set $\bigcup_{i=1}^m B^i$ into $k$ clusters, with $k = \max\{|B^i|\}$. The distance between a behavior $b^j_i$ and a cluster $C$ is 0 iff both the following conditions hold: (i) no other behavior from trace $B^j$ is already in $C$, and (ii) $b^j_i$ is isomorphic to a representative element of $C$ (see Section 7.4.2 for details). Otherwise, the distance between $b^j_i$ and $C$ is $+\infty$. Note that, as a cluster defines an equivalence class with respect to the graph isomorphism relation, any cluster element can be chosen as the representative.

In summary, the output of the behaviors clustering phase is a set of clusters $\{C_1, C_2, \ldots, C_k\}$ of high-level behaviors, where each cluster encompasses behaviors that implement the same malicious activity. Back to our sample high-level execution traces reported in Figure 7.5, we expect the behaviors clustering phase to produce three different clusters as follows: $C_1 = \{b_1^1, b_1^2\}$, $C_2 = \{b_2^1, b_2^2\}$, and $C_1 = \{b_3^1, b_3^2\}$.

Behaviors comparison

In order to cluster high-level behaviors we need a strategy to compare efficiently the behaviors found in the various execution traces. We compare high-level behaviors by checking whether the corresponding graphs are isomorphic or not. However, using a naive isomorphism, the clustering would not be resilient to non-determinism in the behaviors of the program since it could not tolerate any negligible difference in the graphs. As an example consider again the behaviors $b_1^1$ and $b_1^2$ shown in Figure 7.5. Although it is evident that both behaviors are instances of the same malicious activity and their graphs are very similar, they are not isomorphic (the name of the executable created differs, and the number
Chapter 7. Generating remediation procedures

of system calls associated to the **FileCreation** behavior is also different). To address this problem, we normalize the graphs before testing if they are isomorphic. Details about normalization are given in the next section.

Graph isomorphism is a hard problem: currently, no polynomial-time algorithm to solve it is known. However, the graphs describing malicious behaviors are typically very small (few dozen nodes) and thus the complexity of the problem does not affect us. In our current implementation we use the VFlib2 algorithm [31].

**Behaviors normalization**

Clustering of behaviors based on naïve graphs isomorphism is excessively sensitive to non-determinism. The solution we adopt to address this problem consists in normalizing the graphs representing high-level behaviors to hide all the differences that are typically the consequence of non-determinism in the behavior of the malicious program. Obviously the challenge is to find the most appropriate normalization function, that is capable of hiding all the effect of non-determinism, but still allows to distinguish behaviors that are instances of different malicious activities.

The normalization function we developed to pre-process graphs during clustering has been constructed leveraging all our experience about malicious programs and about the way they are developed and operate. Normalization involves the leaf nodes of the graphs, i.e., the system calls monitored during the execution of the program. Some examples of the transformations we perform during normalization follows. System calls arguments representing resources names (e.g., file names) are all replaced with the same constant value indicating the type of resource. For example, we use a different constant for each file type and registry key type. Similarly, system calls arguments representing data being written to resources (e.g., data being written to a file) and offsets are also replaced with a constant. System calls operating sequentially on the same resource (e.g., consecutive writes) are merged in a single synthetic system call. During merging we take into account the order in which system calls occurred and the absolute offset of writes. Finally, we ignore sequences of system calls that operate on the same resource and that kill the effects of the previous ones (e.g., multiple file writes at the same offset); in this case we consider only the last call.

As an example, consider again the high-level behaviors shown in Figure 7.4. Behaviors normalization merges together the multiple and consecutive write operations of the **FileCreation** behavior (syscalls $s_2$ and $s_3$) and replaces the file names and the data written to file with constants. After normalization the two behavior graphs become isomorphic.
7.4.3 Behaviors generalization

Each cluster groups all the instances of the same malicious activity observed in the various executions. The next step is to construct a model of the malicious activity associated with a cluster. The model generalizes the variations of the behavior found in multiple execution traces, and it is then used to construct a remediation procedure able to identify the system resources that have been affected by the malicious activity and consequently should be the target of the remediation. By generalizing the behaviors in a cluster, we relax the constraints used to identify these resources, and produce a remediation procedure that is more resilient to non-determinism in the behavior.

The behaviors in a cluster are generalized by considering the summaries of the behaviors and by generalizing each of their arguments separately. Arguments represent the names of the resources involved with a particular behavior and other information about how resources are modified (e.g., what data are written to each resource). Since all arguments can be converted into strings of characters, we reduce the problem of generalizing a high-level malicious behavior to the problem of generalizing a set of strings, where each set contains all the variations of a certain argument of the behaviors in the cluster. We generalize a set of strings through a regular expression that matches all the strings in the set and possibly others not found in the set. Ideally, the regular expression should match all the strings encoding the arguments the malicious program could generate, and nothing else. Obviously, this is not possible without an accurate inspection of the code of the program. For this reason, we relax the goal and try to infer an approximation of the grammar used for generating the argument strings, starting from the analysis of the set of concrete arguments observed.

As an example, Figure 7.6 reports a sample cluster grouping seven different occurrences of the DropAndAutostart behavior manifested by our sample malware (the corresponding graphs are omitted for conciseness).

![Figure 7.6: Sample cluster grouping seven different occurrences of the DropAndAutostart behavior manifested by our sample malware (the corresponding graphs are omitted for conciseness).](image-url)
Chapter 7. Generating remediation procedures

changes at each execution. In this situation, we would like to generate a regular expression that matches each of the names observed during the monitored executions, but is generic enough to accept also other file names the malware might generated in future executions. The generated expressions should treat as constants all the sub-strings common to all the arguments and generalize only the ones that change from one argument to the other (those underlined in the figure).

Given a set of strings, representing the various occurrences of the same argument in a cluster, we generalize them as follows. We start by constructing a finite state automaton that accepts exactly all the strings in the input set. Subsequently, we minimize the automaton using probabilistic learning, obtaining a minimal probabilistic finite state automaton (PFSA for short). The PFSA is constructed using the *simulated beam annealing* algorithm [84]. The advantage of using a probabilistic minimization algorithm, over a traditional algorithm, is that during the transformation a first step of generalization takes place. Indeed, the PFSA construction algorithm tends to merge states that are probabilistically very similar. The minimized automaton is then converted into a regular expression adopting standard algorithms [42].

During the conversion from PFSA to regular expression we perform widening to further generalize the resulting expression: that is, we identify high-density regions of the automaton, imputable to non-determinism in the behavior, and replace them with wildcards. High-density regions subject to widening are single-entry-single-exit regions of the automaton with a large number of paths [47]. In our case, single-entry-single-exit regions are regions of the automaton where: (i) the entry node is a state of the automaton that is the immediate dominator of the exit node; (ii) the exit node is the immediate post-dominator of the entry node; (iii) the region contains more than one path from the entry node to the exit node. Intuitively these regions encode the sub-strings of the arguments that differ among the various executions. We identify single-entry-single-exit regions of the automaton and analyze the paths from the entry to the exit node to decide whether to perform widening or not. Regions characterized by a large number of paths are widened introducing wildcards in the regular expression. The more dense the region (i.e., the larger the number of paths), the more aggressive the widening applied. Our widening algorithm is rule-based. Rules for widening have been chosen empirically and are selected considering information such as the number of paths in the region, the probabilities associated with the paths, the standard deviation of the length of the paths, and the characters composing the strings associated with each path.

As an example, consider again Figure 7.6. From the first argument, using the technique described above, we construct the minimal automaton shown in Fig-
Figure 7.7: A fragment of the minimized automaton constructed to generalize the first argument of the DropAndAutostart behavior, starting from the occurrences of the argument reported in Figure 7.6.

Subsequently, we convert the automaton into a regular expression. The automaton contains a single-entry-single-exit region with more than one path, highlighted in the figure, that encodes the variable substring of the filename. Since this region is quite dense, we apply widening during the conversion and use a wildcard instead of expliciting all the possible substrings the region can encode. The resulting regular expression is `c:\windows\po[[:alpha:]]{3}\exe`, where `[:alpha:]]{3}` denotes a string of length three composed by any uppercase or lowercase alphabetic character. Such an expression is capable of identifying all the names of the files that our sample malicious program could drop on the system. Indeed, our sample malware generates the last three characters of the filename randomly (lines 2 and 6 of Figure 7.1). After having applied the generalization to all the arguments of the DropAndAutostart behavior associated with the cluster of Figure 7.6, we obtain a generic model of the behavior like `DropAndAutostart("c:\windows\po[[:alpha:]]{3}\exe", data, "...Windows\CurrentVersion\Run", "(vq|qv)", "po[[:alpha:]]{3}\exe")`. It is worth noting that the regular expression `"(vq|qv)"` captures exactly the fact that the argument is an enumeration (line 3 and 13 of Figure 7.1).

7.5 Evaluation

This section presents the results of the testing of our technique for the automatic generation of malware remediation procedures. In particular, our experimental evaluation aimed at assessing the completeness and soundness of the generated procedures. Our results testify the effectiveness of our contribution: our remediation procedures are not far to be complete and sound for the malwares we
analyzed. More than 98% of total malicious activities were successfully reverted
from the infected system. Therefore, our automatically generated remediation
procedures outperforms those found in top-rated malware detectors currently
available on the market [74]. Moreover, only one of the remediation procedures
our system generated overly identified resources to remediate.

7.5.1 Experimental setup

The experiments were performed over a corpus of 200 malicious programs, chosen
randomly from our own database of malware. This malware corpus encompassed
more than 50 different families. Remediation procedures were built for each
sample by analyzing it in multiple execution environments. In order to exercise
the highest number of different behaviors, we tried to make the virtual execution
environments used to execute and monitor the programs as much heterogeneous
as possible (e.g., by installing different applications, changing the system language
and timezone).

To quantify the completeness and soundness level of the generated procedures,
we employed the testing methodology proposed in [74]. For each malware sample,
we performed the following steps.

1. Using the technique described in the previous sections, we analyzed the
malware sample in five different execution environments, three times per
environment, and generated automatically the appropriate remediation pro-
cedures. On average we generated about 20 procedures per sample.

2. We infected 25 different test environments with the malware sample (the set
of test environments was disjoint from the set of execution environments).

3. We executed the generated remediation procedures in each test environ-
ment.

4. Finally, for each test environment, we verified whether the procedures suc-
sessfully reverted all the activities performed by the malicious program
(completeness), and whether the remediation did not involve any system
resource that was not affected by the malicious program (soundness).

7.5.2 Completeness of the generated remediation proce-
dures

Figure 7.8 compares the completeness of our automatically generated remediation
procedures with the completeness of the procedures included in three top-rated
commercial malware detectors evaluated in [74]. The graph reports the average number of malicious activities that were reverted, computed over the whole corpus of malware samples. Malicious activities are divided in three classes, according to the type of system resources involved with each activity: involving files, registry keys, and processes. Each class is further divided in two sub-classes, according to their security impact of the resource on the system: activities involving primary resources are more security relevant, while those involving ancillary resources are less dangerous. More precisely, the creation of an executable file (e.g., .exe, .dll, .bat, .pif, .scr) is considered a primary activity, while the creation of other types of files is considered ancillary. Similarly, all activities involving registry keys that can be used to start programs automatically are considered primary, while all the remaining ones are ancillary. Finally, activities involving processes started from files dropped or infected by the malware and processes whose memory image has been infected are considered primary, while the activities involving other processes are considered ancillary.

In the majority of cases, the completeness of our remediation procedures outperforms the completeness of commercial anti-malware products. As an example, the generated procedures were able to remediate more than 99% of primary file activities, whilst the best commercial product we tested reached only 82%. Similarly, our automatically generated procedures remediated 99% of primary registry
activities, while commercial products never exceeded 86%. It is worth mentioning that ancillary activities are very often ignored by commercial anti-malware products; instead, our procedures reverted 95% of ancillary file activities and more than 98% of ancillary registry activities. The very small percentage of files and registry activities that were not remediated by our procedures corresponds to activities recorded in some test environment, but that had never been observed while the malware sample was monitored in the execution environments. This percentage could be further reduced by increasing the number of execution environments used for the construction of the procedures, together with the number of runs per environment. Finally, our procedures successfully remediated 100% of primary process activities. However, the percentage of ancillary process activities remediated is significantly lower: such activities are not handled by the generated procedures, as we do not have enough information to discern benign processes spawned by a malware sample from processes spawned by users. In the future, we plan to address this limitation by taking into account more information about the created processes (e.g., the title of the window associated to the process).

7.5.3 Soundness of the generated remediation procedures

While the event of some ancillary activities that are not reverted is likely to be tolerated by end-users, the improper modification of existing resources is unacceptable. For this reason, the soundness of the generated remediation procedures is an essential aspect of the evaluation of our proposed methodology. To quantify the soundness we compared the set of resources affected by each malicious program in each test environment with the set of resources our procedures remediated. The procedures that remediated resources not affected by the malicious program were considered unsound.

Overall, only one of the procedures we generated was unsound. Recall, that for each of the malicious programs used for the testing we generated on average about 20 procedures, where each one was responsible for a specific high-level behavior of the malicious program. The cause of the unsound procedure was an overly generalization of behavior arguments, caused by excessive widening during the generation of regular expression to identify resources the malicious program can affect. In turn, excessive widening might be caused by imprecision of the clustering algorithm, but most likely, by highly non-deterministic behaviors (e.g., a malicious program that drop executables, with different payloads, anywhere on the file-system).

Although ideally all remediation procedures should be sound, we believe it is reasonable to accept a small number of unsound procedures. The approach we used for the evaluation can also be used to identify potentially unsound pro-
cedures before they are released to end-users. Preemptively identified unsound procedures can be analyzed and optimized by human analysts to make them sound.

7.6 Discussion

One of the key points for generating a fully complete remediation procedure is to observe the multiple and diverse execution of the malicious program. Although, for simplicity, we used multiple installations of the same operating systems with different configurations, more sophisticated setups can be used to improve the completeness of the analysis [65, 106]. We are also aware that running a malware program in different environments (even when they are many) does not guarantee that all behaviors are covered, especially for today bots that perform actions in response to external commands. Currently we are not able to handle these behaviors in a different way, so we expect that for these kind of malwares our framework will generate incomplete remediation procedures.

Our remediation procedures are currently specific for remediating the effects of a particular malware sample and are supposed to be executed on a system infected by such a sample. An interesting enhancement in our framework could be the development of a remediation procedure generator for multiple variants of the same malware family; however we expect the whole procedures soundness would be degraded.

We also believe that current malicious programs could be easily integrated with smart tricks or corner cases that we currently miss, in order to create complications or impede automated generation of remediation procedures.
Conclusion and Future Work

Over recent years we noticed a massive increase in cybercrime activities: attackers can often compromise victim’s machines and obtain complete control over them for example by using technical means like remote service exploits or by taking advantage of vulnerabilities on client applications. Today also social techniques like social engineering are massively used to trick victims into opening malicious email attachments, or into clicking on malicious web links. All these attacks are commonly used by criminal organizations to compromise large groups of hosts (bots) that will be remotely controlled through the network. These large group of infected hosts are commonly named botnets. We can easily imagine that in the next years botnets will be further used by criminals: currently no completely effective countermeasures against these threats have been found. Botnet coordinated denial of service attacks became popular as also massive junk mailing events or distributed web application attacks, orchestrated sensitive data leakage episodes, aggressive vulnerability scans.

The current mitigation and remediation approaches to all these kind of attacks have several limitations that make sometimes implemented defense strategies ineffective and after infection remediation activities incomplete. Deep attacks analysis is fundamental both for effective mitigation and detection activities and also for complete and accurate remediation, but unfortunately are also complex and time consuming. In this dissertation we proposed effective strategies to fight some recent botnet techniques and attacks that can be implemented in a layered defense architecture. We proposed automated approaches that could be adopted to effectively protect systems against newest attacks, trying to anticipate attackers moves, even if attackers will always find novel ways to compromise systems and use them for nefarious purposes. In the following, we briefly summarize the results of this dissertation outlying directions for future research in each involved area.
Chapter 4: Fast-Flux Service Networks
We studied a particular technique used by malicious remote control networks, named fast-flux. The idea behind this technique is that the attacker does not directly abuse the compromised machines, but uses them to establish a layer of indirection, like a proxy, on top of these infected host in order to hide the identity of botnet core components and to enhance the whole infrastructure availability. We proposed and implemented a precise methodology to identify fast-flux service networks (FFSNs) in an high automated way. Classification strategy is based on all distinguishing features characterizing analyzed domains. Data collected for each domain enable us to decide in a precise way whether or not a given domain name uses the fast-flux technique. Furthermore, we tried to understand the underground economy related to fast-flux service networks trying to analyze and group together scam sites advertised through fast-flux domains inserted into junk mails. Even if we collected lots of malicious domains our understanding of criminal organizations activities is rather limited up to now, thus we would to closely study this phenomenon in order to learn more about their future steps.

Chapter 5: Web attacks mitigation
The spreading of the botnet phenomenon reveals a very critical point of the Internet Infrastructure, namely the web applications, which are the most used attack vector used so far for the malware spreading. Attackers have also been increasingly using the web both client and server side attacks in order to steal information from targets and redirect victims to compromised sites containing malicious codes. In this dissertation we presented a hybrid analysis framework for the detection of vulnerabilities in web applications, like SQL injections and XSS attacks, blending together the strengths of static and dynamic approaches. Experimental results demonstrates that our technique represents a promising approach to mitigate injection attacks, also on legacy systems, mainly because the improvement with respect to a taint analysis entirely dynamic is significant. The current static engine could be significantly improved further reducing the number of false positives and the number of program statements to be monitored dynamically.

Chapters 6 and 7: Remediation Procedures
Unfortunately sometimes malicious activities detection could fail: strategies implemented seems to be always one step behind the attacker, however, by automating data collections and further analysis we could be able to compete against the attackers and learn more about their activities. Typically to defend against malicious programs, users rely on commercial malware detectors, which try to detect and prevent threats before the system is damaged. Unfortunately, in many cases detection and prevention are not possible, and in such a situation, post-infection remediation remains the only solution to get rid of a malware and of the damages
it may have caused to the system, other than reinstalling the entire system. We expect that remediation procedures should be able to remove from infected systems all the resources (e.g., files, processes and registry keys) created during the infection, but also to recover the original state of the resources that have been altered by the malware sample. In the second half part of this dissertation we proposed our methodology to evaluate the remediation capabilities of commercial malware detectors. We implemented the proposed methodology in a framework prototype and evaluated six of the most rated malware detectors on the marked. Experimental results showed that none of the tested commercial solutions turned out to be complete. We proposed and developed also our own system able to automatically generate malware remediation procedures based on malwares dynamic behavioral analysis. The results of the evaluation witness the effectiveness of our generator: our remediation procedures outperformed in completeness and soundness those available in top-rated commercial anti-malware software. Dynamic behavioral analysis could be further enhanced in order to handle many details we currently miss. An interesting enhancement in our framework could also be the development of a remediation procedure generator for multiple variants of the same malware family; even if whole procedures soundness would be degraded.


