UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

Defending Against Cybercrime: Advances in the Detection of Malicious Servers and the Analysis of Client-Side Vulnerabilities

PH.D THESIS

Antonio Nappa

Departamentamento de Lenguajes y Sistemas Informáticos e Ingenieria de Software

Escuela Técnica Superior de Ingenieros Informáticos

Defending Against Cybercrime: Advances in the Detection of Malicious Servers and the Analysis of Client-Side Vulnerabilities

Submitted in partial fulfillment of the requirements for the degree of:
Doctor en Informática

Author: **Antonio Nappa**

Advisor
Dr. Juan Caballero

February 2016

Jury:

Somesh Jha, *Professor of Computer Sciences - University of Wisconsin-Madison*
Lorenzo Cavallaro, *Senior Lecturer of Computer Sciences - Royal Holloway University of London*
Juan Manuel Estévez Tapiador, *Profesor Titular de Universidad - Universidad Carlos III de Madrid*
Victor A. Villagrá, *Profesor Titular de Universidad - Universidad Politécnica de Madrid*
Boris Köpf, *Assistant Research Professor - IMDEA Software Institute*
Carmela Troncoso, *Researcher - IMDEA Software Institute*
Manuel Carro, *Profesor Titular de Universidad - Universidad Politécnica de Madrid*

Resumen de la tesis

Esta tesis se centra en el análisis de dos aspectos complementarios de la ciberdelincuencia (es decir, el crimen perpetrado a través de la red para ganar dinero). Estos dos aspectos son las máquinas infectadas utilizadas para obtener beneficios económicos de la delincuencia a través de diferentes acciones (como por ejemplo, clickfraud, DDoS, correo no deseado) y la infraestructura de servidores utilizados para gestionar estas máquinas (por ejemplo, C & C, servidores explotadores, servidores de monetización, redirectores).

En la primera parte se investiga la exposición a las amenazas de los ordenadores victimas. Para realizar este análisis hemos utilizado los metadatos contenidos en WINE-BR conjunto de datos de Symantec. Este conjunto de datos contiene metadatos de instalación de ficheros ejecutables (por ejemplo, hash del fichero, su editor, fecha de instalación, nombre del fichero, la versión del fichero) proveniente de 8,4 millones de usuarios de Windows. Hemos asociado estos metadatos con las vulnerabilidades en el National Vulnerability Database (NVD) y en el Opens Sourced Vulnerability Database (OSVDB) con el fin de realizar un seguimiento de la decadencia de la vulnerabilidad en el tiempo y observar la rapidez de los usuarios a remiendar sus sistemas y, por tanto, su exposición a posibles ataques.

Hemos identificado 3 factores que pueden influir en la actividad de parches de ordenadores victimas: código compartido, el tipo de usuario, exploits. Presentamos 2 nuevos ataques contra el código compartido y un análisis de cómo el conocimiento usuarios y la disponibilidad de exploit influyen en la actividad de aplicación de parches. Para las 80 vulnerabilidades en nuestra base de datos que afectan código compartido entre dos aplicaciones, el tiempo entre el parche libera en las diferentes aplicaciones es hasta 118 das (con una mediana de 11 das)

En la segunda parte se proponen nuevas técnicas de sondeo activos para detectar y analizar las infraestructuras de servidores maliciosos. Aprovechamos técnicas de sondaje activo, para detectar servidores maliciosos en el internet.

Empezamos con el análisis y la detección de operaciones de servidores explotadores. Como una operación identificamos los servidores que son controlados por las mismas personas y, posiblemente, participan en la misma campaa de infección. Hemos analizado un total de 500 servidores explotadores durante un perodo de 1 ao, donde 2/3 de las operaciones tenian un nico servidor y 1/2 por varios servidores.

Hemos desarrollado la técnica para detectar servidores explotadores a diferentes tipologas de servidores, (por ejemplo, C & C, servidores de monetización, redirectores) y hemos logrado escala de Internet de sondeo para las distintas categoras de servidores maliciosos. Estas nuevas técnicas se han incorporado en una nueva herramienta llamada CyberProbe. Para detectar estos servidores hemos desarrollado una novedosa técnica llamada Adversarial Fingerprint Generation, que es una metodologa para generar un modelo nico de solicitud-respuesta para identificar la familia de servidores (es decir, el tipo y la operación que el servidor apartenece). A partir de una fichero de malware y un servidor activo de una determinada familia, CyberProbe puede generar un fingerprint válido para detectar todos los servidores vivos de esa familia. Hemos realizado 11 exploraciones en todo el Internet detectando 151 servidores maliciosos, de estos 151 servidores 75% son desconocidos a bases de datos publicas de servidores maliciosos.

Otra cuestión que se plantea mientras se hace la detección de servidores maliciosos es que algunos de estos servidores podran estar ocultos detrás de un proxy inverso silente. Para identificar la prevalencia de esta configuración de red y mejorar el capacidades de CyberProbe hemos desarrollado RevProbe una nueva herramienta a traves del aprovechamiento de leakages en la configuración de la Web proxies inversa puede detectar proxies inversos. RevProbe identifica que el 16% de direcciones IP maliciosas activas analizadas corresponden a proxies inversos, que el 92% de ellos son silenciosos en comparación con 55% para los proxies inversos benignos, y que son utilizado principalmente para equilibrio de carga a través de mltiples servidores.

Abstract of the dissertation

In this dissertation we investigate two fundamental aspects of cybercrime: the infection of machines used to monetize the crime and the malicious server infrastructures that are used to manage the infected machines.

In the first part of this dissertation, we analyze how fast software vendors apply patches to secure client applications, identifying shared code as an important factor in patch deployment. Shared code is code present in multiple programs. When a vulnerability affects shared code the usual linear vulnerability life cycle is not anymore effective to describe how the patch deployment takes place. In this work we show which are the consequences of shared code vulnerabilities and we demonstrate two novel attacks that can be used to exploit this condition.

In the second part of this dissertation we analyze malicious server infrastructures, our contributions are: a technique to cluster exploit server operations, a tool named CyberProbe to perform large scale detection of different malicious servers categories, and RevProbe a tool that detects silent reverse proxies.

We start by identifying exploit server operations, that are, exploit servers managed by the same people. We investigate a total of 500 exploit servers over a period of more 13 months. We have collected malware from these servers and all the metadata related to the communication with the servers. Thanks to this metadata we have extracted different features to group together servers managed by the same entity (i.e., exploit server operation), we have discovered that 2/3 of the operations have a single server while 1/3 have multiple servers.

Next, we present CyberProbe a tool that detects different malicious server types through a novel technique called *adversarial fingerprint generation* (AFG). The idea behind CyberProbe's AFG is to run some piece of malware and observe its network communication towards malicious servers. Then it replays this communication to the malicious server and outputs a fingerprint (i.e. a port selection function, a probe generation function and a signature generation function).

Once the fingerprint is generated CyberProbe scans the Internet with the fingerprint and finds all the servers of a given family. We have performed a total of 11 Internet wide scans finding 151 new servers starting with 15 seed servers. This gives to CyberProbe a 10 times amplification factor. Moreover we have compared CyberProbe with existing blacklists on the internet finding that only 40% of the server detected by CyberProbe were listed.

To enhance the capabilities of CyberProbe we have developed RevProbe, a reverse proxy detection tool that can be integrated with CyberProbe to allow precise detection of silent reverse proxies used to hide malicious servers. RevProbe leverages leakage based detection techniques to detect if a malicious server is hidden behind a silent reverse proxy and the infrastructure of servers behind it. At the core of RevProbe is the analysis of differences in the traffic by interacting with a remote server.

To my parents, the best people that I ever met in my life

Acknowledgments

The first thanks goes to Elo, for her infinite patience and love that you demonstrate. I want also to thank my sister Paola, to help me in my moments of crisis. And I want to thank myself for never giving up. I also want to thank the following people that helped with their positive attitudes and jokes to make the journey of PhD more pleasant: Goran Doychev, Srdjan Matic, Miguel Ambrona, German Delbianco, Julian Samborski-Forlese, Alejandro Sanchez, Miriam Garcia, Marcos Sebastian, Luca Nizzardo, Platon Kotzias and Aristide Fattori. I want also to thank all my coauthors for their efforts. The last thanks goes to my advisor Juan Caballero, to be a guidance, a point of reference, a source of wisdom and inspiration.

Contents

List of Figures

List of Tables

List of Publications

This thesis comprises five papers, the first four have been published in peer-reviewed international academic conferences and journals. The last one is currently under submission. The following list summarizes the aforementioned publications:

- Antonio Nappa, Zubair Rafique, and Juan Caballero
  Driving in the cloud: An analysis of drive-by download operations and abuse reporting
  In Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), Berlin July 2013.

- Antonio Nappa, Zubair Rafique, and Juan Caballero
  The MALICIA Dataset: Identification and Analysis of Drive-by Download Operations
  International Journal of Information Security vol. 14 (IJIS), 2014.

- Antonio Nappa, Zhaoyan Xu, Juan Caballero, and Guofei Gu
  Cyberprobe: Towards internet-scale active detection of malicious servers
  In Proceedings of Network and Distributed System Security Symposium (NDSS), San Diego February 2014.

- Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras
  The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching

In Proceedings of 36th IEEE Symposium on Security and Privacy (S&P), San Jose May 2015.

- Antonio Nappa, Rana Faisal Munir, Irfan Khan Tanoli, Christian Kreibich, and Juan Caballero
  RevProbe: Detecting Silent Reverse Proxies in Malicious Server Infrastructures
  Under submission

1 Introduction

Recent years have seen the rise of cybercrime [1], i.e., crime realized through computers and networks whose goal is to make money. The proliferation of cybercrime is likely due to the increasing profitability that derives from it [2, 3].

There exist many types of cybercrime such as clickfraud, distributed denial-of-service (DDoS), spam, targeted attacks, information theft, and identity theft. Cybercrime operations often comprise 3 main components: the group of people that organize and manage the crime, the infected victim machines used to monetize the operation through different malicious actions (e.g., perform clickfraud, launch DDoS, send spam), and the servers used to manage those infected machines. This dissertation focuses on two elements of cybercrime operations: infected victim machines and malicious server infrastructures.

There are two main methods to infect victim machines used for monetization: social engineering and exploitation of software vulnerabilities. In social engineering, the goal is to convince the user to install a piece of malware on his machine. For example, an attacker may rename a piece of malware as a benign software (e.g., photoshop.exe) and make it available through peer-to-peer networks like Bittorrent [4]. One of the most important countermeasures against social engineering is user education [5].

In this dissertation we focus on exploitation of software vulnerabilities. The most effective countermeasure against vulnerabilities are software patches. However, there are factors that limit the effectiveness of software patches. For example, the different policies to manage patch deployment (e.g., semi-automatic or manual), the presence of multiple versions of the same program in the system (i.e., shared code), user behavior, and user education.

Once a vulnerability is exploited, depending of the privilege the exploit obtains, the attacker can execute arbitrary code on the victim machine, which enables the installation of malware. The most frequently used methods to install malware on remote clients through vulnerability exploitation are called *drive-by downloads* [6]. In a drive-by download a user follows a web link that redirects

its browser through potentially multiple websites and eventually ends on a website that exploits a vulnerability on the user's browser and drops malware on the user's machine. Analyzing victim machines patch deployment activity, is fundamental to understand the exposure of these machines to threats, how fast their users are reacting to potential risks, and which factor are slowing the patching process.

When the infection process is completed, the infected hosts need to be managed remotely, which is typically achieved through servers. These servers communicate, host, and store vital information for the cybercrime operation. Different types of servers are used such as command and control (C&C) servers to control networks of infected computers, monetization servers used by ransomware to collect payments from the victims [7], and redirectors used for anonymity. Detecting malicious server infrastructures is crucial to disrupt and take-down cybercrime operations, and helps to identify the people behind a cybercrime operation.

In this thesis, we first analyze and propose novel solutions to the problems that influence the patch deployment process (e.g., user education, shared code) on victim machines. Second, we propose novel approaches to detect and analyze malicious server infrastructures in the wild.

The rest of this Chapter is organized as follows. Section 1.1 introduces vulnerability patching, the challenges that arise during the analysis of this phenomenon, and the approach that we have used to address those challenges. Section 1.2 introduces malicious server infrastructures, shows the challenges to overcome for detecting and analyzing these servers, and illustrates the approach that we have adopted to overcome those challenges, while Section 1.3 details our contributions.

1.1 Victim Infection

Many cybercrime operations monetize infected machines by, among others, sending spam, launching DDoS attacks, and performing clickfraud. To control the infected machines miscreants use malware. To install malware on victim machines the attacker has to either convince the user to install it, or to exploit a vulnerability on the user's machine, for example in the browser or its plugins. However, vulnerabilities do not persist in a system forever. If the user applies the patch from the developer the vulnerability gets fixed and the software cannot be exploited anymore.

In Figure 1.1 we depict the standard linear vulnerability life cycle model, where the horizontal line represents time. On the time line different events are marked. The starting point t_v is the point when an application is released containing a vulnerability. Then, at time t_0 the vulnerability is disclosed and at t_p the patching starts. Exploits might exist before the disclosure time (zero-day) or might be created after disclosure, for example from the patch [8].

Different defense mechanisms against exploitation of vulnerabilities are avail-

able. For example intrusion detection systems (IDS) that monitor the network traffic looking for suspicious payloads [9], and defenses against exploitation implemented in the operating systems like stack canaries, ASLR, PIE [10, 11, 12]. Yet, the most effective countermeasure against vulnerability exploitation are software patches, because they eradicate the root of the problem, that is the vulnerability. However, different factors can slow the patching process, for example manual or semi-automatic patching, user behavior and education, shared code, and the presence of exploits in the wild. These factors affect the speed of patch deployment and hence the immunity to exploitation.

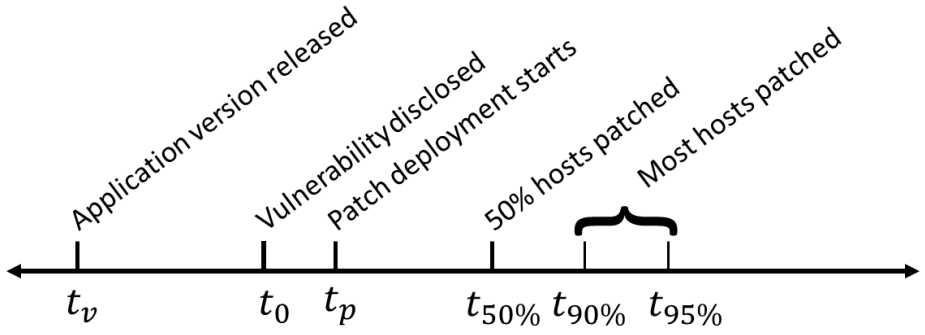


Figure 1.1: Vulnerability life cycle

1.1.1 Problem Statement

In this section we illustrate three fundamental factors that affect vulnerability patching: shared code, type of user, and exploit availability.

Shared Code. The vulnerability model of Figure 1.1 is widely used in previous literature and is a linear model [13, 14, 15, 16, 17], where different events happen sequentially (e.g., application release, vulnerability disclosure, vulnerability patching), but not always in the same order (e.g., exploit might exist before or after patch release).

This model has a fundamental limitation: it does not take into account that multiple versions of the same software may coexist on a system or that the same software libraries (e.g., .dll files on Windows) may be shipped with different programs. These libraries may contain a vulnerability and be managed by different patching programs, and hence are subject to possibly different patching policies.

This introduces the problem of shared code, where the vulnerability life cycle gets more complex. In Figure 1.2 we show the vulnerability life cycle when a user has an old non-updated version of a software (Adobe Reader 5) and another updated version (Adobe Reader 10) that can coexist on a system and share a

vulnerability. It is intuitive to see that the time of starting of the patching t_p is in different positions on the two time lines.

This difference in the patching policies can be exploited by an attacker that can try to force the execution of the most outdated software to reproduce some malicious content and exploit the system of the victim. In this thesis we present two new avenues to exploit shared code.

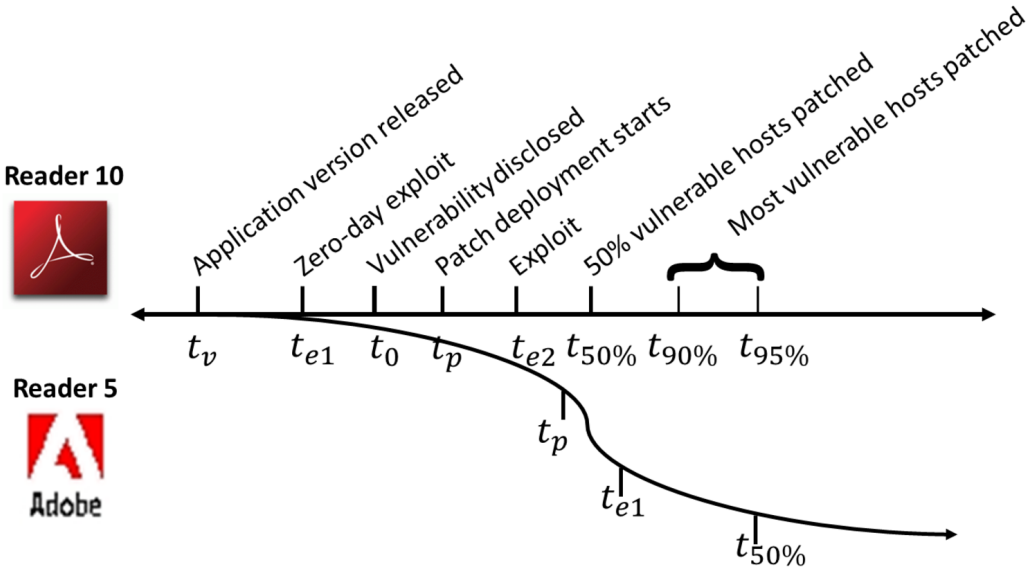


Figure 1.2: Vulnerability life cycle in case of software clones

Type of User. Another fundamental aspect of securing software against vulnerabilities is user education, because different users have different competences and skills. Therefore they have different behaviors with respect to patching. Vendors have made significant efforts to speed-up and automate patching. An example of this effort is the automatic patching policy adopted by Google Chrome since its first release, so that there is no human intervention that can slow down the patching process. However, there are some situations where automatic patching may be disabled. For example it is not uncommon that in large corporate networks, system administrators want to have full control of the software versions of their applications. In this dissertation we will investigate how different type of users adopt different patching behaviors.

Exploits. The presence of exploits in the wild should speed up the patching process because in the very moment an exploit becomes available, all the vulnerable machines are potentially exploitable. For this reason, understanding whether the presence of exploits in the wild hurries the patching process is crucial to design new countermeasures and policies against vulnerability exploitation. In

this thesis we will analyze the reaction of clients and software vendors in case an exploit is available in the wild.

1.1.2 Approach

Previous work has focused mostly on server-side vulnerabilities [18, 19], which can be analyzed by periodically scanning the Internet to measure the vulnerable population. Also, a limited subset of client-side application (i.e., web browsers) have been analyzed while they visit remote websites [20].

In this dissertation we focus on client-side vulnerabilities, i.e., desktop software that is used on client machines, for example document readers like Adobe Reader, document editors like Microsoft Word, browsers like Opera, Firefox, Safari, Internet Explorer, and multimedia players like Adobe Flash and Quicktime. This client-side programs, excluding browsers, do not expose on the network so in order to study them is necessary to have a presence on client machines. For this reason, previous approaches cannot be used.

First, it is not possible to scan the Internet to measure the vulnerable population, because client applications, excluding browsers, usually do not expose any behavior on the network. Second, it is possible that users may install an application and seldom use it. Third, for measuring browser patching activity through visits to remote websites has several limitations: (I) browsers behind Network Address Translation (NAT), where multiple clients appear on the Internet with a single public IP address are not properly measured; (II) users that do not visit the monitoring website will not be part of the measurement. To overcome these limitations we have developed a new approach to study large scale deployment of client software patches and the issues that can affect this process.

Our approach leverages data from different sources: Symantec's WINE-BR and WINE-AV [21] datasets, National Vulnerability Database (NVD) [22], Open Sourced Vulnerability Database (OSVDB) [23], Exploit Database (EDB) [24], and VirusTotal [25].

The WINE-BR dataset comprises file metadata (e.g., filename, hash, date, publisher, version) of 8.4 Million hosts with Symantec software installed. This dataset provides meaningful information (i.e., dates, versions, filenames) to help tracking client-side software patching activity. We have enriched WINE-BR's capabilities with metadata coming from VirusTotal such the content of fields in the header of the binary executable.

We use NVD and OSVDB vulnerability databases along with WINE-BR and VirusTotal datasets to find for a given vulnerability, identified by its CVE (e.g., CVE-2010-0221), which are the vulnerable and the non-vulnerable versions of a given program.

The information available in public vulnerability databases might be not perfect, because usually the vulnerabilities are reported by volunteers so the versions that are reported as vulnerable might not be vulnerable or vice versa.

Also, different vendors have different policies for assigning program versions, using program lines, and issue security advisories that may mislead the classification of a program version to be vulnerable or not.

To address these challenges we have developed a generic approach to map files in a host to vulnerable and non-vulnerable program versions. We aggregate the vulnerabilities in these databases into clusters that are patched by the same program version.

Finally, We track the global decay of the vulnerable host population for each vulnerability cluster, as software updates are deployed, thanks to a statistical technique called *survival analysis*, a technique often used in medical sciences to track how a population of patients decays when a new drug is administered.

This approach allows us to estimate for each vulnerability cluster the delay to issue a patch, the rate of patching, and the vulnerable population in WINE. This approach is helpful also to investigate shared-code (e.g., two programs from the same vendor that share a vulnerability in a .dll shipped with both) and to analyze the attack windows that open when two programs share a vulnerability but follow distinct patching mechanisms.

By checking which applications the WINE users have installed on their systems we classify the owners of the machines into different categories (i.e., software professionals, developers, security analysts) and then measure the patch deployment median time for each different category. This measurement is fundamental to understand whether different type of users have different patching behaviors.

Furthermore, we leverage exploit metadata in the WINE-AV dataset and the Exploit Database to estimate the dates when exploits become available and to determine the percentage of the host population that remains vulnerable upon exploit releases.

1.2 Malicious Infrastructures

As previously stated, cybercrime operations need three fundamental components to work: people, victim machines, and a server infrastructure to manage the victim machines. In this section we describe the most common machine infrastructures used by miscreants for their crimes, and which are the most common types of server used in these infrastructures. Then we show which are the problems that arise in order to detect these kind of infrastructures and the approach that we have adopted to solve them.

To deploy and manage a cybercrime operation miscreants need an infrastructure of different server types that play different roles in the operation. For example, exploit servers are used to infect victim machines, command and control servers manage the infected machines, monetization servers collect ransoms, and reverse proxies hide the location of the final servers. There exist different infrastructures and server types that are used either in benign or in malicious contexts.

In benign contexts the choice of the model might be oriented to performance and efficiency, in malicious contexts the choice is oriented to achieve stealthiness and resiliency.

Client/Server Model. The most simple and widespread infrastructure is the client/server model, where a remote server (C&C) sends commands to the infected machines. This model is also used in Pay-Per-Install infrastructures [26], where the remote server collects information of successful installation of the malware. In this configuration one or more remote servers are used to control the *botnet*. A network of infected machines used for clickfraud, DDoS, spam campaigns and more. So once the IP address of the server is known is very easy to take it down.

Peer-to-peer Model. In the late years also peer-to-peer infrastructures have emerged to be used by miscreants [27]. In this infrastructure there is no central server, and hence no single point of failure. The manager of the infrastructure (also known as botmaster) uses the malware sample (that has server functionality) to control the infrastructure and send commands to bots.

The P2P model offers better resiliency, due to its distributed nature. While a classical client/server architecture offers a more centralized control of the infected machines, it is less resilient to take-down efforts. This because in P2P networks, peers are equipotent participants in the application; they are both client and servers at the same time. Hence, in malicious infrastructures every peer can be a potential malicious server. While in client/server architectures the behavior of the server is completely different from the client.

1.2.1 Malicious Server Types

In this Subsection we describe some important types of of malicious servers used by miscreants to perpetrate their crimes.

1.2.1.1 Exploit Servers

Exploit servers have emerged to be the major distribution vector of malware [28]. These kind of web servers distribute malware by exploiting the client that is visiting them. The exploitation is fully automated by a software that is called exploit kit, which is sold under a license like a legitimate product. There are different roles in the exploit kit ecosystem: *malware owner, exploit kit developer, exploit server owner, exploit server manager*. The malware owner is usually the person that buys the exploit kit license because he wants to install his malicious software on victim machines. The exploit kit developer offers a software kit including a set of exploits for different platforms (i.e., combination of browser, browser plugins, and OS), web pages to exploit visitors and drop files on their hosts, a database to

store all information, and an administration panel to configure the functionality and provide installation statistics.

There are basically three elements needed to make an exploit server successfully work:

1. Exploit Kit
2. Traffic from potential victims.
3. Exploit server hosting.

The Exploit-kit software has three different marketing schemas:

- Host-it-Yourself (HIY).
- Exploitation-as-a-Service(EaaS).
- Pay-Per-Install (PPI).

The drive-by downloads ecosystem has been by now commoditized [28], there are miscreants that offer exploitation software as a service to other miscreants that want to exploit victim machines to install their malware. In the following paragraphs we show which are the different services that a malware owner can purchase to install his malware on victim machines.

Host-it-Yourself. In this model the malware owner pays only the license of the exploitation software. He has to provide the traffic, the malware to install, and the server where to host the exploit kit.

Exploitation-as-a-Service. In this model the malware owner will rent one or more exploit servers from the developers of the exploit kit, he has to provide the traffic and the malware to install, while the the hosting infrastructure will be provided by the developers of the exploit kit generally on bullet-proof hosting [29].

Pay-Per-Install. In this model the malware owner has to provide only the malware that he wants to be installed on the victim machines, the rest (traffic, hosting infrastructure) will be provided by the PPI service upon payment. In this marketing schema there are also affiliates, namely other miscreants that will monetize malware installations by reporting a successful malware installation to the PPI service.

1.2.1.2 Command and Control Servers

C&C servers are one of the most (in)famous categories of malicious servers, since they are used to manage infected hosts. These servers communicate with the infected machines through different C&C protocols, for example they can post the commands on a web blog or inside a social network through HTTP [30] or use a proprietary communication protocol [31]. Usually a botnet has several C&C servers that are pointed by different domains. When law-enforcement agencies want to take-down a botnet they need to discover and seize all the C&C servers, otherwise the botnet will survive.

1.2.1.3 Reverse Proxies

Reverse proxies are a particular category of web servers used to hide the final, more valuable, destination web server. In the architecture of a malicious operation there can be multiple reverse proxies hiding one or more final servers: in this way if one of the reverse proxies is taken down there is little damage to the underlying malicious infrastructure, since the proxy can be easily replaced without affecting the final servers. Understanding whether a final server is hidden behind a reverse proxy is a key factor for a successful take-down.

1.2.1.4 Other Server Types

There are also other server categories used in malicious operations. For example, monetization servers collect payments for fake anti-virus programs or ransomware, and monitoring servers collect statistics about how many exploitations have been successful and their geographical distribution.

1.2.2 Problem Statement

Current techniques to detect malicious server infrastructures suffer several limitations. There are two different approaches to detect and identify malicious infrastructures: passive and active. Passive techniques, include for example honeypots (i.e., vulnerable machines listening on the network) [32], spamtraps [33], and intrusions detection systems [34, 35]. These techniques are usually slow and incomplete because, for example, it takes time to get a honeypot infected, which will eventually contact a malicious server that can then be reported and taken-down. Active techniques, on the other hand, include infiltration into botnets [36], and Pay-Per-Install (PPI) services [26]. Another example of active techniques are Google's Safebrowsing and Microsoft's Forefront technologies that consist of honeyclients (i.e., virtual machines running a full fledged browser) farms that actively crawl web servers to see if they are distributing malware through drive-by downloads. These techniques proactively interact with the cybercrime infrastructure

and try to infer information about it. Active techniques are usually, with respect to passive, more complete and faster, but still they have limitations.

For example, Google Safebrowsing [37] crawls the web detecting exploit servers, but it does not group together the servers that are part of the same criminal operation. The identification of server operations is fundamental to allow precise take-down efforts and may help to identify the people behind the operation.

Previous work has been focused to detect a particular server type, for example the recent analysis and take-over of the Dorkbot botnet [38] focuses on C&C servers and the study of the exploit-kit ecosystem [28] focuses on exploit servers. The limitation of these approaches is that the detection and analysis mechanism are tailored on a specific server type.

Hence detecting and enumerating all the servers of an operation is another key feature that detection tools must have and current approaches lack this capability.

For example, a ransomware operation may have 3 C&C servers, 3 monetization servers used for managing the transactions with the infected users, and 2 exploit server used to install the malware on victim machines. Detecting all 8 servers is one of the most effective ways to disrupt the entire operation. Because if at least one exploit server, one C&C server and one monetization server remain alive, the operation will still survive. Detecting and enumerating all the servers of an operation can also help to find the people behind the operation, thanks to the evidence that can be found on these servers, like logs and hosting provider registration information. Identifying the people behind an operation is an even better way to disrupt the operation. Another limitation of current detection approaches is the capability to understand if the detected server is only a redirector (i.e., silent reverse proxy) that hides the final malicious server.

In this dissertation we develop novel tools and techniques to tackle the problem of detection and analysis of malicious server infrastructures generally. In other words we design tools and techniques to potentially detect any kind of malicious server and any kind of operation starting with a malware sample and an alive server of the operation. In addition, we provide a solution to the problem of silent reverse proxies detection to improve the accuracy of current detection methodology.

1.2.3 Approach

In the second part of dissertation we focus on active detection techniques of malicious server infrastructures. First, we have focused our researches on the detection of exploit server operations. Then, we have developed CyberProbe, a tool to detect different server types (e.g., C&C, monetization, and P2P bots) that improves coverage to Internet scale. Lastly, we have realized RevProbe, a tool that detects silent reverse proxies, which may hide malicious servers behind. RevProbe improves the accuracy of CyberProbe to allow precise take-down efforts.

Exploit Server Operations Detection. We have developed an active probing methodology to detect exploit server operations in the wild. We have built an infrastructure that collects malicious URLs from external feeds leading to exploit servers that drop malware through drive-by downloads. Once we have the URLs, we visit them with specialized milkers and honeyclients. Specialized milkers are HTTP clients tailored for a specific exploit server family that minimize to the bare minimum the dialog with the server to get the malware samples. The second are instrumented virtual machines with a full fledged browser that are directed to the malicious URLs to get the malware samples. Besides the malware samples, our milkers also collect the metadata associated with the exploit servers they visit (e.g., domains, URLs) and save this data in a database for further analysis.

We have collected more than 11,000 unique malware samplesr. To classify these samples we use dynamic and static approaches. We run the malware in a sandbox and capture a screenshot of its execution and the network traffic. We also collect the icons embedded in the executable file. We use perceptual hashing techniques [39] to compare and cluster together malwares with similar screenshots and icons. For the network traffic we use FIRMA [40] an external tool that automatically clusters similar malware together based on its network behavior.

With the information collected during the milking phase and the classification of the malware samples we cluster together exploit servers that belong to the same operation (i.e., controlled by the same individuals). Our approach leverages the fact that exploit servers from the same operation share the configuration of the exploit kit software. By checking different features of the configuration (e.g., domain, urls, distributed malware).

Internet-Scale Malicious Server Detection. After investigating exploit server operations we have expanded our detection capabilities developing a new approach to detect different server types on a large scale. We have developed a tool named CyberProbe that uses a novel active probing approach to detect and enumerate different server types and operations. At the core of CyberProbe is adversarial fingerprint generation (AFG). The difference with previous approaches on network fingerprinting [41], is that in our case we do not control the remote server and hence we cannot send as many probes as we want. AFG is technique that by interacting with one server of an operation generates a fingerprint (i.e. a unique request and response pattern) that will be used for an Internet wide scan to find all the instances of that server family. As a server family we identify the operation which the server belongs plus the type of server. The intuition behind CyberProbe's AFG is to reply a previously seen communication done by a malware sample towards a remote server (i.e., seed server) that needs to be alive at the moment of the fingerprint generation. We extract the relevant communication that the malware has done with one or more remote servers, that are not benign, and then generalize a unique request-response pattern that can

identify all the remote servers of that family.

We extract the request-response-pairs (RRPs) from the communications and automatically identify if they have to be discarded because they are benign, or we have to generalize and replay them to the remote server to generate a fingerprint.

A fingerprint comprises: (I) a port selection function that given an IP address returns the port where to send the probes, (II) a probe construction function that given an IP address returns the request that we will send to remote servers, and (III) a classification function, a function that will classify the responses to the probe as benign or part of the fingerprinted operation.

CyberProbe is able to detect different server types, including P2P, and to work with different transport (e.g., TCP and UDP) and application protocols (e.g., HTTP and proprietary protocols).

We have developed different scanners for probing remote servers with the generated fingerprints. We have a TCP horizontal scanner that scans the target network (e.g., Internet) to see if the selected port is open. Then with the result of the horizontal scan we feed the AppTCP scanner that sends the probes to the servers that have the target port open. We have also developed a UDP scanner that sends probes through UDP and collects the answers.

We have adopted two different approaches for our scanning. First, we have scanned the local ranges where the seed servers are hosted. Then we have also performed Internet-wide scans. Our aim during our scans was to achieve coverage in a reasonable amount of time without raising many complaints by the receiving networks. Moreover we have manually vetted our fingerprints not to contain any harmful information like exploits.

Silent Reverse Proxy Detection. While investigating malicious infrastructure we have faced the problem of reverse proxies that are used to hide the final destination servers that contain evidence of the cybercrime. To discover if a probed server is a reverse proxy we have developed a new tool called RevProbe. By leveraging information leakages RevProbe enhances the capabilities of CyberProbe allowing the detection of reverse proxies used to protect malicious infrastructures. We have adopted an active probing black-box approach, where we do not have any control on the target server. There exists two general approaches to identify proxies through active probing: *timing-based* and *discrepancy-based*. Timing-based approaches leverage the property that proxies introduce an additional hop in the communication and consequently they introduce more delay. Discrepancy-based approaches on the other hand focus on traffic changes introduced by the proxy.

We use discrepancy-based approach, because timing based approaches cannot identify multiple servers hidden behind a web load balancer (WLB) and also because previous work has shown that timing based approaches are not effective on the Internet, due to the fact the delays between two servers within the same network are too small to be significant [42]. Thus, discrepancy based approaches

are more reliable to detect reverse proxies and to reconstruct the infrastructure that lays behind them.

However, discrepancy based approaches may miss to detect a reverse proxy if it does not introduce any change in the traffic, for this reason we have included in RevProbe other techniques that help to detect difficult configuration such as reverse proxy running the same software as the final server behind.

1.3 Thesis Contributions

The main contributions of this thesis are divided in three parts, the first about analyzing the vulnerability life cycle, the second about novel techniques for active detection of malicious servers infrastructures, and the last one is the Malicia dataset, collected over a period of more than one year while analyzing malicious server infrastructures.

The contributions are summarized as the following:

1. A Study of the Impact of Shared Code on Vulnerability Patching (1.3.1)
2. Active Detection of Malicious Server Infrastructures and the Malicia dataset (1.3.2)

1.3.1 A Study of the Impact of Shared Code on Vulnerability Patching

As mentioned previously software vulnerabilities are very important in the building process of a cybercrime operation, because they are the entry point to install malware into remote clients. In Chapter 3 we have developed a general approach to merge information about vulnerabilities from different data sources and measure vulnerability life cycle [43]. With respect to vulnerability life cycle analysis our work makes the following contributions:

1. We conduct a systematic analysis of the patching process of 1,593 vulnerabilities in 10 client-side applications, spanning versions released on a 5-year period.
2. We demonstrate two novel attacks that enable exploitation by invoking old versions of applications that are used infrequently, but remain installed.
3. We measure the patching delay and several patch deployment milestones for each vulnerability.
4. We propose a novel approach to map files on end-hosts to vulnerable and patched program versions.

5. Using these techniques, we quantify the race between exploit creators and the patch deployment, and we analyze the threats presented by multiple installations and shared code for patch deployment.

6. We identify several errors in the existing vulnerability databases, and we release a cleaned dataset at http://clean-nvd.com/.

1.3.2 Active Detection of Malicious Server Infrastructures

We have contributed to different aspects of detection of malicious infrastructures, our contributions can be summarized as the following:

1. In Chapter 4, we develop a methodology to cluster the servers into the cybercrime operations they belong using the information provided by the analysis of the malware they distribute and the configuration of the exploit kit software they host [44]. We investigate 500 exploit servers in the wild for a period of 11 months and the polymorphism and the prevalence of the exploits used by these servers [45].

2. In Chapter 5, we have developed a general framework to fingerprint and identify any kind of malicious server in the Internet. Our technique is general, easy to deploy, and cheap. It represents a significant advance to detect and take down cybercrime operations. Our approach interacts with alive servers from an operation using a novel technique called *Adversarial Fingerprint Generation*. This technique replays the communication of the malware with the malicious server and identifies a distinctive trace in the response that is able to fingerprint all the servers of that type for that particular operation. We have then developed different scanners for different protocols and performed 24 localized and Internet-wide scans to demonstrate the effectiveness of our approach.

3. In Chapter 6, we have faced one issue of CyberProbe, the usage of silent reverse proxies to hide the final servers. For take-down efforts it is necessary to exactly identify which is the server that belongs to the operation. Miscreants, to protect their infrastructure add one or more silent reverse proxies in front of the final servers. In this way law-enforcement agencies will be only able to find the reverse proxies that do not contain evidence about the cybercrime.

   For this reason, we have developed RevProbe a system that, by using information leakage techniques, is able to detect reverse proxies and also effectively identify the infrastructure that lays behind them. We have tested RevProbe on 36 controlled scenarios showing that our tool outperforms current solutions, moreover we have measured the prevalence of reverse proxies in malicious and benign websites on the Internet.

The Malicia Dataset. During the analysis of the malicious drive-by download operations we have collected 11,688 malware over a period of 11 months from 500 different exploit servers. These samples have been analyzed (statically and dynamically) and classified, all the derived data has been collected and organized into a database. In addition to the classification data the DB also contains the details of the different operations to which the exploit servers belong. Giving also a picture of these kind of operations.

We make the Malicia dataset [46, 45] available to the community. At the time of the writing of this thesis the dataset has been released to more than 65 international institutions.

2

Related Work

It was year 1996 when the article by Aleph1 "Smashing The Stack For Fun And Profit" appeared on Phrack magazine, that was the forerunner for a growing economy where exploits are a commodity that can be purchased [28] and where people get rewards for installing malicious software on victim's machine [26]. Software vulnerabilities are a key point in cybercrime because when exploited they allow to take control of a remote machine. The fact that most of the computers in the world run the same software gives to miscreants potential to take control literally of an army of machines for their nefarious actions. Of course to control all these machines an infrastructure is needed.

Vulnerabilities and malicious infrastructures are two fundamentals aspect of cybercrime, these aspects have been extensively studied in literature, in this Section we show related work divided into these categories.

2.1 Vulnerabilty Lifecycle Analysis

Several researchers [13, 15, 16] have proposed vulnerability lifecycle models, without exploring the patch deployment phase in as much detail as we do. Prior work on manual patch deployment has showed that user-initiated patches [47, 48, 49, 50] occur in bursts, leaving many hosts vulnerable after the fixing activity subsides. After the outbreak of the Code Red worm, Moore et. at [47] probed random daily samples of the host population originally infected and found a slow patching rate for the IIS vulnerability that allowed the worm to propagate, with a wave of intense patching activity two weeks later when Code Red began to spread again. Rescorla [48] studied a 2002 OpenSSL vulnerability and observed two waves of patching: one in response to the vulnerability disclosure and one after the release of the Slapper worm that exploited the vulnerability. Each fixing wave was relatively fast, with most patching activity occurring within two weeks and almost none after one month.

Rescorla [48] modeled vulnerability patching as an exponential decay process

with decay rate 0.11, which corresponds to a half-life of 6.3 days. Ramos [49] analyzed data collected by Qualys through 30 million IP scans and also reported a general pattern of exponential fixing for remotely-exploitable vulnerabilities, with a half-life of 20-30 days. However, patches released on an irregular schedule had a slower patching rate, and some do not show a decline at all. While the median time to patch (t_m) for the applications employing silent update mechanisms and for two other applications (Firefox and Thunderbird) is approximately in the same range with these results, for the rest of the applications in our study t_m exceeds 3 months.

Yilek et al. [50] collected daily scans of over 50,000 SSL/TLS Web servers, in order to analyze the reaction to a 2008 key generation vulnerability in the Debian Linux version of OpenSSL. The fixing pattern for this vulnerability had a long and flat curve, driven by the baseline rate of certificate expiration, with an accelerated patch rate in the first 30 days and with significant levels of fixing (linked to activity by certification authorities, IPSes and large Web sites) as far out as six months. 30% of hosts remained vulnerable six months after the disclosure of the vulnerability. Durumeric et al. [18] compared these results with measurement of the recent Heartbleed vulnerability in OpenSSL and showed that in this case the patching occurred faster, but that, nevertheless, more than 50% of the affected servers remained vulnerable after three months.

While the references discussed above considered manual patching mechanisms, the rate of updating is considerably higher for systems that employ automated updates. Gkantsidis et al. [51] analyzed the queries received from 300 million users of Windows Update and concluded that 90% of users are fully updated with all the previous patches (in contrast to fewer than 5%, before automated updates were turned on by default), and that, after a patch is released, 80% of users receive it within 24 hours. Dübendorfer et al. [52] analyzed the User-Agent strings recorded in HTTP requests made to Google's distributed Web servers, and reported that, within 21 days after the release of a new version of the Chrome Web browser, 97% of active browser instances are updated to the new version (in contrast to 85% for Firefox, 53% for Safari and 24% for Opera). This can be explained by the fact that Chrome employs a *silent update* mechanism, which patches vulnerabilities automatically, without user interaction, and which cannot be disabled by the user. These results cover only instances of the application that were active at the time of the analysis. In contrast, we study multiple applications, including 500 different versions of Chrome, and we analyze data collected over a period of 5 years from 8.4 million hosts, covering applications that are installed but seldom used. Our findings are significantly different; for example, 447 days are needed to patch 95% of Chrome's vulnerable host population.

Despite these improvements in software updating, many vulnerabilities remain unpatched for long periods of time. Frei et al. [53] showed that 50% of Windows users were exposed to 297 vulnerabilities in a year and that a typical Windows user must manage 14 update mechanisms (one for the operating system and

13 for the other software installed) to keep the host fully patched. Bilge et al. [17] analyzed the data in WINE to identify zero-day attacks that exploited vulnerabilities disclosed between 2008–2011, and observed that 58% of the anti-virus signatures detecting these exploits were still active in 2012.

2.2 Malicious Infrastructure Detection

Drive-by downloads. A number of works have analyzed drive-by downloads. Wang et al. [54] build honeyclients to find websites that exploit browser vulnerabilities. Moshchuk et al. [55] use honeyclients to crawl over 18 million URLs, finding that 5.9% contained drive-by downloads. Provos et al. [56] describe a number of exploitation techniques used in drive-by downloads. We identify four different aspects of content control responsible for enabling browser exploitation: advertising, thirdparty widgets, user contributed content, and web server security. They follow-up with a large-scale study on the prevalence of drive-by downloads and the redirection chain leading to them, finding that 67% of the malware distribution servers for drive-by downloads were in China [57]. Polychronakis et al. examined the network behavior of malware distributed by drive-by downloads [58]. Recently, Grier et al. [28] investigate, the emergence of exploit kits and exploitation-as-a-service in the drive-by downloads ecosystem, showing that many of the most prominent malware families propagate through such attack technique. Our work differs from prior drive-by downloads analysis in that we focus on identifying and understanding the properties of drive-by operations, rather than individual exploit servers, to show the big picture behind cybercrime. Other work proposes detection techniques for drive-by downloads [6, 59, 60] and could be incorporated into our infrastructure.

Cho et al. [61], infiltrated the MegaD spam botnet and collected evidence on its infrastructure being managed by multiple botmasters. In contrast, our work shows how to automate the identification of servers with shared management, grouping them into operations. In simultaneous work, Canali et al. [62] analyze the security of shared hosting services. Similar to their work, we also issue abuse reports to hosting providers but our focus is on VPS services, which are more adequate for hosting exploit servers.

Rogue Networks. Previous work has studied rogue networks hosting unusually large amounts of malicious activity [63, 64]. Our work also observes autonomous systems hosting unusually large numbers of exploit servers compared to their size, as well as countries more popular than expected from the size of their IP space.

Malware clustering & classification. Prior works on running malware in a controlled environment have influenced our malware execution infrastructure [65,

66, 67]. Our classification builds on a number of prior works on behavioral classification techniques [68, 58, 69, 70, 71, 72, 73, 28] and incorporates the automated clustering of malware icons using perceptual hashing. We could also incorporate techniques to reduce the dimensionality in malware clustering [74] and to evaluate malware clustering results using AV labels [75].

Screenshot clustering. Anderson et al. [72] compare screenshots of a browser visiting scam webpages using image shingling, which splits an image into tiles, hashes the tiles, and compares the percentage of identical hashes. Grier et al. [28] propose a different technique based on the mean squared deviation between their histograms. However, they do not evaluate their technique. Our perceptual hashing approach differs in that it is better fit to compare images of different sizes (e.g., icons).

Active probing. Active probing (or active network fingerprinting) has been proposed for a variety of goals. Comer and Lin first proposed active probing to identify differences between TCP implementations [76] and tools like Nmap popularized the approach to identify the OS of remote hosts [77]. It has also been used to identify the version of application-layer servers [78, 79] and for tracking specific devices based on device clock skews [80]. A variant of active probing identified web users that visit a web server by querying the browser [81].

Related to our work are active probing techniques to detect network-based malware. BotProbe [82] actively injects commands into IRC channels to identify if the target is an IRC bot or a human. PeerPress [83] uses active probing to detect P2P malware in a monitored network. Two fundamental differences with these work are that CyberProbe can detect any type of application that listens on the network, and that it focuses on probing external networks, achieving Internet scale. CyberProbe does not need to inject traffic into existing connections as BotProbe. The fingerprint generation used by CyberProbe differs from the one in PeerPress in that it leverages network traffic rather than binary analysis. This makes it possible to scale to running large quantities of malware. In independent work, Marquis-Boire et al. [84] manually generated fingerprints to identify the servers used by FinSpy, a commercial software that governments employ to spy on activists. Our work differs, among others, in that we propose a novel adversarial fingerprint generation technique that automatically generates fingerprints for a large number of malicious server families.

There has also been work on defeating OS fingerprinting. Smart et al. [85] proposed a stack fingerprinting scrubber that sits on the border of a protected network and limits the information gathered by a remote attacker by standardizing the TCP/IP communication. This work is based on the protocol scrubber proposed by Malan et al. [86]. Recently Xu et al. [87] proposed AutoProbe a work in which we have collaborated. This system leverages program analysis techniques to generate the fingerprints, the difference with CyberProbe is

in the adversarial fingerprint generation step, while CyberProbe uses traffic generated by the malware to create the fingerprint, AutoProbe analyzes the malware executable and is able to generate more general fingerprints and it works also without the presence of seeds servers.

Fingerprint/signature generation. FiG proposed to automatically generate OS and DNS fingerprints from network traffic [88]. CyberProbe follows the same high-level fingerprint generation approach as FiG, but proposes a novel adversarial fingerprint generation technique, with two fundamental differences. First, it does not randomly or manually generate candidate probes, rather it reuses previously observed requests. This greatly reduces the traffic that needs to be sent to the training servers for generating a fingerprint, and produces inconspicuous probes. Both properties are fundamental when fingerprinting malicious servers. In addition, CyberProbe uses network signatures to implement the classification function, so it does not require a specific fingerprint matching component. Furthermore, CyberProbe addresses the problem of Internet-wide scanning.

There is a wealth of prior work on automatically generating network signatures for worm detection. Honeycomb [89], Autograph [90], and EarlyBird [91] proposed signatures comprising a single contiguous token. Polygraph [92] proposed more expressive token set, token subsequence, and probabilistic Bayes signatures. Wang et al. extended PAYL [93] to generate token subsequence signatures for content common to ingress and egress traffic. Nemean [94] introduced semantics-aware signatures and Hamsa [95] generated token set signatures that can handle some noise in the input traffic pool. Beyond worms, Botzilla [96] generated signatures for the traffic produced by a malware binary run multiple times in a controlled environment, Perdisci et al. [71] clustered and generated signatures for malware with HTTP C&C protocols, and FIRMA [40] generalized the approach to handle any protocol. A fundamental difference is that these studies generate network signatures for requests sent by the malware, while CyberProbe generates them on the responses from remote servers. Our signature generation algorithm builds on the one proposed by Hamsa but handles larger amounts of noise in the input traffic. In addition, they either assume a single malware family or small amounts of noise inthe input traffic. Portokalids et al. [97] automatically generate signatures for zero-day attacks using taint tracking.

Scanning. Prior work demonstrates the use of Internet-wide scanning for security applications. Provos and Honeyman used it for identifying vulnerable SSH servers [98], Dagon et al. for finding DNS servers that provide incorrect resolutions [99], and Heninger et al. for detecting weak cryptographic keys in network devices [100]. In this work we propose another security application for active probing: identifying malicious servers. Leonard et al. [101] described how to perform Internet-wide horizontal scans with the goal of maximizing politeness.

The design of our scanning is greatly influenced by their work. Other studies are related to how to perform fast scanning. Staniford et al. described techniques that malware can use to quickly spread through scanning [102]. Netmap [103] proposed a framework for fast packet I/O in software, which enables a single core to generate 14.88 Mpps, enough to saturate a 10Gbps link. Recently, Durumeric et al. proposed Zmap [104], a fast Internet-wide scanner that can do a horizontal scan of the Internet in 45 minutes from a single host. Compared to these studies our goal is to identify malicious servers. CyberProbe could incorporate some of these techniques to speed up the scanning, but currently we cap the scan speed for good citizenship.

ISP proxies. ISPs may setup proxies at their border routers for improving performance (e.g., caching, image scaling) and modifying content. Wang et al. [105] propose that ISP proxies collaborate by forwarding requests among themselves so that cached objects are sent back directly to clients. In this work we call these ISP proxies and do not consider them reverse proxies because they do not hide the existence of the final servers. Recently, Weaver et al. [106] propose a technique to detect ISP and forward proxies by installing an application in the client host that communicates with a Web server under their control. Discrepancies between the response sent by the server and the response received by the client application indicate the presence of a proxy. In our work we focus on detecting reverse proxies where we cannot control the server the client connects to.

Reverse proxies. Tools like the http_trace.nasl [107] plugin detect explicit reverse proxies by examing the HTTP Via header, but cannot detect silent reverse proxies. One approach to detect proxies to send requests that replied by the reverse proxy (if there is one) and build fingerprints on those reponses. Tools like lbmap [108] and Htrosbif [109] use that approach. However, these tools only detect reverse proxies running specific software (e.g., HAProxy, pound, Vanquish) but cannot detect generic Web server software (e.g., Apache, nginx) running as reverse proxy. Gregoire [110] presents the HTTP traceroute tool (TLHS), which leverages the Max-Forwards HTTP header that limits the maximum number of times a request is proxied. However, TLHS works only on limited scenarios. For example, nginx as a reverse proxy ignores the Max-Forwards header and thus cannot be detected by TLHS. In his master's thesis, Weant [111] proposes a different technique to detect reverse proxies through timing analysis of TCP connnections.

Web load balancers. Some approaches focus only on detecting WLBs, a type of reverse proxies. Halberd [112] is a WLB detection tool that sends the same request a large number of times to a given IP address. Differences in some response headers (e.g., E-Tag, Date, Server) indicate a load balancer. This approach can

only detect reverse proxies that load balance and fails if the final web servers run the same program version. Curt Shaffer describes the problems that WLBs introduce for penetration testing [113] and presents two WLB detection techniques: detecting cookies introduced by the WLB in the communication to achieve persistence and examining the IP ID field. We have examined these techniques in detail and proposed several others.

Web application firewalls. WAFW00f [114] is a tool for detecting WAFs. It sends a normal request and malicious request to the same URL. Discrepancies between both responses identify the presence of a WAF. This tool fails if the WAF return different status code on malicious requests. This tool cannot differntiate between a WAF or other filtering performed directly by the final Web server.

Web server fingerprinting. A number of tools exist to fingerprint the program and version run by a remote Web server [115, 116, 117, 118, 119, 120]. Among these, some tools like Nmap [121] or ErrorMint [122] fingerprint the program version exclusively by examining explicit program version information provided by the server, e.g., in the Server headers and error pages. Other tools like HMAP [120], HTTPPrint [115], and HTTPRecon [117] use fingerprints that capture differences between how different web server software and version construct their responses. These type of fingerprints do not rely on the program version information explicitly provided by the server.

Automatic fingerprint generation. Some approches have been proposed to automatically build fingerprints such as FiG [88]. Book et al. [118] propose a technqiue based on bayesian filter to automatically generate fingerprints to identify web server software.

Part I

Analysis of Client-Side Vulnerabilities

3

The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching

3.1 Preamble

This chapter reproduces the content of the paper: "The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching" published at the IEEE Security & Privacy Symposium 2015. This paper describes the analysis of the vulnerability lifecycle of client-side application. This work is a collaboration with Symantec Corporation and the University of Maryland. Antonio Nappa has been the leading author.

3.2 Introduction

In recent years, considerable efforts have been devoted to reducing the impact of software vulnerabilities, including efforts to speed up the creation of patches in response to vulnerability disclosures [15]. However, vulnerability exploits remain an important vector for malware delivery [123, 124, 17].

Prior measurements of patch deployment [47, 48, 49, 50, 18] have focused on server-side vulnerabilities. Thus, the lifecycle of vulnerabilities in client-side applications, such as browsers, document editors and readers, or media players, is not well understood. Such client-side vulnerabilities represent an important security threat, as they are widespread (e.g., typical Windows users are exposed to 297 vulnerabilities in a year [53]), and they are often exploited using spear-phishing as part of targeted attacks [125, 126, 127].

One dangerous peculiarity of client-side applications is that the same host may be affected by several instances of the same vulnerability. This can happen if the host has installed multiple instances of the same application, e.g., multiple software lines or the default installation and an older version bundled with a separate application. Multiple instances of the vulnerable code can also occur owing

to libraries that are shared among multiple applications (e.g., the Adobe library for playing Flash content, which is included with Adobe Reader and Adobe Air installations). These situations break the *linear model for the vulnerability lifecycle* [13, 14, 15, 16, 17], which assumes that the vulnerability is disclosed publicly, then a patch released, and then vulnerable hosts get updated. In particular, vulnerable hosts may only patch one of the program installations and remain vulnerable, while patched hosts may later re-join the vulnerable population if an old version or a new application with the old code is installed. This extends the window of opportunity for attackers who seek to exploit vulnerable hosts. Moreover, the owners of those hosts typically believe they have already patched the vulnerability.

To the best of our knowledge, we present the first systematic study of patch deployment for client-side vulnerabilities. The empirical insights from this study allow us to identify several new threats presented by multiple installations and shared code for patch deployment and to quantify their magnitude. We analyze the patching by 8.4 million hosts of 1,593 vulnerabilities in 10 popular Windows client applications: 4 browsers (Chrome, Firefox, Opera, Safari), 2 multimedia players (Adobe Flash Player, Quicktime), an email client (Thunderbird), a document reader (Adobe Reader), a document editor (Word), and a network analysis tool (Wireshark).

Our analysis combines telemetry collected by Symantec's security products, running on end-hosts around the world, and data available in several public repositories. Specifically, we analyze data spanning a period of 5 years available through the Worldwide Intelligence Network Environment (WINE) [21]. This data includes information about binary executables downloaded by users who opt in for Symantec's data sharing program. Using this data we analyze the deployment of subsequent versions of the 10 applications on real hosts worldwide. This dataset provides a unique opportunity for studying the patching process in client applications, which are difficult to characterize using the network scanning techniques employed by prior research [47, 48, 49, 50, 18].

The analysis is challenging because each software vendor has its own software management policies, e.g., for assigning program versions, using program lines, and issuing security advisories, and also by the imperfect information available in public vulnerability databases. To address these challenges we have developed a generic approach to map files in a host to vulnerable and non-vulnerable program versions, using file meta-data from WINE and VirusTotal [25], and the NVD [22] and OSVDB [23] public vulnerability databases. Then, we aggregate vulnerabilities in those databases into clusters that are patched by the same program version. Finally, using a statistical technique called *survival analysis* [128], we track the global decay of the vulnerable host population for each vulnerability cluster, as software updates are deployed.

Using this approach we can estimate for each vulnerability cluster the delay to issue a patch, the rate of patching, and the vulnerable population in WINE.

Using exploit meta-data in WINE and the Exploit Database [24], we estimate the dates when exploits become available and we determine the percentage of the host population that remains vulnerable upon exploit releases.

We quantify the race between exploit creators and the patch deployment, and we find that the median fraction of hosts patched when exploits are released is at most 14%. All but one of the exploits detected in the wild found more than 50% of the host population still vulnerable. The start of patching is strongly correlated with the disclosure date, and it occurs within 7 days before or after the disclosure for 77% of the vulnerabilities in our study—suggesting that vendors react promptly to the vulnerability disclosures. The rate of patching is generally high at first: the median time to patch half of the vulnerable hosts is 45 days. We also observe important differences in the patching rate of different applications: none of the applications except for Chrome (which employs automated updates for all the versions we consider) are able to patch 90% of the vulnerable population for more than 90% of vulnerability clusters.

Additionally, we find that 80 vulnerabilities in our dataset affect common code shared by two applications. In these cases, the time between patch releases in the different applications is up to 118 days (with a median of 11 days), facilitating the use of patch-based exploit generation techniques [129]. Furthermore, as the patching rates differ between applications, many hosts patch the vulnerability in one application but not in the other one. We demonstrate two novel attacks that enable exploitation by invoking old version of applications that are used infrequently, but that remain installed.

We also analyze the patching behavior of 3 user categories: professionals, software developers, and security analysts. For security analysts and software developers the median time to patch 50% of vulnerable hosts is 18 and 24 days, respectively, while for the general user it is 45 days—more than double.

In summary, we make the following contributions:

- We conduct a systematic analysis of the patching process of 1,593 vulnerabilities in 10 client-side applications, spanning versions released on a 5-year period.
- We demonstrate two novel attacks that enable exploitation by invoking old versions of applications that are used infrequently, but remain installed.
- We measure the patching delay and several patch deployment milestones for each vulnerability.
- We propose a novel approach to map files on end-hosts to vulnerable and patched program versions.
- Using these techniques, we quantify the race between exploit creators and the patch deployment, and we analyze the threats presented by multiple installations and shared code for patch deployment.

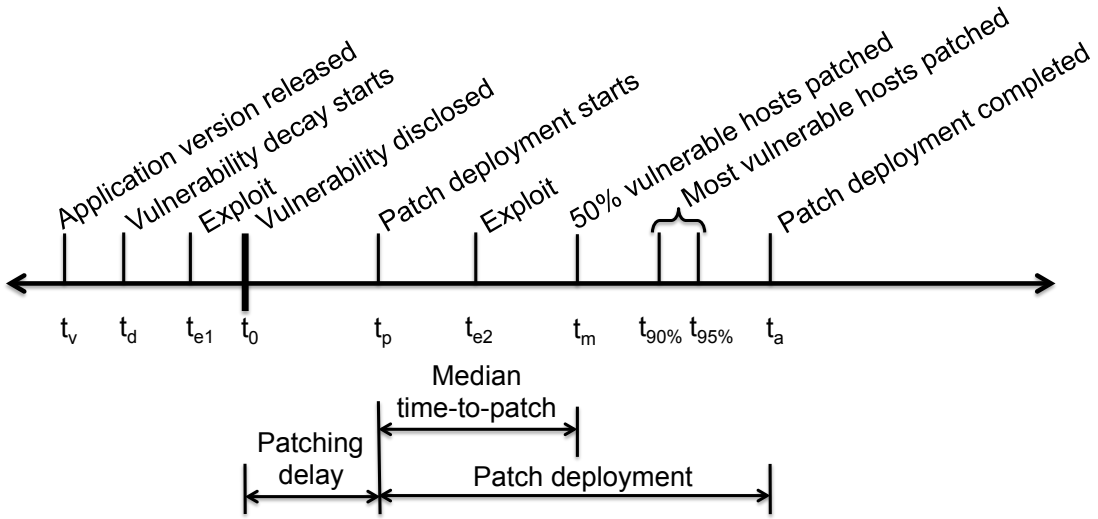


Figure 3.1: Events in the vulnerability lifecycle. We focus on measuring the patching delay $[t_0, t_p]$ and on characterizing the patch deployment process, between t_p and t_a.

- We identify several errors in the existing vulnerability databases, and we release a cleaned dataset at `http://clean-nvd.com/`.

The rest of this paper is organized as follows. Section 3.3 describes the security model for patching vulnerabilities. Section 3.4 details our datasets and Section 3.5 our approach. Our findings are presented in Section 3.6. Section 3.7 reviews prior work, Section 3.8 discusses implications of our findings, and Section 3.10 concludes.

3.3 Security Model for Patching Vulnerabilities.

Prior research [13, 48, 49, 14, 50, 15, 16, 17] generally assumes a *linear model* for the vulnerability lifecycle, illustrated in Figure 3.1. In this model, the introduction of a vulnerability in a popular software (t_v), is followed by the vulnerability's public disclosure (t_0), by a patch release (t_p) and by the gradual deployment of the patch on all vulnerable hosts, which continues until all vulnerable application instances have been updated (t_a). Important milestones in the patch deployment process include the *median time to patch*, i.e., the time needed to patch half of the vulnerable hosts (t_m), and the times needed to patch 90% and 95% of the vulnerable hosts ($t_{90\%}$ and $t_{95\%}$). Additional events may occur at various stages in this lifecycle; for example exploits for the vulnerability may be released before or after disclosure (t_{e1}, t_{e2}), and the vulnerable host population may start decaying earlier than t_p if users replace the vulnerable version with a version that does not include the vulnerability (t_d). Notwithstanding these sources of variability, the

linear model assumes that $t_v < t_0 \leq t_p < t_m < t_{90\%} < t_{95\%} < t_a$.

In practice, however, these events do not always occur sequentially, as a host may be affected by *several instances of the vulnerability*. Software vendors sometimes support multiple product lines; for example, Adobe Reader has several lines (e.g., 8.x, 9.x, 10.x) that are developed and released with some overlap in time and that can be installed in parallel on a host. Additionally, applications are sometimes bundled with other software products, which install a (potentially older) version of the application in a custom directory. For example, device drivers such as printers sometimes install a version of Adobe Reader, to allow the user to read the manual. Furthermore, some applications rely on common libraries and install multiple copies of these libraries side-by-side. For example, Safari and Chrome utilize the WebKit rendering engine [130], Firefox and Thunderbird share several Mozilla libraries, and libraries for playing Flash files are included in Adobe Reader and Adobe Air. In consequence, releasing and deploying the vulnerability patch on a host does not always render the host immune to exploits, as the vulnerability may exist in other applications or library instances and may be re-introduced by the installation of an older version or a different application. The security implications of this *non-linear* vulnerability lifecycle are not well understood.

3.3.1 Threats of Shared Code and Multiple Installations

The response to vulnerabilities is subject to two delays: the patching delay and the deployment delay. The *patching delay* is the interval between the vulnerability disclosure t_0 and the patch release t_p in Figure 3.1. This delay allows attackers to create exploits based on the public details of the vulnerability and to use them to attack all the vulnerable instances of the application *before* they can be patched. After developing and testing a patched version of the application, the vendor must then deploy this version to all the vulnerable application instances. This *deployment delay* controls the window when vulnerabilities can be exploited *after* patches are available, but before their deployment is completed. In recent years, considerable efforts have been devoted to reducing the vulnerability patching delays, through efforts to speed up the creation of patches in response to vulnerability disclosures [15], and the patch deployment delays, through automated software updating mechanisms [51, 52]. While these techniques are aimed at patching vulnerabilities in the linear lifecycle model, attackers may leverage shared code instances and multiple application installations to bypass these defenses. Next, we review some of these security threats.

Overhead of maintaining multiple product lines. When a vendor supports multiple product lines in parallel and a vulnerability is discovered in code shared among them, the vendor must test the patch in each program line. Prior work on optimal patch-management strategies has highlighted the trade-off between the

patching delay and the amount of testing needed before releasing patches [14]. The overhead of maintaining multiple product lines may further delay the release of patches for some of these lines.

Threat of different patch release schedules. When a vulnerability affects more than one application, patch releases for all these applications seldom occur at the same time. Coordinated patch releases are especially difficult to achieve when applications from different vendors share vulnerabilities, e.g., when the vulnerabilities affect code in third-party libraries. When the patch for the first application is released, this gives attackers the opportunity to employ patch-based exploit generation techniques [129] to attack the other applications, which remain unpatched.

Threat of multiple patching mechanisms. When shared code is patched using multiple software update mechanisms, some instances of the vulnerability may be left unpatched. The attacker can use existing exploits to target all the hosts where the default application used for opening a certain type of file (e.g., PDF documents) or Web content (e.g., Flash) remains vulnerable. This situation may happen even after the vendor has adopted automated software updates—for example, when the user installs (perhaps unknowingly, as part of a software bundle) an older version of the application, which does not use automated updates, and makes it the default application, or when the user disables automatic updates on one of the versions installed and forgets about it. The use of multiple software updating mechanisms places a significant burden on users, as a typical Windows user must manage 14 update mechanisms (one for the operating system and 13 for the other software installed) to keep the host fully patched [53].

This problem can also occur with shared libraries because they typically rely on the updating mechanisms of the programs they ship with, but one of those programs may not have automatic updates or they may have been disabled. This scenario is common with third-party libraries deployed with programs from different vendors, which have different update mechanisms and policies. Additionally, we identify 69 vulnerabilities shared between Mozilla Firefox and Mozilla Thunderbird and 3 vulnerabilities shared between Adobe Reader and Adobe Flash; the updating mechanisms used by most of the program versions affected by these vulnerabilities were not fully automated and required some user interaction.

Attacks against inactive program versions through multiple content delivery vectors. Even if all the applications that are actively used on a host are all up to date, an attacker may deliver exploits by using a vector that will open a different runtime or application, which remains vulnerable. Here, we discuss an attack that allows exploiting a vulnerable version even if the patched version is the default one. The user runs both an up-to-date Flash plugin and an old Adobe

Air (a cross-platform runtime environment for web applications), which includes a vulnerable Flash library (npswf32.dll). Adobe Air is used by web applications that want to run as desktop applications across different platforms. In this case the user runs FLVPlayer over Adobe Air. The attacker can deliver the exploit to the user in two ways: as a .flv file to be played locally or as a URL to the .flv file. If the user clicks on the file or URL, the file will be opened with FLVPlayer (associated to run .flv files) and the embedded Flash exploit will compromise the vulnerable Flash library used by Adobe Air. Similarly, Adobe Reader includes a Flash library. The attacker can also target hosts that have up-to-date Flash plugins, but old Adobe Reader installations, by delivering a PDF file that embeds an exploit against the old version of the Flash library.

Attacks against inactive program versions through user environment manipulation. The previous attack relies on the presence of applications to which the attacker can deliver exploit content (e.g., Flash content). Here, we discuss another attack, which allows the attacker to replace the default, up-to-date, version of an application with a vulnerable one. The user is running two versions of Adobe Reader, a default up-to-date version and a vulnerable version. The attacker convinces the user to install a Firefox add-on that looks benign. The malicious add-on has filesystem access through the XPCOM API [131]. It locates the vulnerable and patched versions of the Adobe Reader library (nppdf32.dll) and overwrites the patched version with the vulnerable one. When the user visits a webpage, the malicious add-on modifies the DOM tree of the webpage to insert a script that downloads a remote PDF exploit. The exploit is processed by the vulnerable Adobe Reader version and exploitation succeeds. We have successfully exploited a buffer overflow vulnerability (CVE-2009-1861) on Firefox 33.1, Adobe Reader 11.0.9.29 as patched version and Acrobat Reader 7.0.6 as vulnerable version.

These two attacks demonstrate the dangers of inactive application versions that are forgotten, but remain installed. Note that our goal is not to find exploits for all the programs we analyze, but rather to provide evidence that a sufficiently motivated attacker can find avenues to exploit multiple installations and shared code.

3.3.2 Goals and Non-Goals

Goals. Our goal in this paper is to determine how effective are update mechanisms in practice and to quantify the threats discussed in Section 3.3.1. We develop techniques for characterizing the patching delay and the patch deployment process for vulnerabilities in client-side applications, and use these techniques to assess the impact of current software updating mechanisms on the vulnerability levels of real hosts. The *patching delay* depends on the vendor's disclosure and patching policies and requires measuring t_0 and t_p for each vulnerability. t_0 is

recorded in several vulnerability databases [22, 23], which often include incomplete (and sometimes incorrect) information about the exact versions affected by the vulnerability. Information about t_p is scattered in many vendor advisories and is not centralized in any database. To overcome these challenges, we focus on analyzing the presence of vulnerable and patched program versions on real end-hosts, in order to estimate the start of patching and to perform sanity checks on the data recorded in public databases.

The *patch deployment process* depends on the vendor (e.g., its patching policy), the application (e.g., if it uses an automated updating mechanism), and the user behavior (e.g., some users patch faster than others). The daily changes in the population of vulnerable hosts, reflected in our end-host observations, give insight into the progress of patch deployment. We focus on modeling this process using statistical techniques that allow us to measure the *rate* at which patching occurs and several patch deployment *milestones*. To characterize the *initial deployment phase* following patch deployment (t_p), we estimate the median time to patch (t_m). The point of *patch completion* t_a is difficult to define, because we are unable to observe the entire vulnerable host population on the Internet. To characterize the *tail of the deployment process*, we estimate two additional deployment milestones, ($t_{90\%}$ and $t_{95\%}$), as well as the fraction of vulnerabilities that reach these patching milestones. Our focus on application vulnerabilities (rather than vulnerabilities in the underlying OS) allows us to compare the effectiveness of different software updating mechanisms. These mechanisms are used for deploying various kinds of updates (e.g., for improving performance or for adding functionality); we focus on updates that patch security vulnerabilities. We also aim to investigate the impact of application-specific and user-specific factors on the patching process.

To interpret the results of our analysis, it is helpful to compare our goals with those of the prior studies on vulnerability patching and software update deployment. Several studies [47, 48, 49, 50] conducted remote vulnerability scans, which allowed them to measure vulnerabilities in server-side applications, but not in client-side applications that do not listen on the network. Another approach for measuring the patch deployment speed is to analyze the logs of an update management system [51], which only covers applications utilizing that updating system and excludes hosts where the user or the system administrator has disabled automated updating, a common practice in enterprise networks. Similarly, examining the User-Agent string of visitors to a popular website [52] only applies to web browsers, is confounded by the challenge of enumerating hosts behind NATs, and excludes users not visiting the site. In contrast, we aim to compare multiple client-side applications from different vendors, employing multiple patch deployment mechanisms. Because we analyze data collected on end hosts, we do not need to overcome the problem of identifying unique hosts over the network and our results are not limited to active instances. Instead, we can analyze the patch deployment for applications that are seldom used and that may remain

vulnerable under the radar (e.g., when a user installs multiple browsers or media players).

Applications selected. We select 10 desktop applications running on Windows operating systems: 4 browsers (Chrome, Firefox, Opera, Safari), 2 multimedia players (Adobe Flash Player, Quicktime), an email client (Thunderbird), a document reader (Adobe Reader), a document editor (Word), and a networking tool (Wireshark). We choose these applications because: (1) they are popular, (2) they are among the top desktop applications with most vulnerabilities in NVD [22], and (3) they cover both proprietary and open source applications. Across all these applications, we analyze the patching process of 1,593 vulnerabilities, disclosed between 2008–2012. All programs except Word can be updated free of charge. All programs replace the old version with the new one after an upgrade except Adobe Reader, for which new product lines are installed in a new directory and the old line is kept in its current directory.

Non-goals. We do not aim to analyze the entire vulnerability lifecycle. For example, determining the precise dates when vulnerability exploits are published is outside the scope of this paper. Similarly, we do not aim to determine when a patch takes effect, e.g., after the user has restarted the application. Instead, we focus on patch deployment, i.e., patch download and installation. Finally, we do not aim to determine precisely when the patch deployment is completed, as old versions of applications are often installed along with driver packages or preserved in virtual machine images, and can remain unpatched for very long periods.

3.4 Datasets

We analyze six datasets: WINE's binary reputation [21] to identify files installed by real users, the NVD [22] and OSVDB [23] vulnerability databases to determine vulnerable program versions and disclosure dates, the EDB [24] and WINE-AV for exploit release dates, and VirusTotal [25] for additional file meta-data (e.g., AV detections, file certificates). These datasets are summarized in Table 3.1 and detailed next.

WINE–binary reputation. The Worldwide Intelligence Network Environment (WINE) [132] provides access to data collected by Symantec's anti-virus and intrusion-detection products on millions of end-hosts around the world. Symantec's users have a choice of opting-in to report telemetry about security events (e.g., executable file downloads, virus detections) on their hosts. WINE does not include user-identifiable information. These hosts are real computers, in active use around the world.

Table 3.1: Summary of datasets used.

Dataset	Analysis Period	Hosts	Vul.	Exp.	Files
WINE-BR	01/2008–12/2012	8.4 M	–	–	7.1 M
VirusTotal	10/2013-04/2014	–	–	–	5.1 M
NVD	10/1988-12/2013	–	59 K	–	–
OSVDB	01/1972-01/2012	–	77 K	–	–
EDB	01/2014	–	–	25 K	–
WINE-AV	12/2009–09/2011	–	–	244	–

We use the binary reputation dataset in WINE (WINE-BR), which is collected from 8.4 million Windows hosts that have installed Symantec's consumer AV products. WINE-BR records meta-data on all—benign or malicious—As shown in Figure 3.2, the WINE hosts are concentrated in North America and Europe with the top-20 countries accounting for 89.83% of all hosts. Since these are real hosts, their number varies over time as users install and uninstall Symantec's products. Figure 3.3 shows the number of simultaneous reporting hosts over time. It reaches a plateau at 2.5 M hosts during 2011.

The hosts periodically report new executables found. Each report includes a timestamp and for each executable, the hash (MD5 and SHA2), the file path and name, the file version, and, if the file is signed, the certificate's subject and issuer. The binaries themselves are not included in the dataset. Since we analyze the vulnerability lifecycle of popular programs our analysis focuses on the subset of 7.1 million files in WINE-BR reported by more than 50 users between 2008 and 2012.

NVD. The National Vulnerability Database [22] focuses on vulnerabilities in commercial software and large open-source projects. Vulnerability disclosures regarding less-well-known software, e.g., small open source projects, are typically redirected to other vulnerability databases (e.g., OSVDB). NVD uses CVE identifiers to uniquely name each vulnerability and publishes XML dumps of the full vulnerability data. We use the NVD dumps until the end of 2013, which comprise 59,875 vulnerabilities since October 1988[1].

OSVDB. The Open Sourced Vulnerability Database[2] [23] has the goal of providing technical information on every public vulnerability, so it contains vulnerabilities in more programs than NVD. OSVDB uses its own vulnerability identifier but also references the CVE identifier for vulnerabilities with one. Up to early

[1]CVE identifiers start at CVE-1999-0001, but CVE-1999-* identifiers may correspond to vulnerabilities discovered in earlier years.

[2]Previously called Open Source Vulnerability Database.

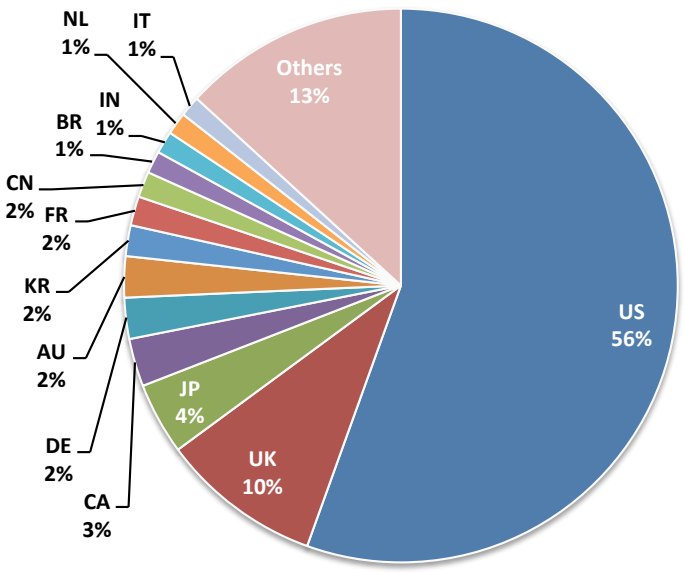


Figure 3.2: Geographical distribution of reporting hosts in WINE-BR.

2012, OSVDB made publicly available full dumps of their database. However, OSVDB has since moved away of their original open source model and public dumps are no longer available. Our OSVDB data comes from one of the latest public dumps on January 12, 2012. It contains 77,101 vulnerabilities since January 2004.

EDB. The Exploit Database [24] is an online repository of vulnerability exploits. We crawl their web-pages to obtain meta-data on 25,331 verified exploits, e.g., publication date and vulnerability identifier (CVE/OSVDB).

WINE–AV. The AV telemetry in WINE contains detections of known cyber threats on end hosts. We can link some of these threats to 244 exploits of known vulnerabilities, covering 1.5 years of our observation period. We use this dataset to determine when the exploits start being detected in the wild.

VirusTotal. VirusTotal [25] is an online service that analyzes files and URLs submitted by users with multiple security / anti-virus products. VirusTotal offers a web API to query meta-data on the collected files including the AV detection rate and information extracted statically from the files. We use VirusTotal to obtain additional meta-data on the WINE files, e.g., detailed certificate informa-

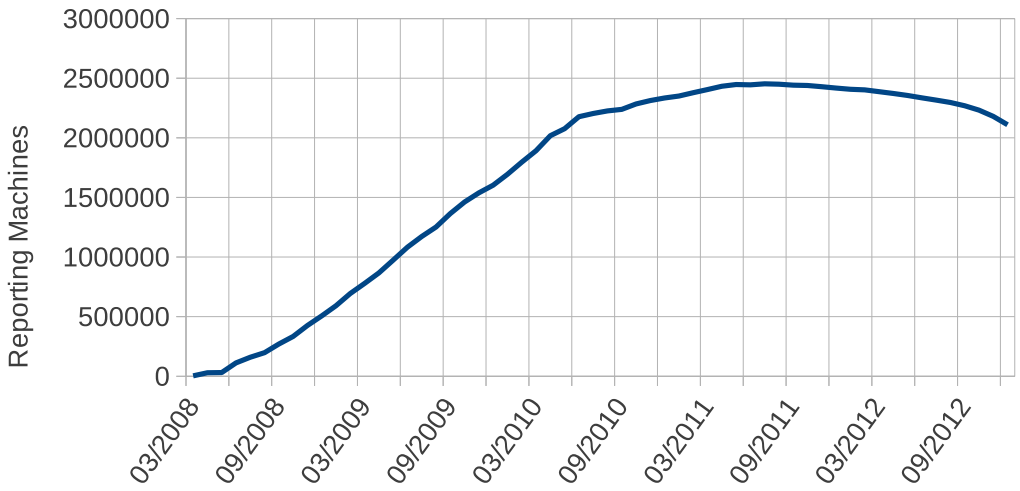


Figure 3.3: Reporting hosts in WINE-BR over time.

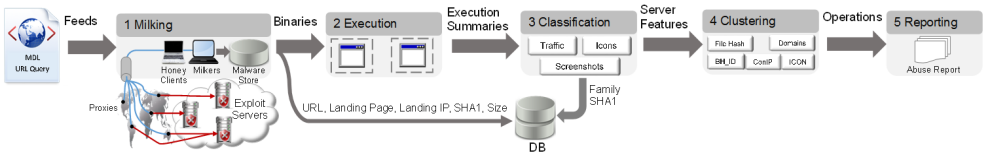


Figure 3.4: Approach overview.

tion and the values of fields in the PE header. This information is not available otherwise as we do not have access to the WINE files, but we can query VirusTotal using the file hash. Overall, VirusTotal contains an impressive 5.1 million (72%) of the popular files in WINE's binary reputation dataset.

Release histories. In addition to the 6 datasets in Table 3.1, we also collect from vendor websites the release history for the programs analyzed, e.g., Chrome [133], Safari [134]. We use release histories to differentiate beta and release program versions and to check the completeness of the list of release versions observed in WINE.

3.5 Vulnerability Analysis

Our study of the vulnerability lifecycle comprises two main steps. First, we develop an approach to map files on end-hosts to vulnerable and patched program versions (Sections 3.5.1 through 3.5.5). Then, we perform survival analysis to

study how users deploy patches for each vulnerability (Section 3.5.4). Our approach to map files to vulnerable and patched program versions is novel, generic, and reusable. While it comprises some program-specific data collection, we have successfully adapted and applied our approach to 10 programs from 7 software vendors. Ensuring the generality of the approach was challenging as each vendor has their own policies. Another challenge was dealing with the scattered, often incomplete, and sometimes incorrect data available in public repositories. We are publicly releasing the results of our data cleaning for the benefit of the community (Appendix 3.9). We designed our approach to minimize manual work so that it can be applied to hundreds of vulnerabilities for each program and that adding new programs is also easy. The approach is not specific to WINE; it can be used by any end host application that periodically scans the host's hard-drive producing tuples of the form <machine, timestamp, file, filepath> for the executable files it finds. Most antivirus software and many other security tools fit this description.

Figure 3.4 summarizes our approach. First, we pre-process our data (Section 3.5.1). Then, we identify version-specific files that indicate a host has installed a specific program version (Section 3.5.2). Next, our vulnerability report generation module automatically identifies which files indicate vulnerable and non-vulnerable versions for each vulnerability (Section 3.5.3). We detail our cleaning of NVD data in Appendix 3.9. Finally, we perform a survival analysis for each vulnerability (Section 3.5.4).

3.5.1 Data Pre-Processing

The first step in our approach is to pre-process the 7.1 million files in our dataset to have the information needed for the subsequent steps. Note that while we focus on 10 selected applications, we do not know a priori which files belong to those. We first assign to each file (identified by MD5 hash) a default filename, first seen timestamp, and file version. The default filename is selected by majority voting across all WINE hosts reporting files. Since most users do not modify default filenames and the files are reported by at least 50 hosts, the filename used by the largest number of hosts is highly likely the vendor's default. The first seen timestamp is the earliest time that the file was reported by any host. For the file version, each Windows executable contains version information in its PE header, which WINE normalizes as four decimal numbers separated by dots, e.g., 9.0.280.0.

In addition, we query VirusTotal for the files' metadata including the number of AV products flagging the file as malicious and the file certificate. We consider malware the 1.3% of the 7.1 M files flagged by 3 or more AV products as malicious, removing them from further analysis. If the file has a valid certificate, we extract the publisher and product information from its metadata.

Table 3.2: Summary of the selected programs.

	Configuration		WINE			NVD vulnerabilities			Auto Updates Introd.	
Program	**Leading Filename**	**Lines**	**Files**	**Ver.**	**Rel.**	**Total**	**Selected**	**Clust.**	**Ver.**	**Date**
Chrome	chrome.dll	1	544	531	531	886	545 (61%)	87	1.0	2008-12-12
Firefox	firefox.exe	10	457	440	140	1,013	226 (22%)	88	15	2012-08-26
Flash	npswf32%.dll	12	187	121	121	316	123 (38%)	22	11.2	2012-03-27
Opera	opera.exe	1	95	90	90	25	10 (40%)	8	10	2009-08-30
Quicktime	quicktimeplayer.exe	1	68	43	41	206	67 (33%)	15	-	2008-01-01
Reader	acrord32.dll	9	102	73	66	330	159 (48%)	30	10.1	2011-06-14
Safari	safari.exe	1	35	29	26	460	227 (49%)	39	-	2008-01-01
Thunderbird	thunderbird.exe	17	91	91	83	594	31 (15%)	60	17	2012-11-20
Wireshark	wireshark.exe	1	56	38	38	261	110 (42%)	66	1.10.0	2013-06-05
Word	winword.exe	7	247	104	7	92	36 (39%)	12	2003	2005-06-01
			TOTAL	2,006	1,610	1,242	4,347	1,593 (39%)	408	

3.5.2 Mapping Files to Program Versions

To determine if a WINE host installed a specific program version we check if the host has installed some files specific to that program version. Such *version-specific files* should not be part of other program versions (or other programs) and should not correspond to installer files (e.g., firefox_setup_3.0.7.exe) but rather to the final executables or libraries (e.g., firefox.exe) that are only present on the host's file system if the user not only downloaded the program version, but also installed it. The goal is to generate a list of triples $< h, p, v_p >$ capturing that the presence of a file with hash h in a host indicates the installation of program p and version v_p.

An intuitive method to identify version-specific files would be to install a program version, monitoring the files it installs. Files installed by one program version but not by any other version would be version-specific. However, such approach is difficult to generalize because for some programs it is not easy to obtain all the program versions released between 2008–2012, and because automating the installation process of arbitrary programs is challenging given the diversity in installation setups and required user interaction.

We have developed an alternative analyst-guided approach to identify version-specific files. Our approach leverages the fact that large software vendors, e.g., those of our selected programs, sign their executables to provide confidence in their authenticity, and that, to keep software development manageable, they keep filenames constant across program versions (e.g., Firefox's main executable is named firefox.exe in all program versions) and update the file version of files modified in a new program version.

For each program analyzed, we first query our database for how many different file versions each default filename associated with the program has (e.g., files with vendor "Adobe%" and product "%Reader%"[3]). We select the non-installer filename with most file versions as the *leading filename*. In 7 of our 10 programs the leading filename corresponds to the main executable (e.g., firefox.exe, wireshark.exe) and in other 2 to the main library (chrome.dll for Chrome, acrord32.dll

[3]We use % as a wildcard as in SQL queries

for Reader). For Flash Player, Adobe includes the version in the default filename since March 2012, e.g., npswf32_11_2_202_18.dll, so we use a wildcard to define the leading filename, i.e., npswf32%.dll.

Mapping file version to program version. The file version of files with the leading filename may correspond directly to the program version (e.g., chrome.dll 5.0.375.70 indicates Chrome 5.0.375.70), to a longer version of the program version (e.g., acrord32.dll 9.1.0.163 indicates Reader 9.1), or the relationship may not be evident, (e.g., firefox.exe 1.9.0.3334 indicates Firefox 3.0.7). For each program we build a *version mapping*, i.e., a list of triples $< v_f, v_p, type >$ indicating that file version v_f corresponds to program version v_p. We use the type to indicate the maturity level of the program version, e.g., alpha, beta, release candidate, release. We limit the patch deployment analysis to release versions of a program. The mapping of file to program versions is evident for 7 programs, the exception being Safari and old versions of Firefox and Thunderbird[4]. For Safari we use its public release history [134] and for Firefox/Thunderbird we leverage the fact that the developers store the program version in the product name field in the executable's header, available in the VirusTotal metadata. For the program version type, we leverage the program release history from the vendor's site.

Checking for missing versions. In most cases, the file with the leading filename is updated in every single program version. However, for Adobe Reader we found a few program versions that do not modify the file with the leading filename but only other files (e.g., Adobe Reader 9.4.4 did not modify acrord32.dll or acrord32.exe). To identify if we are missing some program version we compare the version mapping with the vendor's release history. For any missing version, we query our database for all files with the missed file version and signed by the program's vendor (regardless of the filename). This enables identifying other files updated in the missing version, e.g., nppdf32.dll in Reader 9.4.4. We add one of these file hashes to our list of version-specific files. Of course, if the vendor did not update the version of any executable file in a new program version, we cannot identify that program version.

Product lines. Some programs have product lines that are developed and released with some overlap in time. For our analysis it is important to consider product lines because a vulnerability may affect multiple lines and each of them needs to be patched separately. For example, vulnerability CVE-2009-3953 is patched in versions 8.2 and 9.3 of Adobe Reader, which belong to lines 8 and 9 respectively. To map product versions to product lines we use regular expressions. Next section describes our handling of product lines.

[4]Firefox and Thunderbird use file versions similar to program versions since Firefox 5 and Thunderbird 5.

As the output of this process the analyst produces a program configuration file that captures the list of version-specific files, the version mapping, and the product line regular expressions. This configuration is produced once per program, independently of the number of program vulnerabilities that will be analyzed.

The left-side section of Table 3.2 summarizes the configuration information for the 10 selected programs: the program name, the vendor, the leading filename, and the number of product lines. The right-side section shows on what date, and in which version, the application started using an automated updating mechanism. The WINE section captures the number of version-specific files (Files), the number of file versions for those files (Ver.), and the number of program versions they correspond to (Rel.) The 3 numbers monotonically decrease since two files may occasionally have the same file version, e.g., 32-bit and 64-bit releases or different language packs for the same version. Two file versions may map to the same program version, e.g., beta and release versions.

Note that we start with 7.1 M files and end up with only 2,006 version-specific files for the 10 programs being analyzed. This process would be largely simplified if program vendors made publicly available the list of file hashes corresponding to each program version.

3.5.3 Generating Vulnerability Reports

Next, our vulnerability report generation module takes as input the configuration file for a program and automatically determines for each vulnerability for the program in NVD, which versions are vulnerable and not vulnerable.

For this, it first queries our database for the list of NVD vulnerabilities affecting the program. If a vulnerability affects multiple programs (e.g., Firefox and Thunderbird), the vulnerability will be processed for each program. For each vulnerability, it queries for the list of vulnerable program versions in NVD and splits them by product line using the regular expressions in the configuration. For each vulnerable product line (i.e., with at least one vulnerable version), it determines the range of vulnerable program versions $[x_l, y_l]$ in the line. Note that we have not seen lines with multiple disjoint vulnerable ranges. Finally, it annotates each version-specific file as not vulnerable (NV), vulnerable (V), or patched (P). For a file specific to program version v_p, it compares v_p with the range of vulnerable versions for its line. If the line is not vulnerable, the file is marked as not vulnerable. If the line is vulnerable but $v_p < x_l$, it is marked as not vulnerable; if $x_l \leq v_p \leq y_l$ as vulnerable; and if $v_p > y_l$ as patched.

We discard vulnerabilities with no vulnerable versions, no patched versions, or with errors in the NVD vulnerable version list. We may not find any vulnerable versions for vulnerabilities in old program versions (that no WINE host installs between 2008–2012). Similarly, we may find no patched versions for vulnerabilities disclosed late in 2012 that are patched in 2013. We detail how we identify NVD errors in Appendix 3.9.

Multiple vulnerabilities may affect exactly the same program versions. Such vulnerabilities have the same vulnerable population and identical patching processes. We therefore group them together into *vulnerability clusters* to simplify the analysis. All vulnerabilities in a cluster start patching on the same date, but each vulnerability may have a different disclosure date.

The "NVD vulnerabilities" section of Table 3.2 shows the total number of vulnerabilities analyzed for each program (Total), the selected vulnerabilities after discarding those with no vulnerable or patched versions, or with errors (Selected), and the number of vulnerability clusters (Clust.). Overall, we could analyze 39% of all vulnerabilities in the 10 programs.

Patching delay. For each cluster, we compute the disclosure date t_0 as the minimum of the disclosure dates in NVD and OSVDB. The start of patching t_p is the first date when we observe a patched version in the field data from WINE. The start of patching date often differs from the disclosure date and is not recorded in NVD or OSVDB. The patching delay for a cluster is simply $pd = t_p - t_0$.

3.5.4 Survival Analysis

To analyze the patch deployment speed we employ *survival analysis techniques*, which are widely used in medicine and biology to understand the mortality rates associated with diseases and epidemics [128]. In our case, survival analysis measures the probability that a vulnerable host will "survive" (i.e., remain vulnerable) beyond a specified time. Intuitively, the population of vulnerable hosts decreases (i.e., the vulnerability "dies" on a host) when one of two death events happen: (1) a user installs a *patched* program version or (2) a user installs a *non-vulnerable* program version. While both events decrease the vulnerable population, only the first one is directly related to patch deployment. When we analyze patch deployment we consider only the first death event; when analyzing vulnerability decay we consider both.

Survival function. To understand how long a vulnerability remains exploitable in the wild, we consider that the vulnerability lifetime is a random variable T. The *survival function* $S(t)$ captures the likelihood that the vulnerability has remained unpatched until time t:

$$S(t) = \Pr[T > t] = 1 - F(t)$$

where $F(t)$ is the cumulative distribution function.

We estimate $S(t)$ for each vulnerability cluster. Figure 3.5 illustrates the output of this analysis through an example. Using our $S(t)$ estimations, we can compare the deployment of two software updates in Chrome. Both updates reached 50% of the vulnerable hosts in a few days, days, but update #1 reached

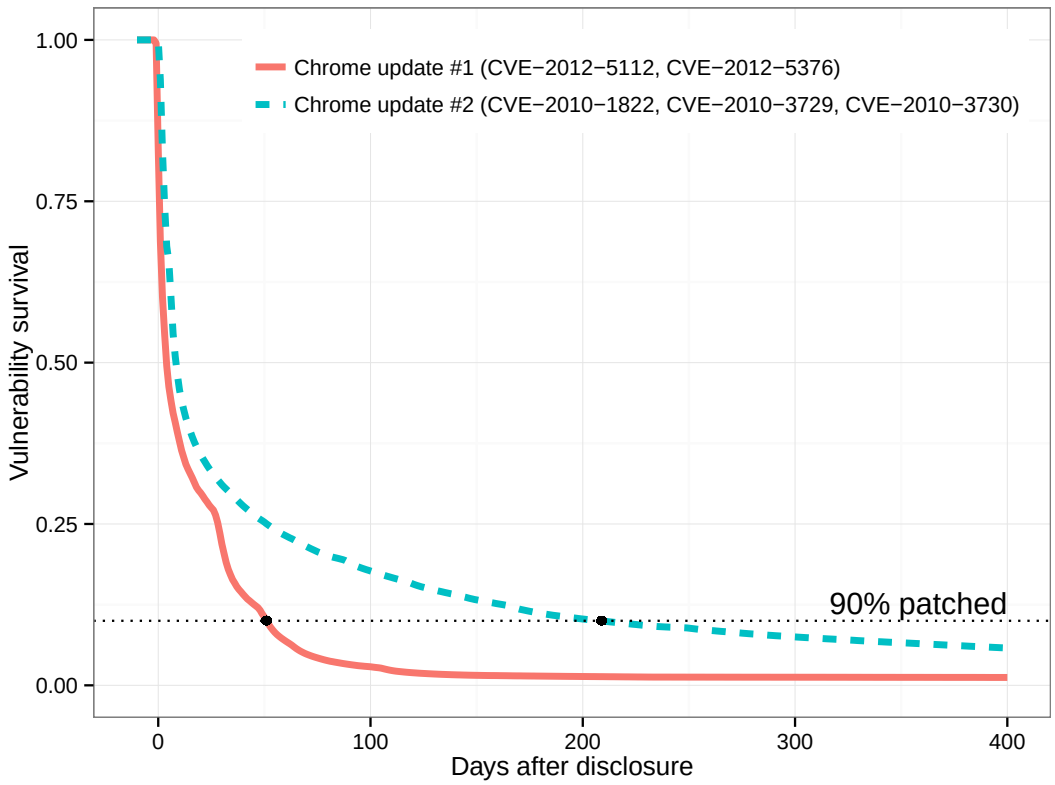


Figure 3.5: Examples of vulnerability survival, illustrating the deployment of two updates for Google Chrome.

90% of vulnerable hosts in 52 days, while update #2 needed 4× as much time to reach the same milestone.

The survival function describes how the probability of finding vulnerable hosts on the Internet decreases over time. $S(t)$ is 1 in the beginning, when no vulnerable hosts have yet been patched, and 0 when there are no vulnerable hosts left. By definition, $S(t)$ is monotonically decreasing; it decreases when one of the two death events mentioned earlier occurs on a vulnerable host. In our analysis of the patch deployment process, the installation of a patched version closes the vulnerability on the host. In this case, the point in time when $S(t)$ starts decreasing corresponds to the *start of patching* t_p, which we use to estimate the patching delay as described in Section 3.5.3. In our analysis of the vulnerability decay process, the installation of either a patched version or a non-vulnerable version closes the vulnerability window. The point where $S(t)$ starts decreasing corresponds to the *start of the vulnerability decay* (t_d in Figure 3.1). In both cases, the host's death event for a given vulnerability corresponds to the first installation of a patched or non-vulnerable version *after* the installation of a vulnerable version on the host.

The time needed to patch a fraction α of vulnerable hosts corresponds to the inverse of the survival function: $t_\alpha = S^{-1}(1-\alpha)$. The survival function allows us to quantify the milestones of patch deployment such as the median time to patch $t_m = S^{-1}(0.5)$ and the time to patch most vulnerable hosts: $t_{90\%} = S^{-1}(0.1)$ and $t_{95\%} = S^{-1}(0.05)$.

Right censoring and left truncation. When estimating $S(t)$ we must account for the fact that, for some vulnerable hosts, we are unable to observe the patching event before the end of our observation period. For example, some hosts may leave our study before patching, e.g., by uninstalling the Symantec product, by opting out of data collection, or by upgrading the OS. In statistical terms, these hosts are independently *right-censored*. We determine that a host becomes right censored by observing the last download report (not necessarily for our selected applications) in WINE. For these hosts, we do not know the exact time (or whether) they will be patched; we can only ascertain that they had not been patched after a certain time. In other words, when a host becomes censored at time t, this does not change the value of $S(t)$; however, the host will no longer be included in the vulnerable host population after time t. In addition, we cannot determine whether vulnerable versions have been installed on a host before the start of our observation period, so some vulnerable hosts may not be included in the vulnerable population. In statistical terms, our host sample is *left-truncated*. Right-censoring and left-truncation are well studied in statistics, and in this paper we compute the values of $S(t)$ using the Kaplan-Meier estimator, which accounts for truncated and censored data. We compute this estimator using the `survival` package for R [135]. We consider that we have reached the end of the observation period for a vulnerability when the vulnerable population falls below 3 hosts or when we observe that 95% or more of the vulnerable hosts become censored within 5% of the total elapsed time. This prevents artificial spikes in the hazard function at the end of our observation period, produced by a large decrease in the vulnerable host population due to right-censoring.

3.5.5 Threats to Validity

Selection bias. The WINE user population is skewed towards certain geographical locations; for example, 56% of the hosts where the data is collected are located in the United States (Figure 3.2). Additionally, WINE's binary reputation only covers users who install anti-virus software. While we cannot exclude the possibility of selection bias, the prevalence of anti-virus products across different classes of users and the large population analyzed in our study (1,593 vulnerabilities on 8.4 million hosts) suggests that our results have a broad applicability. Similarly, our study considers only Windows applications, but the popular client applications we analyze are cross-platform, and often use the same patching mechanism on different platforms. Moreover, cyber attacks have pre-

dominantly targeted the Windows platform.

Sampling bias. The WINE designers have taken steps to ensure WINE data is a representative sample of data collected by Symantec [136]. For our study, this means that the vulnerability survival percentages we compute are likely accurate for Symantec's user base, but the absolute sizes of the vulnerable host population are underestimated by at least one order of magnitude.

Data bias. The WINE binary reputation does not allow us to identify program uninstalls. We can only identify when a user installs new software and whether the newly installed version overwrote the previous one (i.e., installed on the same path). The presence of uninstalls would cause us to under-estimate the rate of vulnerability decay, as the hosts that have completely removed the vulnerable application would be counted as being still vulnerable at the end of our observation period. We do not believe this is a significant factor, as we observe several vulnerabilities that appear to have been patched completely during our observation (for example, update #1 from Figure 3.5).

3.6 Evaluation

We analyze the patching of 1,593 vulnerabilities in 10 applications, which are installed and are actively used on 8.4 M hosts worldwide. We group these vulnerabilities into 408 clusters of vulnerabilities patched together, for an average of 3.9 vulnerabilities per cluster. We conduct survival analysis for all these clusters, over observation periods up to 5 years.

In this section, we first summarize our findings about the update deployment process in each application (Section 3.6.1). Then, we analyze the impact of maintaining parallel product lines on the patching delay (Section 3.6.2), the race between exploit creators and patch deployment (Section 3.6.3), the opportunities for patch-based exploit generation (Section 3.6.4), and the impact of parallel installations of an application on patch deployment (Section 3.6.5). Next, we analyze the time needed to reach several patch deployment milestones (Section 3.6.6) and, finally, the impact of user profiles and of automated update mechanisms on the deployment process (Section 3.6.7).

3.6.1 Patching in Different Applications

The prior theoretical work on optimal patch-management strategies makes a number of assumptions, e.g., that there is an important trade-off between the patching delay and the amount of testing needed before releasing patches, or that patch deployment is instantaneous [14]. In this section, we put such assumptions to the test. We focus on the patching delay and on two patch deployment milestones:

Table 3.3: Milestones for patch deployment for each program (medians reported).

Program	**%Vers. Auto**	**Vul. Pop.**	**Patch Delay**	**Days to patch (%clust.)**	
				t_m	$t_{90\%}$
Chrome	100.0%	521 K	-1	15 (100%)	246 (93%)
Firefox	2.7%	199 K	-5.5	36 (91%)	179 (39%)
Flash	14.9%	1.0 M	0	247 (59%)	689 (5%)
Opera	33.3%	2 K	0.5	228 (100%)	N/A (0%)
Quicktime	0.0%	315 K	1	268 (93%)	997 (7%)
Reader	12.3%	1.1 M	0	188 (90%)	219 (13%)
Safari	0.0%	146 K	1	123 (100%)	651 (23%)
Thunderbird	3.2%	11 K	2	27 (94%)	129 (35%)
Wireshark	0.0%	1 K	4	N/A (0%)	N/A (0%)
Word	37.4%	1.0 M	0	79 (100%)	799 (50%)

reaching 50% and 90% of the vulnerable hosts. For this analysis, we consider only the deployment of a patched version as a vulnerability death event, and we aggregate the results of the survival analysis for each of the 10 selected applications. Because it is difficult to compare $t_{90\%}$ for two applications with different vulnerable populations, we report both the time to reach this milestone and the percentage of updates that reach it. Applications with effective updating mechanisms will be able to reach 90% deployment for a large percentage of updates. The time needed to reach this milestone illustrates how challenging it is to patch the last remaining hosts within a large vulnerable population.

Table 3.3 summarizes the patch deployment milestones for each application. The second column shows the percentage of versions that were updated automatically. Chrome had an automated updating mechanism since its first version; Word used Microsoft Update throughout the study; Wireshark had completely manual updates throughout our study; Safari and Quicktime use the Apple Software Updater that periodically checks and prompts the user with new versions to install; the remaining programs introduced silent updates during our study.

The next column shows the vulnerable host population (we report the median across all vulnerability clusters for the program). For 7 out of the 10 applications the median vulnerable population exceeds 100,000 hosts and for 3 (Flash, Reader, Word) it exceeds one million hosts. In comparison, the 2014 Heartbleed vulnerability in OpenSSL affected 1.4 million servers [137], and the Internet worms from 2001–2004 infected 12K–359K hosts [47, 138, 139]. As explained in Section 3.5.5 the host population in our study only reflects the hosts in WINE, after sampling. The vulnerable host population in the unsampled Symantec data is likely to be

at least one order of magnitude higher [136], and the real vulnerable population when including hosts without Symantec software much larger. These numbers highlight that vulnerabilities in client-side applications can have significant larger vulnerable populations than server-side vulnerabilities.

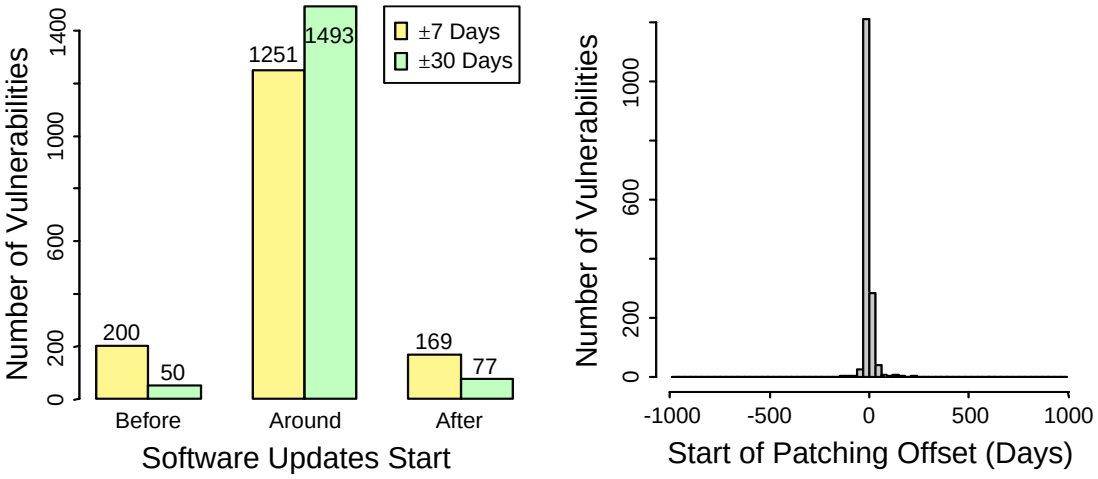
The fourth column reports the median patch delay (in days) for the vulnerability clusters. Chrome and Firefox have negative values indicating that patching starts before disclosure for most of their clusters. Flash, Reader, and Word have zero values indicating that Adobe and Microsoft are coordinating the release of patches and advisories, e.g., at Patch Tuesday for Microsoft. The remaining programs have positive patch delay; the larger the patch delay the longer users are exposed to published vulnerabilities with no patch available.

The last two columns capture the times needed to reach 50% and 90% patch deployment (we report medians across clusters), as well as the percentage of clusters that reached these milestones before the end of our observation period. Chrome has the shortest t_m, followed by Thunderbird and Firefox. At the other extreme, Wireshark exhibits the slowest patching: no cluster reaches 50% patch deployment by the end of 2012—not even for vulnerabilities released in 2008. Excluding Wireshark and Flash, all applications reach 50% patching for over 90% of vulnerability clusters, with Chrome, Opera, Safari, and Word reaching this milestone for 100% of clusters. At t_{90} only Chrome patches more than 90% of the vulnerability clusters. We also observe that Firefox and Thunderbird appear to reach this milestone faster than Chrome, but this comparison is biased by the fact that for these two applications only 35–40% of clusters reach this milestone, in contrast with Chrome's 93%.

Chrome's automated update mechanism results in the fastest patching among the 10 applications. However, it still takes Chrome 246 days (approximately 8.2 months) to achieve 90% patching. This finding contradicts prior research that concluded that 97% of active Chrome instances are updated to the new version within 21 days after the version's release of a new version [52]. The main difference in our approach is that, because we analyze end-host data, our measurements include inactive applications, which are installed on a host but rarely used. Moreover, our measurements do not suffer from confounding factors such as applications on hosts behind NATs, which complicate the interpretation of version numbers extracted from network traces.

3.6.2 Patching Delay

In this section we ask the questions: *How quickly are patches released, following vulnerability disclosures?* and *What is the impact of maintaining multiple product lines on these releases?* The patch can be released *on* the disclosure date (e.g., coordinated disclosure, where the vendor is notified and given some time to create a patch before publicizing the vulnerability), *after* the disclosure date, (e.g., discovery of a zero-day attack in the wild or full disclosure, where the vulnerability is



(a) Correlation with disclosure.

(b) Histogram.

Figure 3.6: Distribution of the patching delay. For each vulnerability, we compare the start of patching with the disclosure date. The left plot shows how many vulnerabilities start patching within a week (±7 days) or a month (±30 days) of disclosure and how many start patching outside these intervals. The right plot shows a histogram of the patch delay (each bar corresponds to a 30-day interval).

publicized despite the lack of a patch in order to incentivize the vendor to create the patch), or *before* disclosure (e.g., the vulnerability does not affect the latest program version). Thus, the patching delay may be positive, zero, or negative. Figure 3.6 illustrates the distribution of the patching delay for all the 1,593 vulnerabilities in our study. We find that the start of patching is strongly correlated with the disclosure date (correlation coefficient $r = 0.994$). Moreover, Figure 3.6 shows that 77% of the vulnerabilities in our study start patching within 7 days before or after the disclosure dates. If we extend this window to 30 days, 92% of vulnerabilities start patching around disclosure. This suggests that software vendors generally respond promptly to vulnerability disclosures and release patches that users can deploy to avoid falling victim to cyber attacks.

It is interesting to observe that maintaining multiple parallel lines does not seem to increase the patching delay. Firefox, Flash, Reader, Thunderbird and Word all have multiple product lines; in fact, each vulnerability cluster in these products affects all the available product lines. When a vulnerability affects multiple product lines, patches must be tested in each program line, which may delay the release of patches. However, Table 3.3 suggests that the median patching delay is not significantly higher for these applications than for the applications with a single product line, like Google Chrome. We note that this doesn't suggest that multiple product lines do not impact software security, as there may be additional software engineering issues associated with product lines; however, we do not find

empirical evidence for these effects.

When we take into consideration both types of vulnerability death events in the survival analysis, (i.e., the installation of non-vulnerable versions contributes to the vulnerability decay), we observe that the vulnerable host population starts decaying almost as soon as a vulnerable version is released, as some users revert to older versions. In this case, 94% of the clusters start decaying earlier than 30 days before the disclosure date. While this is a mechanism that could prevent zero-day exploits in the absence of a patch, users are not aware of the vulnerability before its public disclosure, and the vulnerability decay due to non-vulnerable versions is small.

3.6.3 Patches and Exploits

We now ask the question: *When attackers create exploits for these vulnerabilities, what percentage of the host population can they expect to find still vulnerable?* This has important security implications because exploit creators are in a race with the patch deployment: once the vulnerability is disclosed publicly, users and administrators will start taking steps to protect their systems.

We use the WINE-AV dataset to identify attacks for specific vulnerabilities. This dataset only covers 1.5 years of our observation period (see Table 3.1). After discarding vulnerabilities disclosed before the start of the WINE-AV dataset and intersecting the 244 exploits in WINE-AV with our vulnerability list, we are left with 13 exploits, each for a different vulnerability. We also add 50 vulnerabilities that have an exploit release date in EDB. For each of the 54 vulnerabilities in the union of these two sets, we extract the earliest record of the exploit in each of the two databases, and we determine the value of the survival function $S(t)$ on that date. In this analysis, we consider both types of death events, because installing non-vulnerable program versions also removes hosts from the population susceptible to these exploits.

Figure 3.7 illustrates the survival levels for these vulnerabilities; on the right of the X-axis 100% (1.00) of hosts remain vulnerable, and on the left there are no remaining vulnerable hosts. Each vulnerability is annotated with the number of days after disclosure when we observe the first record of the exploit. This *exploitation lag* is overestimated because publicizing an exploit against an unknown vulnerability amounts to a disclosure (for EDB) and because AV signatures that can detect the exploit are often deployed after disclosure (for WINE-AV). For example, while CVE-2011-0611, CVE-2011-0609, CVE-2010-3654, CVE-2010-2862, CVE-2010-1241, and CVE-2011-0618 are known to have been exploited in zero-day attacks [140, 141, 17], the exploitation lag is ≥ 0. In consequence, the vulnerability survival levels in Figure 3.7 must be interpreted as *lower bounds*, as the exploits were likely released prior to the first exploit records in WINE-AV and EDB.

Additionally, these results illustrate the opportunity for exploitation, rather

than a measurement of the successful attacks. Even if vulnerabilities remain unpatched, end-hosts may employ other defenses against exploitation, such as anti-virus and intrusion-detection products or mechanisms such as data execution prevention (DEP), address space layout randomization (ASLR), or sandboxing. Our goal is therefore to assess the effectiveness of vulnerability patching, by itself, in defending hosts from exploits.

All but one of the real-world exploits in WINE found more than 50% of hosts still vulnerable. Considering both databases, at the time of the earliest exploit record between 11% and 100% of the vulnerable population remained exploitable, and the median survival rate was 86%. As this survival rate represents a lower bound, *the median fraction of hosts patched when exploits are released is at most 14%.*

3.6.4 Opportunities for Patch-Based Exploit Generation

Attackers have an additional trump card in the race with the patch deployment: once a patch is released, it may be used to derive working exploits automatically, by identifying the sanitization checks added in the patched version and generating inputs that fail the check [129]. While prior work has emphasized the window of opportunity for patch-based exploit generation provided by slow patch deployment, it is interesting to observe that in some cases a vulnerability may affect more than one application. For example, Adobe Flash vulnerability CVE-2011-0611 affected both the Flash Player and Acrobat Reader (which includes a library allowing it to play Flash objects embedded in PDF documents). For Reader, the patching started 6 days later than for Flash, giving attackers nearly one week to create an exploit based on the Flash patch.

In our dataset, 80 vulnerabilities affect common code shared by two applications. The time between patch releases ranges from 0 (when both patches are released on the same date, which occurs for 7 vulnerabilities) to 118 days, with a median of 11 days. 3 Flash vulnerabilities also affect Adobe Reader (as in the case of CVE-2011-0611 described above), and the patches for Flash were released before or on the same day as the Reader patches. 7 vulnerabilities affect the Chrome and Safari browsers, which are based on the WebKit rendering engine; in one case, the Safari patch was released first, and in the other cases the Chrome patch was released first. 1 vulnerability affects Chrome and Firefox, which use the Angle[5] library for hardware-accelerated graphics, and in this case the Chrome patch was released first. Finally, 69 vulnerabilities affect Firefox and Thunderbird, which share multiple Mozilla libraries, and in all these cases the Firefox patches were released before or on the same day as the Thunderbird patches.

These delays in patching all the applications affected by shared vulnerabilities may facilitate the creation of exploits using patch-based exploit generation

[5] https://code.google.com/p/angleproject/

techniques [129]. In practice, however, the attacker may experience additional delays in acquiring the patch, testing the exploit and delivering it to the intended target. We therefore compare the time between patch releases with our empirical observations of the exploitation lag, illustrated in Figure 3.7. While the median exploitation lag is slightly longer (19 days) the two distributions largely overlap, as shown in Figure 3.8. In consequence, the time between patch releases for applications sharing code is comparable to the typical time that attackers need to create exploits in practice. This is a serious threat because an exploit derived from the first patch to be released is essentially a *zero-day exploit for the other applications*, as long as patches for these applications remain unavailable.

3.6.5 Impact of Multiple Installations on Patch Deployment

While in Section 3.6.3 we investigated the threat presented by the time elapsed between patch releases for applications with common vulnerabilities, we now ask the question *how do multiple versions of an application, installed in parallel, impact the patch deployment process?* Failing to patch all the installed versions leaves the host exposed to the attack described in Section 3.3.1. This is a common situation: for example, our analysis suggests that 50% of WINE users who have installed the Flash plugin have Adobe Air installed as well.

Figure 3.9 compares the patching of CVE-2011-0611 in Flash and Adobe Reader. This vulnerability is a known zero-day attack, discovered on 12 April 2011. For Flash, the patching started 3 days after disclosure. The patching rate was high, initially, followed by a drop and then by a second wave of patching activity (suggested by the inflection in the curve at $t = 43$ days). The second wave started on 25 May 2011, when the vulnerability survival was at 86%. According to analyst reports, a surge of attacks exploiting the vulnerability started on 15 May 2011 [142]. The second wave of patching eventually subsided, and this vulnerability did not reach 50% completion before the end of our observation period. Perhaps because of this reason, CVE-2011-0611 was used in 30% of spear phishing attacks in 2011 [143]. In contrast, for Reader the patching started later, 9 days after disclosure, after CVE-2011-0611 was bundled with another vulnerability in a patch). Nevertheless, the patch deployment occurred faster and reached 50% completion after 152 days. This highlights the fact that, in general, Adobe Reader patched faster than Flash Player, as Table 3.3 indicates.

This highlights the magnitude of the security threat presented by keeping multiple versions installed on a host, without patching all of them. Even if some of these installations are used infrequently, the attacker may still be able to invoke them, as demonstrated in Section 3.3.1. Moreover, a user patching the frequently used installation in response to news of an exploit active in the wild may unknowingly remain vulnerable to attacks against the other versions that remain installed.

3.6.6 Patching Milestones

In this section we ask the question: *How quickly can we deploy patches on all the vulnerable hosts?* Figure 3.10 illustrates the distribution for the median time-to-patch (t_m), and the time needed to patch 90% and 95% of the vulnerable hosts ($t_{90\%}$ and $t_{95\%}$, respectively), across all vulnerability clusters for the 10 applications. We find that the rate of updating is high in the beginning: 5% of clusters reach t_m within 7 days, 29% of clusters reach it within 30 days and 54% of clusters reach it within 100 days. Figure 3.10a suggests that the median time-to-patch distribution is decreasing, with a long tail, as most vulnerability clusters reach this milestone within a few weeks, but a few clusters need as much as three years to reach it.

In contrast, the distribution of the time needed to reach high levels of update completion seems to have two modes, which are not visible in the distribution of t_m. The bimodal pattern starts taking shape for $t_{90\%}$ (Figure 3.10b) and is clear for $t_{95\%}$ (Figure 3.10c). We observe that the first mode stretches 150 days after the start of patching, suggesting that, even for the vulnerabilities that exhibit the highest updating rate, the median time to patch may be up to several months.

3.6.7 Human Factors Affecting the Update Deployment

Finally, we analyze whether specific user profiles and automated patching mechanisms have a positive effect on how fast the vulnerable applications are patched. To this end, we first define three user categories that are presumably more security-aware than common users: professionals, software developers, and security analysts. We classify WINE hosts into these 3 categories by first assigning a set of applications to each category and then checking which hosts have installed some of these applications. A host can belong to multiple categories. For professionals we check for the existence of applications signed, among others, by SAP, EMC, Sage Software, and Citrix. For software developers we check for software development applications (Visual Studio, Eclipse, NetBeans, JDK, Python) and version control systems (SVN, Mercurial, Git). For security analysts, we check for reverse engineering (IdaPro), network analysis (Wireshark), and forensics tools (Encase, Volatility, NetworkMiner). Using these simple heuristics, we identify 112,641 professionals, 32,978 software developers, and 369 security analysts from our dataset.

We perform survival analysis for each user category separately to obtain the median time-to-patch. Table 3.4 presents the results for Adobe Reader, Flash, Firefox, and the mean for the 10 applications. We focus on these three applications because they are popular and because the update mechanisms used in most of the versions in our study were manual and, therefore, required user interaction. In addition, we are interested in checking if automated update mechanisms improve the success of the patching process, and these three applications started

Table 3.4: Number of days needed to patch 50% of vulnerable hosts, for different user profiles and update mechanisms.

Categories	**Median time-to-patch (% reached)**			
	All	**Reader**	**Flash**	**Firefox**
Professionals	30 (79%)	103 (90%)	201 (73%)	25 (92%)
Software Developers	24 (80%)	68 (90%)	114 (86%)	23 (90%)
Security Analyst	18 (93%)	27 (87%)	51 (91%)	13 (89%)
All users	45 (78%)	188 (90%)	247 (60%)	36 (91%)
Silent Updates	27 (78%)	62 (90%)	107 (86%)	20 (89%)
Manual Updates	41 (78%)	97 (90%)	158 (81%)	26 (88%)

using automated updates in 2012. As shown in Table 3.4, all 3 user categories reach 50% patching faster than the common category encompassing all users. This indicates that these categories react to patch releases more responsibly than the average user. Among our three categories, the security analysts patch fastest, with a patching rate almost three times higher than the general-user category.

The bottom of Table 3.4 we compare manual and automated updating shows the survival analysis results when splitting the program versions into those with manual and automated patching, based on when silent updates were introduced by the program. As expected, automated update mechanisms significantly increase patching deployment, improving security.

3.7 Related Work

Several researchers [13, 15, 16] have proposed vulnerability lifecycle models, without exploring the patch deployment phase in as much detail as we do. Prior work on manual patch deployment has showed that user-initiated patches [47, 48, 49, 50] occur in bursts, leaving many hosts vulnerable after the fixing activity subsides. After the outbreak of the Code Red worm, Moore et. at [47] probed random daily samples of the host population originally infected and found a slow patching rate for the IIS vulnerability that allowed the worm to propagate, with a wave of intense patching activity two weeks later when Code Red began to spread again. Rescorla [48] studied a 2002 OpenSSL vulnerability and observed two waves of patching: one in response to the vulnerability disclosure and one after the release of the Slapper worm that exploited the vulnerability. Each fixing wave was relatively fast, with most patching activity occurring within two weeks and almost none after one month.

Rescorla [48] modeled vulnerability patching as an exponential decay process with decay rate 0.11, which corresponds to a half-life of 6.3 days. Ramos [49] analyzed data collected by Qualys through 30 million IP scans and also reported

a general pattern of exponential fixing for remotely-exploitable vulnerabilities, with a half-life of 20-30 days. However, patches released on an irregular schedule had a slower patching rate, and some do not show a decline at all. While t_m for the applications employing silent update mechanisms and for two other applications (Firefox and Thunderbird) is approximately in the same range with these results, for the rest of the applications in our study t_m exceeds 3 months.

Yilek et al. [50] collected daily scans of over 50,000 SSL/TLS Web servers, in order to analyze the reaction to a 2008 key generation vulnerability in the Debian Linux version of OpenSSL. The fixing pattern for this vulnerability had a long and flat curve, driven by the baseline rate of certificate expiration, with an accelerated patch rate in the first 30 days and with significant levels of fixing (linked to activity by certification authorities, IPSes and large Web sites) as far out as six months. 30% of hosts remained vulnerable six months after the disclosure of the vulnerability. Durumeric et al. [18] compared these results with measurement of the recent Heartbleed vulnerability in OpenSSL and showed that in this case the patching occurred faster, but that, nevertheless, more than 50% of the affected servers remained vulnerable after three months.

While the references discussed above considered manual patching mechanisms, the rate of updating is considerably higher for systems that employ automated updates. Gkantsidis et al. [51] analyzed the queries received from 300 million users of Windows Update and concluded that 90% of users are fully updated with all the previous patches (in contrast to fewer than 5%, before automated updates were turned on by default), and that, after a patch is released, 80% of users receive it within 24 hours. Dübendorfer et al. [52] analyzed the User-Agent strings recorded in HTTP requests made to Google's distributed Web servers, and reported that, within 21 days after the release of a new version of the Chrome Web browser, 97% of active browser instances are updated to the new version (in contrast to 85% for Firefox, 53% for Safari and 24% for Opera). This can be explained by the fact that Chrome employs a *silent update* mechanism, which patches vulnerabilities automatically, without user interaction, and which cannot be disabled by the user. These results cover only instances of the application that were active at the time of the analysis. In contrast, we study multiple applications, including 500 different versions of Chrome, and we analyze data collected over a period of 5 years from 8.4 million hosts, covering applications that are installed but seldom used. Our findings are significantly different; for example, 447 days are needed to patch 95% of Chrome's vulnerable host population.

Despite these improvements in software updating, many vulnerabilities remain unpatched for long periods of time. Frei et al. [53] showed that 50% of Windows users were exposed to 297 vulnerabilities in a year and that a typical Windows user must manage 14 update mechanisms (one for the operating system and 13 for the other software installed) to keep the host fully patched. Bilge et al. [17] analyzed the data in WINE to identify zero-day attacks that exploited vulnerabilities disclosed between 2008–2011, and observed that 58% of the anti-

virus signatures detecting these exploits were still active in 2012.

3.8 Discussion

Recent empirical measurements suggest that only 15% of the known vulnerabilities are exploited in real-world attacks and that this ratio is decreasing [144]. Our findings in this paper provide insight into the continued effectiveness of vulnerability exploits reported in prior work [123]. For example, when exploits are released is at most 14%, which suggests that vulnerability exploits work quite well when they are released—even if they are not zero-day exploits. Additionally, because vulnerabilities have a non-linear lifecycle, where the vulnerable code may exist in multiple instances on a host or may be re-introduced by the installation of a different application, releasing and deploying the vulnerability patch does not always provide immunity to exploits. In the remainder of this section, we suggest several improvements to software updating mechanisms, to risk assessment frameworks, and to vulnerability databases, in order to address these problems.

Handling inactive applications. Inactive applications represent a significant threat if an attacker is able to invoke (or convince the user to invoke) installed, but forgotten, programs that remain vulnerable. In Section 3.3.1 we present two attacks against such inactive versions, and we find that it is common for users to have multiple installations of an application (for example, 50% of the hosts in our study have Adobe Flash installed both as a browser plugin and as part of the Adobe Air runtime). We therefore recommend the developers of software updating systems to check the hard drive for all the installed versions and to implement the updater as a background service, which runs independently of the application and which automatically downloads and updates all the vulnerable software detected.

Consolidating patch mechanisms. We find that, when vendors use multiple patch dissemination mechanisms for common vulnerabilities (e.g., for Acrobat Reader and Flash Player), some applications may remain vulnerable after the user believes she has installed the patch. This highlights a bigger problem with code shared among multiple applications. Many applications include third-party software components—both commercial and open source—and they disseminate security patches for these components independently of each other. In this model, even if the developer notifies the application vendor of the vulnerability and provides the patch, the vendors must integrate these patches in their development and testing cycle, often resulting in delays. For example, vulnerabilities in the Android kernel and device drivers have a median time-to-patch of 30–40 weeks, as the mobile device manufacturers are responsible for delivering the patches to the end-users [145]. Unpatched code clones are also common in OS code deployed in

the field [146]. Our measurements suggest that, when a vulnerability affects several applications, the patches are usually released at different times—even when the two applications are developed by the same organization—and that the time between patch releases gives enough window for attackers to take advantage of patch-based exploit generation techniques. In consequence, we recommend that patch dissemination mechanisms be consolidated, with all software vendors using one or a few shared channels. However, in ecosystems without a centralized software delivery mechanism, e.g., workstations and embedded devices, consolidation may be more difficult to achieve. Moreover, coordinating patch releases among multiple vendors raises an interesting ethical question: when one vendor is not ready to release the patch because of insufficient test coverage, is it better to delay the release (in order to prevent patch-based exploit generation) or to release the patch independently (in order to stop other exploits)?

Updates for libraries. A more radical approach would be to make library maintainers responsible for disseminating security patches for their own code, independently of the applications that import these libraries. This would prevent patching delays, but it would introduce the risk of breaking the application's dependencies if the library is not adequately tested with all the libraries that use it. This risk could be minimized by recording the behavior of the patch in different user environments and making deployment decisions accordingly [147]. These challenges emphasize the need for further research on software updating mechanisms.

Program versioning. We observe that identifying all vulnerabilities affecting a host is challenging due to the various vendor approaches for maintaining program and file versions. We conclude that software vendors should have at least one filename in a product whose file version is updated for each program version. Otherwise, it becomes very complicated for a user or analyst to identify the installed program version. This situation happens with popular programs such as Internet Explorer and in some versions of other programs like Adobe Reader. This situation also creates issues for the vendor, e.g., when clicking the "About" menu entry in Reader 9.4.4 it would still claim it was an earlier version. So, even if it is possible to patch a vulnerability by modifying only a secondary file, we believe vendors should still update the main executable or library to reflect a program version upgrade. This would also establish a total ordering of how incremental patches should be applied, simplifying the tracking of dependencies.

3.8.1 Improving Security Risk Assessment

This work contributes new vulnerability metrics that can complement the Common Vulnerability Scoring System (CVSS), currently the main risk assessment metric for software vulnerabilities [148]. The CVSS score captures exploitability

and impact of a vulnerability, but it does not address the vulnerable population size, the patching delay for the vulnerability, the patching rate, and the updating mechanisms of the vulnerable program. Enhancing CVSS with the above information would provide stronger risk assessment, enabling system administrators to implement policies such as subjecting hosts patched infrequently to higher scrutiny, prioritizing patching of vulnerabilities with larger vulnerable populations, or scanning more frequently for vulnerabilities as the updating rate decreases. These metrics have several potential applications:

- *Customizing security:* Security vendors could customize the configuration of their security tools according to the risk profile of a user. For example, they could enable more expensive, but also more accurate, components of their product for users at higher risk.
- *Improving software whitelists:* One issue when using software whitelists for malware protection [149] is that newly developed benign software would be considered malicious since it is not yet known to the static whitelist. Our software developer profiles can reduce false positives by not flagging as malicious new executables from developers that have not been externally obtained (e.g., from the Internet, USB, or optical disk).
- *Educating the users:* Many users may be interested in receiving feedback about their current security risk profile and suggestions to improve it.
- *Cyber-insurance:* The new vulnerability metrics and risk profiles could significantly improve the risk assessment methods adopted by insurance companies that currently rely on questionnaires to understand how specific security measures are taken by a given organization or individual to establish their policy cost.

3.9 Clean NVD

An important challenge to the automatic generation of vulnerability reports are NVD inaccuracies. We have spent significant effort on a `Clean NVD` subproject, whose goal is identifying and reporting discrepancies in NVD vulnerability entries. This section briefly introduces our efforts. We have contacted the NVD managers about these issues and have set up a website to detail our findings and help track fixes to NVD at http://clean-nvd.com/.

The 3 main NVD inaccuracies we found are programs with vulnerable product lines rather than program versions, and *missing* and *extraneous* vulnerable versions. Surprisingly, we found that vulnerabilities in popular programs such as Microsoft Word and Internet Explorer only contain vulnerable program lines. For example, NVD states that CVE-2009-3135 affects Word 2002 and 2003, but those are product lines rather than program versions. Note that these programs

do not have program versions proper, i.e., there is no such thing as Word 2003.1 and Word 2003.2. Instead, Microsoft issues patches to those programs that only update file versions. Currently, an analyst/user cannot use the NVD data to determine if the Word version installed on a host is vulnerable. We believe that NVD should add the specific file versions (for a version-specific filename such as msword.exe) that patched the vulnerability. To validate that this is possible, we crawled the Word security advisories in MSDN for 2008-2012, which contain the CVEs fixed and link to pages describing the specific msword.exe file versions released with the advisory. We have used that information to build the Word version mapping and cluster files needed for the analysis.

A vulnerability in NVD may miss some vulnerable program versions. These are likely due to errors when manually entering the data into the database as the missing versions typically appear in the textual vulnerability descriptions and the vendor's advisories. A vulnerability may also contain extraneous vulnerable versions. We find two reasons for these: errors when inserting the data and vendors conservatively deciding which program versions are vulnerable. Specifically, vendors seem to often determine the last vulnerable version in a product line and then simply consider all prior versions in the line vulnerable, without actually testing if they are indeed vulnerable. This cuts vulnerability testing expenses and helps pushing users to the latest version.

Finding errors. To automatically identify missing and extraneous vulnerable versions in NVD we use two approaches. For Firefox and Thunderbird, we have built scripts that collect Mozilla security advisories, parse them to extract the patched versions in each line, and compare them with the last vulnerable versions in NVD. In addition, we have developed a generic differential testing approach to compare the vulnerable ranges in the textual vulnerability description with the vulnerable versions in the NVD XML dumps. To automate this process we use natural language processing (NLP) techniques to extract the vulnerable ranges from the textual description. We have applied the differential testing approach to the 10 programs finding 608 vulnerabilities with discrepancies, which we exclude from the analysis. While these approaches are automated and identify a large number of NVD errors, they may not find all errors. Thus, we also manually check one vulnerability in each remaining cluster comparing the version list against the vendor's advisories and the disclosure dates against the release dates of the patched versions.

3.10 Conclusion

We investigate the patch deployment process for 1,593 vulnerabilities from 10 client-side applications. We analyze field data collected on 8.4 million hosts over an observation period of 5 years, made available through the WINE platform from Symantec. We show two attacks made possible by the fact that multiple versions of the same program may be installed on the system or that the same library may be distributed with different software. We find that the median fraction of vulnerable hosts patched when exploits are released is at most 14%. For most vulnerabilities, patching starts around the disclosure date, but the patch mechanism has an important impact on the rate of patch deployment. For example, applications updated automatically have a median time-to-patch 1.5 times lower than applications that require the user to apply patches manually. However, only 28% of the patches in our study reach 95% of the vulnerable hosts during our observation period. This suggests that there are additional factors that influence the patch deployment process. In particular, users have an important impact on the patch deployment process, as security analysts and software developers deploy patches faster than the general user population. Most of the vulnerable hosts remain exposed when exploits are released in the wild. Our findings will enable system administrators and security analysts to assess the the risks associated with vulnerabilities by taking into account the milestones in the vulnerability lifetime, such as the patching delay and the median time-to-patch.

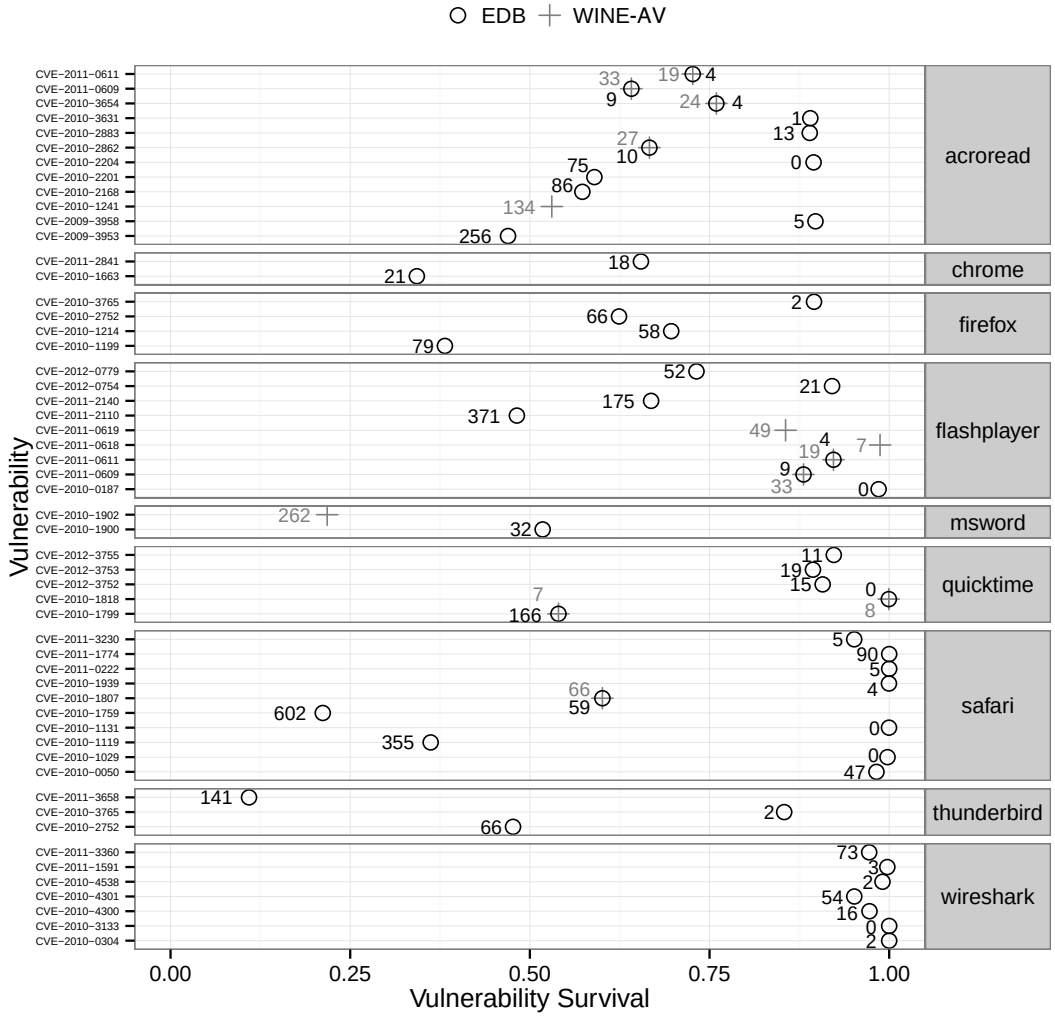


Figure 3.7: Vulnerability survival (1 = all hosts vulnerable) at exploitation time. Data points are annotated with estimations of the exploitation lag: the number of days after disclosure when the the exploits were publicized (for EDB) and when the first detections occurred (for WINE-AV).

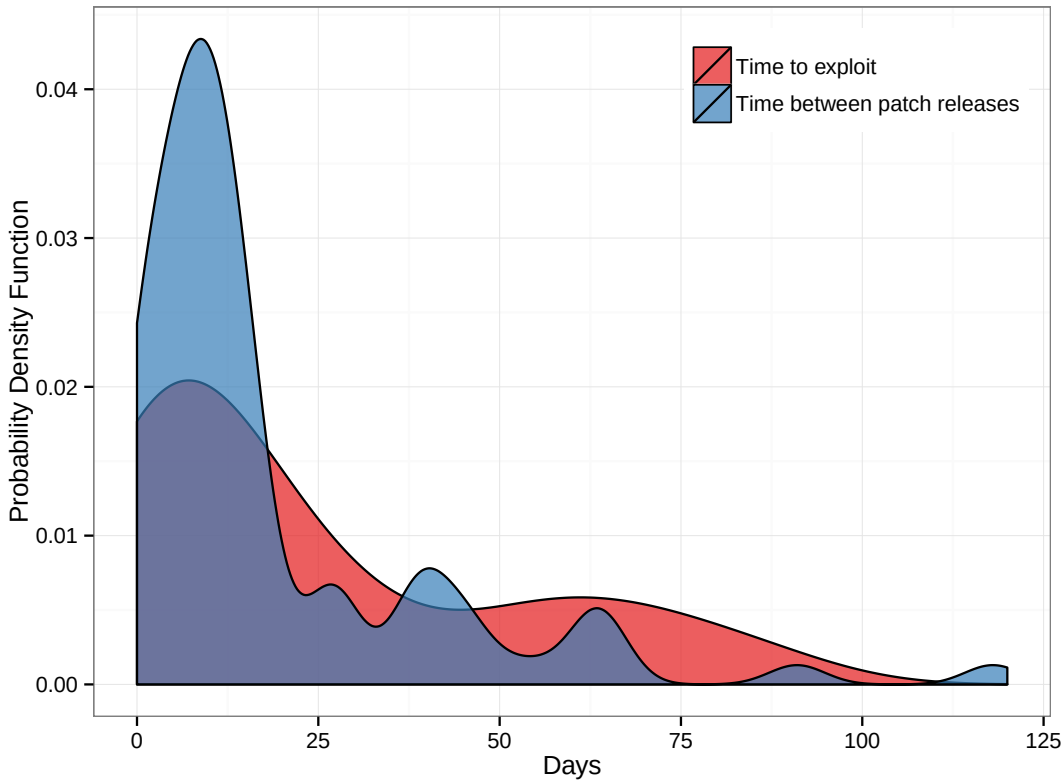


Figure 3.8: Distributions of the time between patch releases for vulnerabilities affecting multiple applications and the time needed to exploit vulnerabilities.

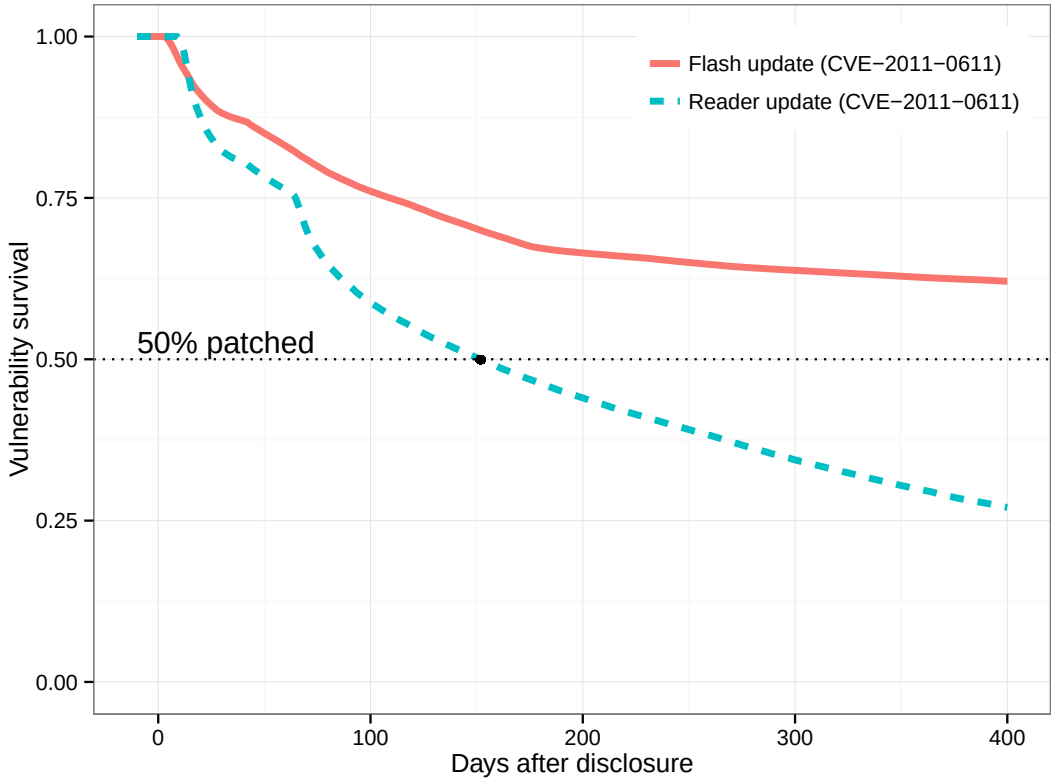
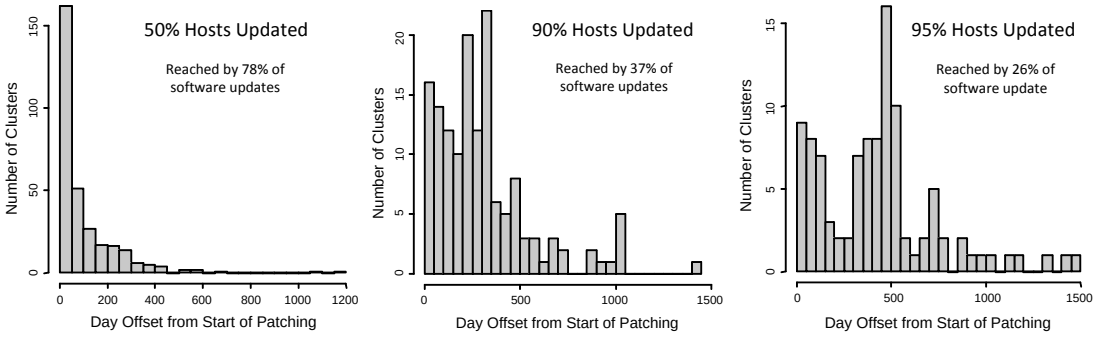


Figure 3.9: Patching of one vulnerability from the Flash library, in the standalone installation and in Adobe Reader.



(a) Time to patch 50% of hosts. (b) Time to patch 90% of hosts. (c) Time to patch 95% of hosts.

Figure 3.10: Distribution of the time needed to reach three patch deployment milestones. Each bar corresponds to a 50-day window.

Part II

Active Detection of Malicious Servers Infrastructures

4

The MALICIA Dataset: Identification and Analysis of Drive-by Dowload Operations

4.1 Preamble

This chapter reproduces the content of two papers: "Driving in the Cloud: An Analysis of Drive-by Download Operations and Abuse Reporting" published at DIMVA 2013 and "The MALICIA Dataset: Identification and Analysis of Drive-by Download Operations". The papers present a study of the Drive-by Download ecosystem with a focus on exploit servers. Both papers have been realized with the contribution of other people from the IMDEA Software institute, in both works Antonio Nappa has been the leading author.

4.2 Introduction

Drive-by downloads have become the preferred distribution vector for many malware families [56, 28]. A major contributing factor has been the proliferation of specialized underground services such as exploit kits and exploitation-as-a-service that make it easy for miscreants to build their own drive-by distribution infrastructure [28]. In this ecosystem many organizations license the same exploit kit, essentially running the same software in their exploit servers (upgrades are free for the duration of the license and promptly applied). This makes it challenging to identify which drive-by operation a exploit server belongs to. This is fundamental for understanding how many servers an operation uses, which operations are more prevalent, how long operations last, and for prioritizing takedown efforts and law enforcement investigations.

A drive-by operation is a group of exploit servers managed by the same organization, and used to distribute malware families the organization monetizes. An operation may distribute multiple malware families, e.g., for different monetization schemes. A malware family may also be distributed by different operations.

For example, malware kits such as zbot or spyeye are distributed by many organizations building their own botnets. And, pay-per-install (PPI) affiliate programs give each affiliate organization a customized version of the same malware to distribute [73].

In this paper, we propose a technique to identify exploit servers managed by the same organization, even when those exploit servers may be running the same software (i.e., exploit kit). Our technique enables reducing the large number of individual exploit servers discovered daily, to a smaller, more manageable, number of operations. Our intuition is that servers managed by the same organization are likely to share parts of their configuration. Thus, when we find two servers sharing configuration (e.g., pointed by the same domain, using similar URLs, or distributing the same malware) this is a strong indication of both being managed by the same organization. To collect the configuration information we track exploit servers over time and classify the malware they distribute.

Our clustering can be used by law enforcement during the pre-warrant (plain view) phase of a criminal investigation [150]. During this phase, criminal activity is monitored and targets of importance are selected among suspects. The goal of the plain view phase is gathering enough evidence to obtain a magistrate-issued warrant for the ISPs and hosting providers for the servers in the operation.

Our clustering can identify large operations that use multiple servers, rank operations by importance, and help understanding whether they belong to individual owners or to distribution services.

Results. Our data collection has been running for one year and has tracked 502 exploit servers. Our analysis reveals two types of drive-by operations. Two thirds of the operations use a single server and are short-lived. The other third of the operations use multiple servers to increase their lifetime. These multi-server operations have a median lifetime of 5.5 days and some live for several months, despite individual exploit servers living a median of 16 hours. Miscreants are able to run long-lived operations by relying on pools of exploit servers, replacing dead servers with clones. We also observe a few short-lived multi-server operations (lasting less than a day) that use over a dozen exploit servers in parallel to achieve a burst of installations. While most short-lived operations distribute a single malware family, we observe multi-server operations often distributing more than one. Furthermore, we identify two PPI affiliate programs (the *winwebsec* fake antivirus and the *zeroaccess* botnet) that manage exploit servers so that their affiliates can convert their traffic into installations, without investing in their own drive-by infrastructure.

We also analyze the hosting infrastructure. We find that to sustain long-lived multi-server operations, in the presence of increasing pressure from defenders, miscreants are turning to the cloud. Over 60% of the exploit servers belong to cloud hosting services. Long-lived operations are using pools of exploit servers, distributed among different countries and autonomous systems (ASes) for re-

siliency, replacing dead servers with clones. Miscreants are taking advantage of a booming cloud hosting services market where hosting is cheap, i.e., virtual private servers (VPS) start at $10 per month and dedicated servers at $60 [151]. These services are easy to contract (e.g., automated sign-up procedures requiring only a valid credit card) and short leases are available (e.g., daily billing) so that the investment loss if the exploit server is taken down can be less than a dollar. In this environment, cloud hosting providers have started reporting that 50% of their automated VPS subscriptions are being abused [152].

In addition, we analyze the exploits used by the monitored exploit servers. In particular, we measure the exploit polymorphism, which as far as we know has not previously been studied. Our results show that different exploit types (e.g., Java, PDF, Windows fonts) exhibit different repacking rates. For example, some exploit kits like BlackHole 2 have integrated automated repacking of PDF exploits for each exploitation attempt. On the opposite side, font exploits are not being repacked likely due to the lack of repacking tools. Exploits for Java vulnerabilities are repacked at a lower and varying frequency indicating that the repacking tool is likely invoked manually and not fully integrated in the exploit kit.

To understand how difficult is to take down exploit servers, we issue abuse reports for 19 long-lived servers. We analyze the abuse reporting process, as well as the interaction with the ISPs and hosting providers. We use our infrastructure to monitor the result of the report (i.e., whether the server is taken down). The results are disheartening. Over 61% of the reports do not produce a reply and the average life of a exploit server after it is reported is 4.3 days.

Our work reveals a growing problem for the take down of drive-by download operations. While miscreants enjoy a booming hosting market that enables them to set up new exploit servers quickly, defenders face a tough time reporting abuse due to uncooperative providers and inadequate business procedures. Takedown procedures need to be rethought. There is a need to raise the cost for miscreants of a server being taken down, monitor short-lived VPS subscriptions, and shift the focus to prosecuting the organizations that run the operations, as well as the organizations behind specialized underground services supporting the ecosystem.

Finally, this work has produced a dataset that includes the malware binaries we collected, the metadata of when and how it was collected, and the malware classification results. To foster further research we make our Malicia dataset available to other researchers [153].

Contributions:

- We propose a technique to identify drive-by operations by grouping exploit servers based on their configuration and the malware they distribute.
- We report on aspects of drive-by operations such as the number of servers

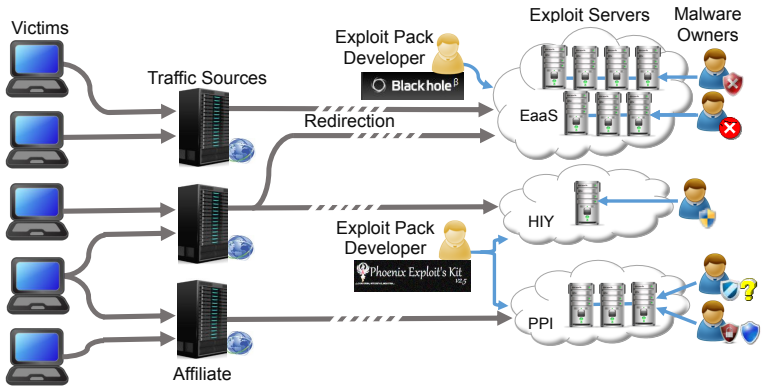


Figure 4.1: Exploit kit ecosystem.

they use, their hosting infrastructure, their lifetime, and the malware families they distribute.

- We measure the polymorphism of the exploits served by the monitored exploit servers.
- We analyze the abuse reporting procedure by sending reports on exploit servers.
- We build a dataset with the collected malware, their classification, and associated metadata. We make this dataset available to other researchers.

4.3 Overview

Drive-by downloads are a popular malware distribution vector. To distribute its products over drive-by downloads a malware owner needs 3 items: exploitation software, servers, and traffic. To facilitate the process, 3 specialized services exist (Figure 4.1). A malware owner can license an exploit kit (host-it-yourself), rent a exploit server with the exploit kit installed (exploitation-as-a-service), or simply buy installs from a pay-per-install service that provides the exploit server and the traffic.

4.3.1 Roles

The exploit kit ecosystem has four main roles: *malware owner*, *exploit kit developer*, *exploit server owner*, and *exploit server manager*. Exploit kit developers offer a software kit including a set of exploits for different platforms (i.e., combination of browser, browser plugins, and OS), web pages to exploit visitors and

drop files on their hosts, a database to store all information, and an administration panel to configure the functionality and provide installation statistics. Exploit kits are offered through two licensing models: host-it-yourself (HIY) and exploitation-as-a-service (EaaS). In both models access to the exploit kit (or server) is time-limited and clients obtain free software updates during this time. Also in both models the client provides the traffic as well as a domain name to which the kit is linked. The client pays for domain changes (e.g., $20 for BlackHole [154]) unless it buys a more expensive multi-domain license.

The exploit server provider is the entity that contracts the hosting and Internet connectivity for the exploit server. It can be the malware owner in the HIY model or the exploit kit developer in EaaS. Exploit kits are designed to be installed on a single host that contains the exploits, malware files, configuration, and statistics. Thus, exploit servers are typically dedicated, rather than compromised, hosts. A robust hosting infrastructure is needed to launch long-lived operations as most exploit servers are short-lived. Exploit server providers acquire a pool of servers and favor hosting providers and ISPs where exploit servers live longer, i.e., those that are not diligent in handling abuse reports, or those who offer a specific abuse protection policy.

The exploit server manager is the entity that manages the exploit server through its administration panel. The manager is a client of the exploit kit developer and corresponds to the malware owner or a PPI service. PPI affiliate programs may run their own exploit server providing each affiliate with a unique affiliate URL. Affiliates credit installs by installing their affiliate-specific malware executable in hosts they have compromised, or by sending traffic to their affiliate URL, which would in turn install their affiliate-specific malware if exploitation succeeds. In these programs, affiliates can point their traffic sources to their affiliate URL in the program's exploit server or to their own exploit server. The latter requires investment but has two advantages: they can configure their exploit server to install other malware on the compromised machine, and they can avoid the affiliate program skimming part of their traffic for their own purposes. Our operation analysis reveals both exploit servers managed by individual affiliates and by PPI affiliate programs.

4.3.2 Exploit Server Clustering

In this work we cluster exploit servers under the same management using information about the server's configuration. Two servers sharing configuration, (e.g., pointed by the same domain, hosting similar URLs, using the same exploits, or distributing the same malware) indicates that they may be managed by the same organization. We focus on server configuration because the software is identical in many exploit servers since kit updates are free and promptly applied (19 days after the launch of BlackHole 2.0 we could no longer find any live BlackHole 1.x servers).

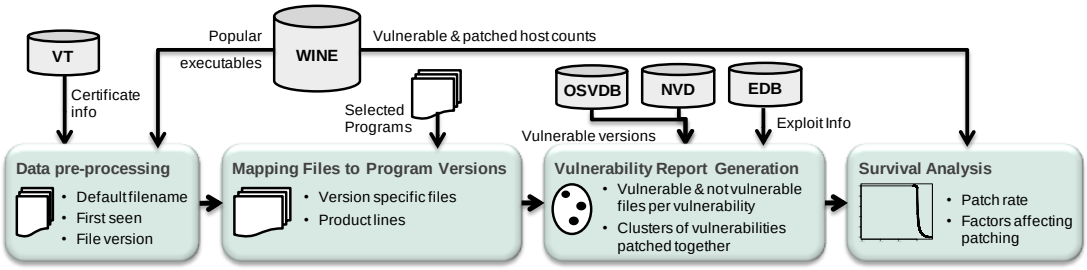


Figure 4.2: Architecture of our milking, classification, and analysis.

Our results show that attackers are using pools of exploit servers to sustain malware distribution operations over time. Such operations often run multiple exploit servers simultaneously behind a Traffic Direction System (TDS) that distributes the incoming traffic among them [155]. When one of their exploit servers goes down, attackers replace it with another server from their pool. The intuition behind our clustering is that in this environment new exploit servers often reuse the configuration of older servers. This happens because, when installing a new exploit server, the attacker simply clones an existing server including its configuration, e.g., by uploading to a new cloud hosting provider a previously created virtual machine image file where an exploit server is installed and configured.

4.4 Methodology

To collect the information needed to cluster servers into operations, we have built an infrastructure to track individual exploit servers over time, periodically collecting and classifying the malware they distribute. Our pipeline is described in Figure 4.2. We receive feeds of drive-by download URLs (Section 4.4.1), use honeyclients as well as specialized milkers to periodically collect the malware from the exploit servers those URLs direct to (Section 4.4.2), classify malware using icon information and behavioral reports obtained through execution in a contained environment (Section 4.4.3), store all information in a database, and use the collection and classification data for clustering exploit servers into operations (Section 4.5) and for abuse reporting (Section 4.6). An earlier version of the milking and classification components were used to collect the BlackHole/Phoenix feed in [28]. Since that work, we have upgraded those two components. This section describes their latest architecture, detailing the differences with [28].

4.4.1 Feeds

To identify exploit servers for the first time, we use two publicly available feeds: Malware Domain List (MDL) [156] and urlQuery [157]. MDL provides a public forum where contributors report and discuss malicious URLs. The reported URLs

are manually checked by volunteers. Once verified they are published through their webpage and feeds. urlQuery is an automatic service that receives URLs submitted by analysts and publishes the results of visiting those URLs on their webpage. We periodically scan the webpages of MDL and urlQuery for URLs matching our own regular expressions for the landing URLs of common exploit kits. The volume of URLs in urlQuery is much larger than in MDL, but the probability of finding a live exploit server is larger in MDL because URLs in urlQuery are not verified to be malicious and URLs long dead are often re-reported. We selected these feeds based on their public availability but the infrastructure is designed to work with any URL feed.

4.4.2 Milking

Our milking component differs from the one used to collect the BlackHole/Phoenix feed in [28] in that it identifies an exploit server by its *landing IP*, i.e., the IP address hosting the landing URL, which provides the functionality (typically some obfuscated JavaScript code) to select the appropriate exploits for the victim's platform. In [28] we identified exploit servers by the domain in their URLs. This was problematic because a large number of domains often resolve to the IP address of an exploit server. When the domains in the URLs known to us went down, our milking would consider the exploit server dead, even if it could still be reachable through other domains. Currently, if all domains in the landing URLs of a server stop resolving, the milking queries two passive DNS services [158, 159] for alternative domains recently observed resolving to the exploit server. If no alternative domain is found, the milking continues using the landing IP.

In addition, our infrastructure now resolves the malicious domains periodically, which enables locating previously unknown exploit servers if the same domain is used to direct traffic to different exploit servers over time. This information is used in our clustering (Section 4.5). Using this separate resolution we discover an additional 69 servers not present in our feeds and another 30 servers before they appear in the feeds.

Another difference is that in [28] we relied exclusively on lightweight *specialized milkers*, i.e., custom HTTP clients that collect the malware from the exploit server, without running a browser or going through the exploitation process, simply by replaying a minimized network dialog of a successful exploitation. Our specialized milkers take advantage of the lack of replay protection in the BlackHole 1.x and Phoenix exploit kits. For details on the construction of the specialized milkers, we refer the interested reader to our technical report [160]. Since then we have added support for milking other exploit kits by adding *honeyclients*, i.e., Windows virtual machines installed with an unpatched browser (and browser plugins), which can be navigated to a given landing URL [54, 55].

Milking policy. Our milking tries to download malware from each known exploit server every hour on average. If no malware is collected, it increments a failure counter for the exploit server. If a failure counter reaches a threshold of 6, the state of its exploit server is changed to offline. If malware is collected before 6 hours, its failure counter is reset. This allows milking to continue through temporary failures of the exploit server. In addition, the milking component runs a separate process that checks if an offline exploit server has resurrected every 2 days. If three consecutive resurrection checks fail, the exploit server is considered dead. If the server has resurrected, its failure and resurrection counters are reset.

4.4.3 Malware Classification

Our classification process leverages icon information extracted statically from the binary as well as network traffic and screenshots obtained by executing the malware in a contained environment. Compared to the classification process in [28], we propose the automated clustering of malware icons and screenshots using perceptual hashing and have implemented a novel malware clustering and signature generation tool [40]. In addition, we evaluate the accuracy of the icon and screenshot clustering using a manually generated ground truth.

Additional behavioral features could be used for malware classification such as the system and API calls invoked by the malware during execution [68, 70]. We have not yet added those due to resource constraints. Their addition would likely increase our malware classification results. However, our current classification only leaves 2% of the malware samples unclassified, so the benefit would be limited.

Malware execution. We execute each binary in a virtualized environment designed to capture the network traffic the malware produces, and to take a screenshot of the guest VM at the end of the execution. We use Windows XP Service Pack 3 as the guest OS and only allow DNS traffic and HTTP connections to predefined benign sites to leave our contained environment. All other traffic is redirected to internal sinks.

Our classification applies automatic clustering techniques separately to the icons, the screenshots, and the network traffic. Then, an analyst manually refines the generic labels by comparing cluster behaviors against public reports. Finally, majority voting on the icon, screenshot, and network labels decides the family label for an executable.

Icons. A Windows executable can embed an icon in its header. Many malware families use icons because it makes them look benign and helps them establish a brand, which is important for some malware classes such as rogue software. Icons can be extracted statically from the binary without running the executable, so feature extraction is very efficient. A naive icon feature would simply compute

Figure 4.3: Icon polymorphism. Each pair of icons comes from two different files of the same family and is perceptually the same, although each icon has a different hash.

	Feature	Th.	Clus.	Precision	Recall	Time
I	avgHash	3	141	99.7%	91.3%	2.4s
I	pHash	13	149	99.8%	89.6%	1m17s
S	avgHash	1	64	99.2%	60.4%	7m37s
S	pHash	13	43	97.9%	52.4%	16m8s

Table 4.1: Clustering results for icons (top) and screenshots (bottom).

the hash of the icon. However, some malware families use polymorphism to obfuscate the icons in their executables, so that two malware of the same family have icons that look the same to the viewer, but have different hashes (Figure 4.3). To capture such polymorphic icon variants we use a perceptual hash function [161]. Perceptual hash functions are designed to produce similar hashes for images that are perceptually (i.e., visually) similar. A good perceptual hash returns similar hashes for two images if one is a version of the other that has suffered transformations such as scaling, aspect ratio changes, or small changes in brightness, contrast, and color. We have experimented with two different perceptual hash functions: average hash (avgHash) [39] and pHash [161]. We use the Hamming distance between hashes as our distance metric. If the distance is less than a threshold both icons are clustered together using the aggressive algorithm in Section 4.5.2. We experimentally select the threshold value for each feature. Table 4.1 (top) shows the clustering results on 5,777 icons compared with the manually generated ground truth, which an analyst produces by examining the clusters. The results show very good precision for both features and slightly better recall and runtime for avgHash.

Screenshots. The screenshot clustering also uses perceptual hashing. Table 4.1 (bottom) shows the clustering results on 9,896 screenshots. This time avgHash achieves better precision but slightly worse recall. The lower recall compared to the icons is due to the perceptual hashing distinguishing error windows that include different text or the icon of the executable. Still, the clustering reduces 9,896 screenshots to 50–60 clusters with very high precision, so it becomes easy

for an analyst to manually label the clusters. We ignore clusters that capture generic error windows or do not provide family information, e.g., the Windows firewall prompting the user to allow some unspecified traffic.

Network traffic. Our network clustering has evolved over time. In our earlier works [28, 44] the features used by the network clustering were the C&C domains contacted by the malware and tokens extracted from the URLs and User-Agent headers in HTTP requests sent by the malware. Those features did not handle malware using non-HTTP traffic for C&C communication. To support non-HTTP C&C traffic we developed a novel malware clustering and signature generation tool called FIRMA [40].

FIRMA takes as input a set of network traces obtained by running unlabeled malware binaries in a contained environment. It outputs: (1) a *clusters file* with a partition of the malware binaries that produced the network traces into *family clusters*, (2) a *signature file* with network signatures annotated with the family cluster they correspond to, and (3) an *endpoints file* with the C&C domains and IP addresses that the malware binaries in each family cluster contacted across the input network traces.

The network signatures produced by FIRMA enable classifying new executables from previously seen malware families efficiently, without having to rerun the clustering. More importantly, they can be used to identify malware-infected computers in network monitored by an IDS. FIRMA produces network signatures in the format supported by the open source signature-matching IDSes Snort [34] and Suricata [162], so that those popular IDSes can be used to match the signatures on network traffic.

In addition to not being limited to any type of traffic (e.g., HTTP) or specific fields (e.g., HTTP URL, User-Agent), other salient characteristics of FIRMA are that it produces a set of network signatures for each malware family (i.e., family cluster) and that the signature generation is protocol-aware. A set of signatures is produced for each family cluster so that each signature captures a different network behavior of the family, e.g., one signature for HTTP traffic and another for a binary C&C protocol or different signatures for different protocol messages. Using a protocol-aware traffic clustering and signature generation means that if the C&C traffic uses a known application protocol such as HTTP, IRC, or SMTP, the traffic is parsed into fields and the signatures capture that a token may be specific to a field and should only be matched on that field, increasing signature accuracy.

It is important to note that FIRMA supports obfuscated C&C protocols, commonly used by malware. In fact, much of the malware in our datasets uses such obfuscation. It is still possible to generate signatures on those because obfuscated C&C protocols often contain value invariants. For fully polymorphic C&C protocols [163], i.e., protocols where every single byte changes in each request making it impossible to generate a signature on the ciphertext, FIRMA

can still cluster malware samples from the same family if they reuse the same C&C domains or IP addresses. We refer the reader to the FIRMA paper for more details [40].

Once FIRMA outputs the family clusters, an analyst tries to map the automatically generated cluster labels (e.g., CLUSTER:A) to well-known traffic families (e.g., zbot). Overall, our classification produces traffic labels for 80% executables, icon labels for 51%, and screenshot labels for 21%. It classifies 93% of the executables, 5% fail to execute, and 2% remain unclassified.

4.4.4 Exploit Analysis

An exploit is an input to a program that takes advantage of a software vulnerability, which affects some versions of the program, to cause an unintended behavior of the program. An exploit is able to drive the program execution to the vulnerability point where the vulnerability can be triggered (e.g., an arithmetic operation) and to trigger the vulnerability condition (e.g., make the arithmetic operation overflow).

Similar to malware, exploits can also exhibit polymorphism. That is, the same exploit can be repacked multiple times to generate different *exploit instances*, i.e., variants of the same exploit. For example, a PDF exploit, in addition to the data that leads the program to the vulnerability point and triggers the vulnerability condition, may also contain much other data that can be modified to generate a different PDF file that still exploits the same vulnerability in the same way. In this example, we say that the old and new PDF files are two instances of the same exploit.

Exploit polymorphism can be introduced by exploit kit developers to bypass exploit signatures used by AV vendors. It also happens when the payload of the exploit (e.g., the code run after exploitation and the data used by that code) needs to be updated, without affecting how the vulnerability is exploited. For example, an exploit may embed URLs from where some malware is downloaded after exploitation. Those URLs may need to be changed periodically to bypass URL blacklists. Every time they are changed, a new exploit instance is created.

In this section we perform what we believe is the first analysis of how exploit polymorphism works in prevalent exploit kits. In particular, we examine how often different exploit types (e.g., Java, PDF, Windows fonts) are repacked to introduce polymorphism.

Exploit collection. To analyze the exploits used in the drive-by downloads we leverage that (starting on November 19, 2012) every time a honeyclient visits a potentially malicious URL, a trace of the complete network communication is stored. This produces 14,505 network traces, of which 19.7% correspond to exploitation attempts. The rest correspond to landing URLs that no longer lead to an exploit server, or exploit servers temporarily down. Of those exploitation

attempts 74.5% (14.7% of all network traces) lead to malware installation. The unsuccessful exploitation attempts may be due to unreliable exploits and, rarely, to the honeyclient execution finishing before malware was downloaded. The collected exploits come from the BlackHole 2.x (BH2) and CoolExploit exploit kits. For other exploit kits for which we use milkers, e.g., Phoenix and BlackHole 1.x, no network traces are generated.

The network traces contain multiple instances of the exploits that the exploit servers use against our honeyclients. Extracting and classifying all exploit instances in a network trace is challenging. We focus on exploits that are individual files, namely Java JAR archives, PDF documents, and EOT font files. There may be other types of exploits embedded in the landing page, e.g., JavaScript exploits. We do not attempt to identify and classify those because the landing page is obfuscated and while it could be deobfuscated, understanding what parts of it correspond to an exploit is difficult, as the landing page contains additional functionality like checking the victim's software configuration and redirecting the user after exploitation. We extract a total of 172 unique (by SHA1 hash) exploit instances: 114 JAR, 55 PDF, and 3 EOT files. The JAR and PDF files target vulnerabilities in the Java and Adobe Reader browser plugins respectively, and the EOT files target Windows vulnerabilities.

Exploit kits select which exploits to use from their exploit pool based on the software configuration of the victim, typically checked by some JavaScript code in the landing page. Thus, the exploits in our network traces depend on the configuration of our honeyclients. Our honeyclients have the same configuration: Windows XP SP3 with IE 7, Java JRE 6u14, and Adobe Reader 8. It is important to note that our goal is not to analyze all the exploits that a exploit kit contains. Our approach cannot see exploits that exploit servers do not serve against our honeyclients. For example, the BlackHole and CoolExploit kits are known to contain Flash exploits [164]. However, we do not observe Flash exploits in our network traces, likely because our honeyclients do not have Flash installed, thus the exploit kits do not serve those exploits. Rather, our goal is to analyze a novel question: whether exploit kits periodically repack their exploits and if so, how often.

Exploit classification. To verify that the extracted files are exploits we leverage two online services: VirusTotal [25] and Wepawet [165]. We use VirusTotal for JAR and EOT files and Wepawet for PDF files, as VirusTotal does not support PDF files. The 3 EOT files and 112/114 JAR files were present in VirusTotal and flagged by at least one AV vendor to be malicious. We submitted to VirusTotal the 2 JAR files that they had not seen yet, which were both flagged as malicious. We submit all 55 PDF documents to Wepawet, which identifies them as exploits. Thus, all extracted files are indeed exploits, which shows that the file extraction from the network traces does not identify benign files as exploits.

We classify the exploits according to the vulnerability they target. We first

Type	CVE	Instances	Collected	Ratio
JAR	CVE-2012-0507	88	2,419	3.5%
	CVE-2012-1723	15	203	7.4%
	CVE-2010-4476	3	103	2.9%
	Unknown	8	158	5.0%
	Subtotal	114	2,894	3.9%
EOT	CVE-2011-3402	3	777	0.3%
PDF	CVE-2009-0927	43	47	91.4%
	CVE-2010-0188	12	12	100%
	Subtotal	55	69	93.2%
Total		172	3,740	4.6%

Table 4.2: Exploit classification.

submit the exploit files to VirusTotal, which scans them using a variety of AV products and publishes the AV detection labels. We leverage the fact that some AV vendors include the CVE identifier of the vulnerability in their labels. For each AV that detects the file as malicious we check if a CVE identifier is included in the detection label using the following regular expression: CVE[-_]?([0-9]{4})[-_]+([0-9]{4}). Different AV vendors may not agree on the vulnerability targeted by the exploit.To address this issue, we use majority voting to select the most commonly seen CVE identifier for a exploit file. In case of a tie, we choose the most frequent CVE identifier between the tied ones, where frequency is computed across all exploits that present no tie. For the PDF files Wepawet outputs a single CVE for each exploit, so majority voting is not needed.

Using AV labels for classification is not ideal as it is well known that AV vendors focus on detection and their classification is often inaccurate [68]. Indeed, we found two exploit instances where AV vendors disagreed on the targeted vulnerability, and the majority voting was wrong. In this case it was possible to spot the error because the vulnerabilities output by the majority voting did not affect the Java versions run by our honeyclients. These examples illustrate the need for developing accurate exploit classification techniques.

Table 4.2 presents the exploit classification results. For each exploit type, it shows the CVE identifier of the different vulnerabilities or *Unknown* if no CVE identifiers were found. For each vulnerability, it presents the number of unique exploit files labeled with that vulnerability, the total number of times those instances appear in the network traces, and the ratio of both numbers as a percentage. The larger the percentage the more likely it is that when we collect a exploit instance for a vulnerability, we have never seen that instance before. The results show that Java vulnerabilities are most targeted against our honeyclients, followed by the CVE-2011-3402 TrueType font parsing vulnerability in Windows, with PDF exploits comprising only 1.8% of all exploits used against our honeyclients. The most common Java vulnerability is CVE-2012-0507, which

is targeted by 65% of the collected exploits. As explained earlier, if we configured our honeyclients differently, exploit kits may serve them other exploits.

Exploit polymorphism. We analyze whether exploit kits have integrated packing tools to automatically provide polymorphism for their exploits. For this, we analyze each exploit file type (JAR, PDF, EOT) separately. For PDF files, the ratio in Table 4.2 shows that 93.2% of the times that we collect a PDF exploit instance it is new, i.e., we have not collected it before. This indicates high polymorphism as we periodically visit each exploit server until it dies, indicating that the same server distributes different exploit instances for the same vulnerability over time. A detailed analysis of PDF exploit distribution reveals 11 Blackhole 2.0 (BH2) servers distributing PDF exploits. Consecutive visits to the same BH2 exploit server, separated by as little as 4 minutes, yield different PDF exploits for the same vulnerability. In addition, when we collect multiple instances of the same PDF exploit, all the instances appear in a single network trace (i.e., on the same drive-by download session). These indicate that automatic repacking of PDF exploits is likely integrated into the BH2 exploit kit and occurs for every drive-by download session. This means that byte signatures are unlikely to help in detecting PDF exploits.

In contrast, the ratio for EOT exploits is very low (0.3%), which indicates low polymorphism. Detailed analysis of the EOT files shows that of the 3 files one is served by the CoolExploit kit and the other 2 by different operations that use BH2. Each exploit instance uses a different URL to download the malware after exploitation and is never repacked, i.e., BH2 and CoolExploit servers always distribute the same EOT exploit instance over time. For example, the longest lived server in our dataset distributes the same EOT exploit for over 2 months. This likely indicates that there are no repacking tools available for EOT exploits. Once an EOT exploit is created with an embedded URL, attackers need to ensure that the URL stays up over long periods of time, e.g., by using bullet-proof hosting. This produces an easy avenue for detection. We expect that in the near future repacking tools will become available due to market demand, or that EOT exploits will be replaced by easier to repack exploit types.

The ratio for JAR exploit instances is 3.9% indicating some polymorphism, but much lower than PDF exploits. Figure 4.4 shows the CDF of the lifetime of JAR exploit instances, where the lifetime of an instance is measured as the difference between the last and first time it was collected. The median lifetime is 14.4 hours, the average is 39.3 hours, and the standard deviation is 121.9 hours (i.e., 5 days). 69.2% of the exploits have a lifetime less or equal to one day, 19.2% have a lifetime between one and two days, and the remaining more than2 days. The high standard deviation in lifetime and the overall low repacking rate indicate that Java exploits are not automatically repacked by the monitored exploit kits. They are likely repacked using tools that need to be manually invoked by the exploit server manager. These tools may be shipped (but not integrated) with

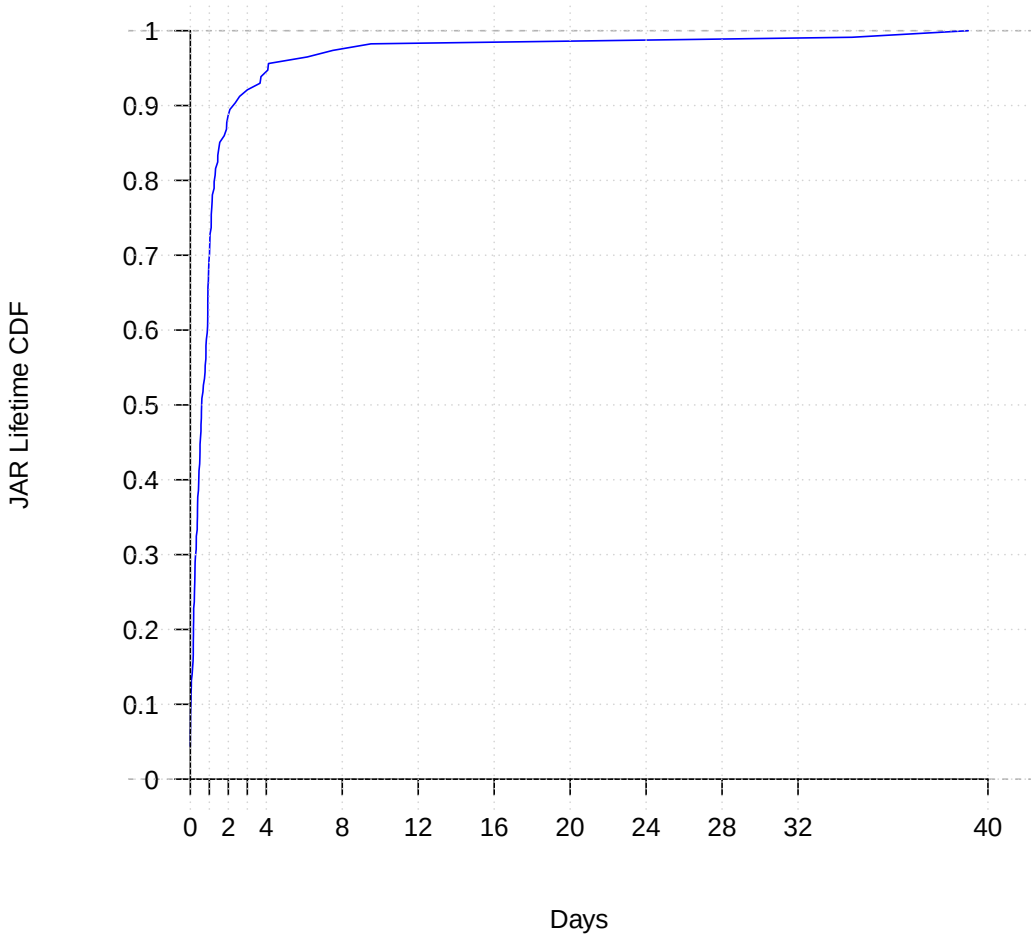


Figure 4.4: CDF of the lifetime of Java exploits.

the exploit kit or may be commercial-off-the-self (COTS) tools [166, 167].

The fact that EOT exploits are not repacked means that different exploit servers in the same operation may deliver the same EOT exploit instance. In addition, since the repacking of Java exploits requires some manual steps, exploit server managers are likely to copy the newly packed exploit instances across their exploit servers. We leverage these to incorporate exploit polymorphism as one of the features used in our exploit server clustering, presented in the next section.

Vulnerabilities exploited over time. Figure 4.5 shows the distribution over time of the exploits for each vulnerability. Of the 8 vulnerabilities, exploits for 5 of them were collected from the time we started producing network traces with

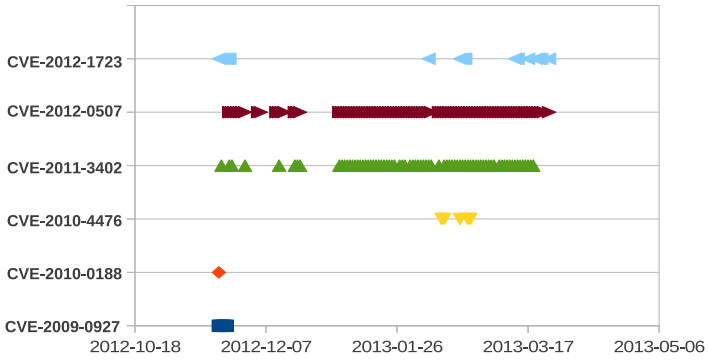


Figure 4.5: Exploit distribution of different vulnerabilities over time.

the honeyclients. Exploits for the 3 other vulnerabilities started to be collected 3 months later, which indicates that those vulnerabilities were added later to the exploit kits. Two of these (CVE-2013-0431 and CVE-2013-0422) are recent vulnerabilities. This illustrates the benefit of the exploit kit ecosystem where exploit kit writers focus on developing exploits for new vulnerabilities, and new exploits for old ones, and customers achieve improved infection rates by adding those exploits.

4.5 Exploit Server Clustering

To identify exploit servers managed by the same organization we propose an unsupervised clustering approach that groups exploit servers that have similar exploit kit configurations. The intuition is that exploit servers in the same operation are more likely to be similarly configured because attackers may reuse parts of the configuration across their servers. For example, attackers may clone an existing exploit server to create a new one in a different hosting provider by copying the image file with the exploit kit configured. Or, they may use a malicious domain to point to different exploit servers over time. In contrast, exploit servers in different operations are less likely to be similarly configured.

To cluster the exploit servers we define a distance metric between two exploit servers based on their configuration. The distance metric combines 7 features that capture how a exploit server is configured including the domains that point to the server, the server's IP address, the URLs it hosts, the exploits it serves, and the malware it distributes. These features are derived from our milk data.

This section details our 7 similarity features (Section 4.5.1) and the clustering algorithms we use (Section 4.5.2).

4.5.1 Features

We define 7 boolean server similarity features:

1. *Landing URL feature:* The landing URL of a exploit server contains elements that are specific to the configuration of the exploit kit. In particular, the file path in the landing URL (the directory where the kit's files are installed and the name of those files) and the parameter values (typically used to differentiate traffic sources) are configurable and changed from the default by the manager to make it difficult to produce URL signatures for the kit. This feature first extracts for each landing URL the concatenation of the file path (including the file name) and the list of parameter values. The similarity is one if the set intersection is non-empty, otherwise it is zero.

2. *Domain feature:* If the same DNS domain has resolved to the IP addresses of two exploit servers, that is a strong indication that both exploit servers belong to the same organization, i.e., the one that owns the domain. This feature first extracts the set of DNS domains that have resolved to the IP address of each server. The similarity between two servers is one if the set intersection is non-empty, otherwise the similarity is zero.

3. *File hash feature:* A malware owner can distribute its malware using its own infrastructure (HIY or EaaS) or a PPI service. However, it is unlikely that it will use both of them simultaneously because outsourcing distribution to a PPI service indicates a willingness to avoid investing in infrastructure. Thus, if the same malware executable (i.e., same SHA1 hash) is distributed by two servers, that is a strong indication of both exploit servers belonging to the same organization. This feature first extracts the set of file hashes milked from each exploit server. The similarity is one if the set intersection is non-empty, otherwise it is zero.

4. *Icon feature:* The icon in a malware executable is selected by the creator of the executable, i.e., malware owner or an affiliate PPI program (the program is typically in charge of repacking the affiliate-specific malware [73]). In both cases a shared icon in files distributed by different servers is a strong indication of both servers distributing malware from the same owner. This feature is related to the file hash feature but covers files that may have been repacked while keeping the same icon. This feature first extracts the set of icons in files milked from each exploit server. The similarity is one if the set intersection is larger than 1 otherwise it is zero.

5. *Family feature:* If two servers distribute the same malware family, and the malware family is neither a malware kit (e.g., zbot, spyeye) nor an affiliate program, then the two servers distribute malware of the same owner and thus share management. This feature is optional for the analyst to use because it requires a priori knowledge of which malware families are malware kits or affiliate programs, otherwise it may overcluster. This boolean feature

first extracts the set of non-kit, non-affiliate malware families distributed by each exploit server. The similarity is one if the set intersection is non-empty, otherwise it is zero.

6. *Consecutive IP feature:* If two exploit servers have consecutive IP addresses that is a strong indicator that both servers have been contracted by the same entity because the probability of two separate exploit server owners selecting the same ISPs and those ISPs selecting consecutive IP addresses for their servers is very small. The similarity between two servers is one if their IP addresses are consecutive, otherwise it is zero.

7. *Exploit hash feature:* Similar to malware, exploits may also be periodically repacked. Such repacking is either performed on-the-fly by the exploit kit or manually using a repacking tool. In both cases, if the same exploit (i.e., same SHA1 hash) is distributed by two servers, that is a strong indication of both exploit servers being managed by the same entity. This feature first extracts the set of exploit hashes collected from each exploit server. The similarity is one if the set intersection is non-empty, otherwise it is zero. Note that exploit types that are automatically repacked in each exploitation session, e.g., PDF exploits, will not capture similarity in this feature since each exploit server serves different PDF exploit instances. In our experiments, this feature captures similarity on the Java and EOT exploits only. However, with other exploit kits it is unknown a priori whether automatic repacking for a exploit type has been integrated in the kit. Thus, it is important to include all exploit types when computing this feature.

4.5.2 Clustering Algorithms

We experiment with two clustering algorithms: the partitioning around medoids (PAM) [168] and an aggressive clustering algorithm that groups any servers with some similarity.

Partitioning around medoids. The input to the PAM algorithm is a distance matrix. To compute this matrix we combine the server similarity features into a boolean server distance metric as $d(s_1, s_2) = 1 - (\bigvee_{i=1}^{5} f_i(s_1, s_2))$, where f_i is the server similarity feature i. Note that the features compute similarity (one is similar), while the distance computes dissimilarity (zero is similar). Once a distance matrix has been computed, we apply the PAM algorithm. Since PAM takes as input the number k of clusters to output, the clustering is run with different k values, selecting the one which maximizes the Dunn index [169], a measure of clustering quality.

Aggressive clustering. Our aggressive clustering first computes a boolean server similarity metric: two servers have similarity one if any of the server feature

similarities is one (logical OR). Then, it iterates on the list of servers and checks if the current server is similar to any server already in a cluster. If the current server is only similar to servers in the same cluster, we add the server to that cluster. If it is similar to servers in multiple clusters, we merge those clusters and add the current server to the merged cluster. If it is not similar to any server already in the clusters, we create a new cluster for it. The complexity of this algorithm is $O(n^2)$, but since the number of servers is on the hundreds, the clustering terminates in a few seconds.

4.6 Reporting

Reporting abuse is an important part of fighting cybercrime, largely overlooked by the research community. In this section we briefly describe the abuse reporting process and the challenges an abuse reporter faces. In Section 4.7.5 we detail our experiences reporting exploit servers and discuss the current situation.

Five entities may be involved in reporting an exploit server: the *abuser*, the *reporter*, the *hosed* that owns the premises where the exploit server is installed, the *abuser's ISP* that provides Internet access to the exploit server, and *national agencies* such as CERTs and law enforcement. Sometimes, the ISP is also the hoster because it provides both hosting and Internet access to the exploit server. The abuser can also be the hoster if it runs the exploit server from its own premises.

The most common practice for reporting exploit servers (and many other abuses[1]), is to first email an abuse report to the ISP's abuse handling team, who will forward it to their customer (i.e., the hoster) if they do not provide the hosting themselves. If this step fails (e.g., no abuse contact found, email bounces, no action taken), the reporter may contact the CERT for the country where the exploit server is hosted or local law enforcement. There are two main reasons to notify first the abuser's ISP. First, in most cases a reporter does not know the abuser's or hoster's identity. But, the abuser's ISP is the entity that has been delegated the IP address of the exploit server, which can be found in the WHOIS databases [170]. Second, ISPs that are provided evidence of an abuse of their terms of service (ToS) or acceptable use policy (AUP) by a host unlikely to have been compromised (e.g., an exploit server), can take down the abusing server without opening themselves to litigation. This removes the need for law enforcement involvement, speeding the process of stopping the abuse.

Next, we describe 3 challenges a reporter faces when sending abuse reports.

[1]This practice also applies to other types of abuse such as C&C servers, hosts launching SSH and DoS attacks, and malware-infected machines. However, spam is commonly reported from a receiving mail provider to the sender mail provider and web server compromises are commonly first reported to the webmaster.

Abuse report format and content. The Messaging Abuse Reporting Format (MARF) [171, 172, 173] defines the format and content for spam abuse reports. Unfortunately, it does not cover other types of abuse and proposals for extending it (e.g., X-ARF [174]) are still work-in-progress. In this work we use our own email template for reporting exploit servers. The key question is what information will convince an ISP of the abuse. The goal is to provide sufficient evidence to convince the ISP to start its own verification. The key evidence we include is a network trace of a honeyclient being exploited by the exploit server. We also include the IP address of the server, the first day we milked it, and pointers to public feeds listing the server.

Abuse contact address. Finding the correct abuse contact is not always easy (or possible). For spam, RFC 6650 states that abuse reports should only be sent to email addresses clearly intended to handle abuse reports such as those in WHOIS records or on a web site of the form abuse@domain [173]. Unfortunately, not all ISPs have an abuse@domain address. Such addresses are only required for ISPs that (care to) have an abuse team [175] and have not been mandatory in WHOIS databases until recently. Even now, they are often only mandatory for new or updated WHOIS entries and the objects and attributes holding this information are not consistent across databases. We are able to find abuse addresses for 86% of all exploit servers we milk. In practice, reporters use WHOIS to identify the organization that has been delegated the abuser's IP address. If an abuse email does not exist for the organization (or cannot be found in its website) abuse reports are sent to the organization's technical contact, which is mandatory in WHOIS. Unfortunately, after finding an email address to send the report, there is no guarantee on its accuracy.

Sender's identity. Abuse reports may end up being received by malicious organizations (e.g., bullet-proof ISPs or hosters). Thus, using an individual's real identity in an abuse report can be problematic. On the other hand, abuse teams may be suspicious of pseudonyms. Organizations that issue many abuse reports such as SpamHaus [176] can rely on their reputation, but they do not act as abuse aggregators. In this work, we use a pseudonym to hide our identities and still get access to the communication with ISPs and hosters.

4.7 Analysis

Table 4.3 summarizes our milking, which started on March 7, 2012 and has been operating for 12 months (the BlackHole/Phoenix dataset in [28] covered only until April 20). We have milked a total of 502 exploit servers, hosted in 57 countries and 242 ASes, and downloaded from them 46,514 malware executables, of which

Milking Period	2012-03-07 – 2013-03-25
Malware executables milked	46,514
Unique executables milked	11,363
Domains milked	603
Servers milked	502
ASes hosting servers	242
Countries hosting servers	57
Malware executions	21,765
Total Uptime days	383

Table 4.3: Summary of milking operation.

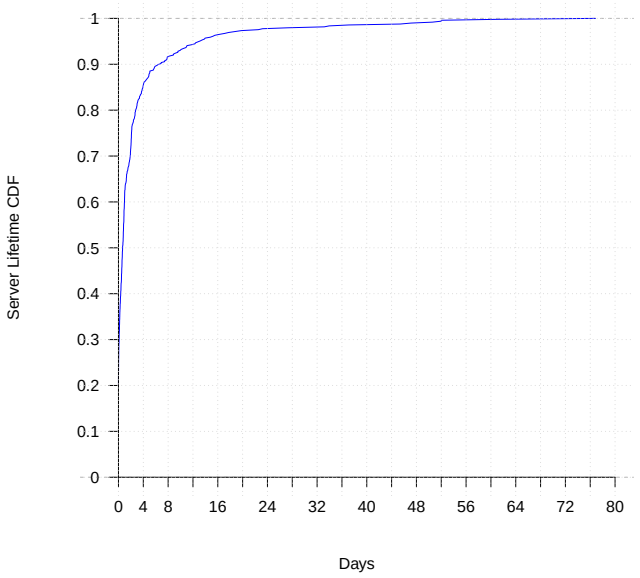


Figure 4.6: CDF of exploit server lifetime.

11,363 are unique (by SHA1 hash). A total of 603 DNS domains were observed pointing to the 502 servers.

4.7.1 Exploit Server Lifetime

To understand how well defenders are reacting to the drive-by download threat, we measure the exploit server lifetime, i.e., the period of time during which it distributes malware. For this measurement we use only exploit servers found after we updated our infrastructure to identify servers by landing IP (Section 4.4.2) and remove servers for which we have sent abuse reports (Section 4.7.5). Figure 4.6 presents the CDF for the exploit server lifetime. The majority of exploit servers

ASN	Name	CC	Days up	ES	AS Rank Size	AS Rank FIRE
16276	ovh	FR	194.84	21	5,623	10
701	uunet	US	139.60	1	8	-
44038	swisscom	CH	76.8	1	8,496	-
47869	netrouting	NL	70.0	18	4,395	-
43637	sol	AZ	61.1	1	6,828	-
48716	ps	KZ	52.0	1	8,530	-
56964	rmagazin	RO	49.5	2	8,337	-
12695	di-net	RU	47.6	9	175	-
36992	etisalat	EG	47.1	1	1,136	-
197145	infiumhost	RU	44.8	8	1,384	-
36444	nexcess.net.l.l.c	US	37.4	4	5,798	-
56413	proservis	LT	37.1	1	8,553	-
16265	leaseweb	NL	36.8	8	3,089	7
58182	kadroviy	RU	30.5	3	-	-
5577	root	LU	28.7	7	1,171	-
40676	psychz	US	28.1	5	6,939	-
21788	burst	US	27.8	14	3,942	-
28762	awax	RU	27.0	15	4,644	-
44784	sitek	UA	23.2	1	-	-
15971	ecosoft	RO	19.1	5	-	-

Table 4.4: Top ASes by cumulative exploitation time.

are short-lived: 13% live only for an hour, 60% are dead before one day, and the median lifetime is 16 hours. However, it is worrying to observe a significant number of long-lived servers: 10% live more than a week, 5% more than two weeks, and some servers live up to 2.5 months.

The median exploit server lifetime we measure is more than six times larger than the 2.5 hours median lifetime of a exploit domain (a domain resolving to the landing IP of an exploit server) measured by Grier et al. using passive DNS data [28]. This shows the importance of identifying exploit servers by their IP address, accounting for multiple domains pointing to the same server over time.

4.7.2 Hosting

In this section we analyze the hosting infrastructure. We find that miscreants are abusing cloud hosting services. We also find, similar to prior work [63, 64], autonomous systems hosting an inordinate number of exploit servers, compared to the size of their IP space.

Cloud hosting services. Using WHOIS we can first determine which organization has been delegated the IP address of an exploit server and then use web

Family	**Kit**	**ES**	**Files**	**Milk**	**Repack Rate**
zbot	Kit	170	2,168	11,619	18.6
cridex		64	74	2,555	2.8
zeroaccess	Aff	21	1,306	3,755	18.0
winwebsec	Aff	18	5,820	16,335	59.5
spyeye	Kit	11	7	342	2.0
CLUSTER:A		9	14	266	2.2
securityshield		5	150	307	11.8
CLUSTER:B		4	45	51	30.4
CLUSTER:C		4	1	4	1.0
smarthdd		4	68	453	3.1
CLUSTER:D		3	3	32	3.0
CLUSTER:E		3	1	4	1.0
CLUSTER:F		3	9	531	0.7
webprotect		3	3	26	3.9
cleaman		2	32	103	7.7
CLUSTER:G		2	5	148	1.5
CLUSTER:H		2	24	43	21.7
CLUSTER:I		2	9	17	9.4

Table 4.5: Top malware families by number of exploit servers observed distributing the family.

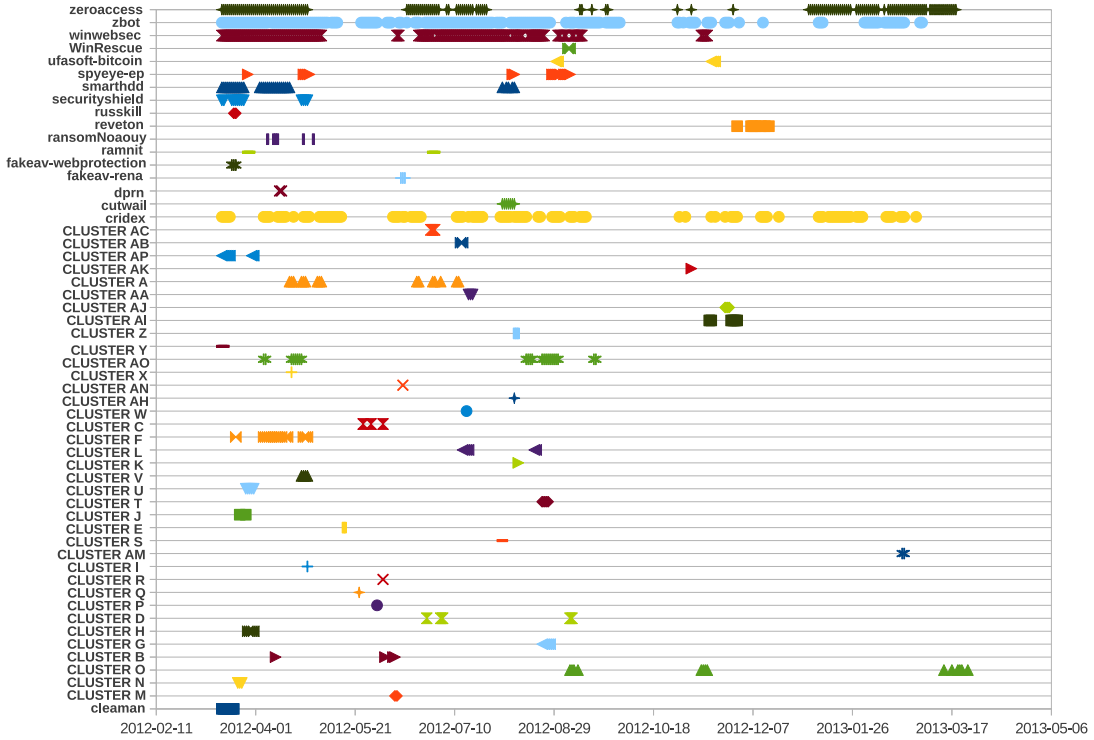


Figure 4.7: Malware family distribution.

searches to determine if it offers cloud hosting services. Our results show that at least 60% of the exploit servers belong to cloud hosting services, predominantly to Virtual Private Server (VPS) providers that rent VMs where the renter gets root access. This number could be larger because ISPs do not always reveal in WHOIS whether an IP address has been delegated to a customer, who may be a hosting provider. This indicates that drive-by operations have already embraced the benefits of outsourcing infrastructure to the cloud.

AS distribution. Table 4.4 shows the top ASes by the cumulative uptime (in days) of all exploit servers we milked in the AS. It also shows the number of exploit servers in the AS, the CAIDA ranking of the AS by the number of IPv4 addresses in its customer cone (the lower the ranking the larger the AS) [177], and the FIRE ranking for malicious ASes [63]. The two ASes with the largest number of exploit servers are in Europe and the average life of an exploit server in those ASes is 10 days and 4 days respectively, well above the median lifetime of 16 hours. Some small ASes host an inordinate number of exploit servers compared to their ranking such as `awas` and `infiniumhost`, both located in Russia. There are also 3 ASes in Eastern Europe that do not advertise any IP addresses or no longer exist, which could indicate that they were set up for such operations. We milked servers in 3 ASes that appear in the 2009 FIRE ranking. Two of them (`ovh`

and `leaseweb`) appear also among our top ASes, which indicates that practices at those ASes have not improved in 3 years.

4.7.3 Malware Families

Our classification has identified a total of 55 families. Table 4.5 shows the top 18 families sorted by the number of exploit servers observed distributing the family. For each family, Table 4.5 shows whether the family is a known malware kit or affiliate program, the number of servers distributing the family, the number of unique files milked, the total number of binaries milked from that family, and its repacking rate. Overall, the most widely distributed families are information stealers (zbot, cridex, spyeye), PPI downloaders (zeroaccess), and rogue software (winwebsec, securityshield, webprotect, smarthdd). The family most milked was winwebsec, a fake antivirus affiliate program, while the one distributed through most servers was zbot, a malware kit for stealing credentials.

Figure 4.7 shows the distribution of malware families over time. While most families are distributed through short operations, there are a few families such as zeroaccess, cridex, and zbot, which have been distributed throughout most of our study.

Families with shared ownership. Since different malware families target different monetization mechanisms, malware owners may operate different families to maximize income from compromised hosts. There are 50 servers distributing multiple malware families. Nine servers distribute different malware families through the same landing URL, during the same period of time, and to the same countries, e.g., a visit from the US with no referer would drop family 1, another visit from the US a few minutes later family 2, and then again family 1. This indicates those families share ownership, as there is no way to separate the installs from the different families. Some families that manifest shared ownership are: CLUSTER:D and cleaman, and securityshield and smarthdd. There is also shared ownership involving families known to be malware kits or affiliate programs such as winwebsec affiliates installing zbot and CLUSTER:L, and zbot botmasters installing ramnit.

Repacking rate. Malware owners repack their programs periodically to avoid detection by signature-based AV. On average, a malware family (excluding kits and affiliate programs) is repacked 5.4 times a day in our dataset. This is a sharp rise compared to the 0.1 times a day prior work reported during August 2010 [73]. This trend will further harm the detection rate of signature-based AVs. The rightmost column in Table 4.5 shows the repacking rate for our top families. The rate for families known to be kits or affiliate programs is artificially high, covering multiple botnets or affiliates. There are other families with high repacking rates

	Features 1–4			Features 1–5		
Alg.	**Clusters**	**Largest**	**Singletons**	**Clusters**	**Largest**	**Singletons**
Agg.	174	64	121	109	142	71
PAM	256	31	188	204	31	141
	Features 1–4,6,7			Features 1–7		
Alg.	**Clusters**	**Largest**	**Singletons**	**Clusters**	**Largest**	**Singletons**
Agg.	156	86	111	101	195	73
PAM	294	31	229	261	34	199

Table 4.6: Operation clustering results.

such as securityshield, CLUSTER:B, and CLUSTER:H. This could indicate that those families are malware kits or affiliate programs.

4.7.4 Operations Analysis

In this section we evaluate our clustering approach to identify operations that use multiple exploit servers. Unfortunately, we do not have ground truth available to evaluate our clustering results in a quantitative fashion. In fact, if such ground truth was available, then there would be no need for the clustering. Instead, we argue qualitatively that our clustering identifies meaningful and interesting drive-by operations.

Table 4.6 summarizes the clustering results. It compares the clustering results with both algorithms and using 4 different feature sets. The two leftmost feature sets (Features 1–4 and 1–5) correspond to the ones used in our prior work [44] while the 2 rightmost feature sets correspond to adding the new features on consecutive IP addresses and exploit polymorphism. For the new feature sets, we include the clustering results with and without the family feature for comparison. However, for the operation analysis below we focus on the results without the family feature (Features 1–4,6,7), since we suspect some families like securityshield to be affiliate programs. Since those are distributed alongside other malware, the family feature can overcluster. For each clustering algorithm and feature set the table shows the number of clusters, the size of the largest cluster, and the number of singleton clusters with only one server. As expected, the aggressive algorithm groups the most, minimizing the number of clusters.

We first present a number of operations our clustering reveals (for the aggressive clustering with 6 features unless otherwise noted), evaluating their correctness with information not used by our features such as which kit was installed in the exploit server and for affiliate programs, which affiliate a malware executable belongs to (we extract the affiliate identifier from the network traffic). Finally, we summarize the types of operations the clustering reveals and their distribution properties including the number of servers used, their hosting, and the operation lifetime.

Phoenix operation. Using both PAM and aggressive all 21 Phoenix servers are grouped in the same cluster, which exclusively distributes zbot. Here, the clustering reveals that the Phoenix servers belong to the same operation *without* using any features about the exploit kit. Both algorithms do not include servers from other kits in the cluster, so they are not overclustering.

Reveton operation. We observe two clusters exclusively distributing the Reveton ransomware, which locks the computer with fake police advertisements. One cluster has 14 CoolExploit servers, the other 3 CoolExploit and one BlackHole 2.0. This agrees with external reports on the Reveton gang switching from BlackHole to the newer CoolExploit kit [178]. Here, the clustering captures an operation using different exploit kits, but possibly underclusters as both clusters likely belong to the same operation.

Winwebsec operation. We observe the winwebsec fake AV affiliate program distributed through 22 different servers in 8 clusters. There exists 3 singleton clusters, each exclusively distributing the winwebsec executable of a different affiliate. Another cluster of 8 servers distributes affiliate 60830 as well as another unknown malware family and zbot. The other 4 clusters distribute the executables of multiple affiliates. Here, there exist two possibilities: the same group could have signed up to the winwebsec program multiple times as different affiliates, or the affiliate program is managing the exploit server so that affiliates can convert their traffic into installs. To differentiate between both cases, we check their landing URLs. One of these clusters uses the same landing URL to distribute the executables of affiliates 66801, 66802, and 66803. In this case, there is no way to separate the installs due to each affiliate, which indicates those affiliates belong to the same entity. The other three clusters use different landing URLs for each affiliate, which indicates those servers are run by the affiliate program, which provides a distinct landing URL to each affiliate.

We confirm that the winwebsec program manages their own exploit servers through external means. We leverage a vulnerability on old versions of BlackHole, where the malware URLs used a file identifier that was incremented sequentially, and thus could be predicted. On March 12, we tried downloading file identifiers sequentially from one of the servers distributing multiple winwebsec affiliates. We found 114 distinct executables, of which 108 were winwebsec executables for different affiliates, one did not execute, and the other 5 corresponded to other malware families, including smarthdd and the Hands-up ransomware [179]. This indicates that on March 12, the winwebsec program had 108 affiliates and that the winwebsec managers, in addition to their own program, were also distributing other rogue software.

Zeroaccess operations. Zeroaccess is also an affiliate program [31]. With the aggressive algorithm there are 8 clusters distributing zeroaccess: 5 distribute a

single affiliate identifier, the other 3 multiple. For two of these 3 the distribution is simultaneous and on a different landing URL for each affiliate, which indicates that the zeroaccess affiliate program also manages their own exploit server. The other distributes two affiliate identifiers on the same URL, indicating those affiliates belong to the same entity.

Zbot operations. There are 51 clusters distributing zbot in the aggressive clustering. Of these, 38 clusters distribute exclusively zbot, the largest using 21 servers over 6 days. For each of these 38 clusters we compute the set of C&C domains contacted by the malware milked from servers in the cluster. Only 3 of the 38 clusters have C&C overlap, which indicates that our non-family features capture enough shared configuration to differentiate operations distributing the same malware kit.

Broken malware operation. We identify a cluster with 13 servers that operates on a single day and distributes a single file. Surprisingly, the file does not execute. Apparently, the malware owners realized the malware was corrupt and stopped the operation.

Operations summary. The clustering reveals two types of operations. Two thirds of the clusters are singletons. They correspond to small operations with one server that lives on average 14 hours. Most singletons distribute a single family, which is often zbot or one of the generic families for which we have not found a published name. The remaining are operations that leverage multiple servers for their distribution. Multi-server operations use on average 6.2 servers and diversify their hosting. On average, each multi-server operation hosts 1.2 servers per country, and 2 servers per AS. Multi-server operations last longer with a median life of 5.5 days and only 1.2 servers operate on the same day. This indicates that they are replacing servers over time to sustain distribution, rather than using them for sudden bursts of installs (although we observe bursts like the broken malware operation mentioned earlier).

Feature set comparison. We compare the feature sets in our prior work [44] with the new feature sets, which add the consecutive IP addresses and exploit hash features. With aggressive clustering, the new features reduce the number of clusters and increase the average cluster size, while with PAM the two new features further separate the clusters, slightly increasing their number. With aggressive clustering, the exploit hash feature is most helpful, merging 13 small clusters into the largest cluster (86 servers). To verify that indeed the exploit hash feature is capturing exploit servers that share management, we analyze the time distribution of the exploit instances that are served by multiple servers, finding that their distribution happens very close in time. This supports our hypothesis

that once the exploits are repacked by the managers, they are distributed to the different exploit servers in the operation. The consecutive IP addresses feature merges 5 old clusters into 2 new clusters. These new clusters correspond to operations distributing zbot and cridex. The new clusters show that the miscreants are running 4-6 servers in the same cloud hosting provider, with all exploit servers using consecutive IPs. This indicates that if a cloud hosting server is abused the miscreants may install multiple servers in that hosting provider.

4.7.5 Reporting Analysis

We started sending abuse reports on September 3rd, 2012 for exploit servers that we had been milking for 24 hours. Most abuse reports did not produce any reply. Of the 19 reports we sent, we only received a reply in seven; 61% of the reports were not acknowledged. For two of the ISPs we were unable to locate an abuse@domain address in WHOIS. One of these had no technical support contact either, so we resorted to web searches to find an email address. The absence of an abuse@domain address indicates a lack of interest in abuse reports. As expected, those reports did not produce a reply.

All initial replies contained a ticket number, to be included in further communications about the incident. Three of them also provided a URL for a ticket tracking system. Two of the replies came from ISPs to whom we had sent more than one report (on different dates). Surprisingly, only one of the two reports produced a reply. This lack of consistency indicates manual processing and that the response to an incident may depend on the abuse team member that first reviews the report.

After reporting a server, we keep milking it to understand how long it takes to act on a report. Note that, these reaction times are lower bounds because the servers could have been reported earlier by other parties. On average an exploit server lives 4.3 days after a report. Exploit servers whose report did not generate a response lived on average for 5.1 days after our report. Servers whose report produced a reply lived for 3.0 days. Thus, the probability of action being taken on the report when no reply is received is significantly smaller. Next, we detail the reactions to the 7 reports with replies.

Report 1. The most positive report. The exploit server was a VPS hosted by the ISP, which immediately disconnected it and notified us of the action (which we confirmed).

Report 2. This large US ISP replied with an automated email stating that they take abuse reports seriously but cannot investigate or respond to each of them. No further reply was received and the server lived for 4 days.

Report 3. A ticket was open with medium priority promising further notification. No further response was received and the server lived for another day.

Report 4. The report was forwarded to a customer. After a day the server was still alive so we sent a second report stating that the customer had not taken action and the ISP proceeded to disconnect the server.

Report 5. The report was forwarded to a customer and our ticket closed without waiting for the customer's action. The server was still alive for 1.7 days.

Report 6. The reply stated they would try to get back within 24 hours and definitely before 72 hours. The server lived two more hours and we never heard back.

Report 7. The initial reply stated that it was a customer's server and that according to the Dutch Notice and Take-down Code of Conduct [180], we had to notify the customer directly. Only if the customer did not reply after 5 days, or their reply was unsatisfactory, we could escalate it to them. We reported it to the client and after 5 days the server was still alive. We re-reported the exploit server to the ISP who told us to contact the customer again, which we did copying the ISP. This time the customer replied but was not willing to act on the response unless we reveal our real identity, which we declined. It seems that the ISP called them requesting the disconnection. The ISP later notified us about the disconnection. As far as we can tell, the five days waiting time is not part of the Dutch Notice and Take-down Code of Conduct.

These reports show that if the exploit server is hosted by a hosting provider who is a customer of the ISP, the ISP simply forwards them the abuse report and does no follow-up. It is up to the reporter to monitor the customer's action and re-report to the ISP in case of inaction. They also show how painful abuse reporting can be and the need for an homogeneous code of conduct for takedowns.

4.8 Malicia Dataset

To foster future research, e.g., in malware classification, we have compiled the data collected in this work into the Malicia dataset, which we make available to other researchers. The Malicia dataset can be requested by researchers at academia, research labs, and industry labs following the instructions at the dataset's webpage [153]. It is released under an agreement to not redistribute the dataset and only to researchers under contract from a research institution. Students need to ask their supervisors to request to the dataset on their behalf. We use basic identity checks (e.g., that the email address from which the request is sent belongs to the institution requesting it) before releasing the dataset. At the

time of writing the dataset has been released to 17 institutions: 15 universities and 2 industrial research labs.

This section briefly describes the publicly available 1.0 release of the Malicia dataset. Then, it describes the updates that we plan to add into release 1.1, which we will make available upon publication of this manuscript.

4.8.1 Release 1.0

The current release of the Malicia dataset contains data for all the milking period (March 7th, 2012 – March 25th, 2013). The dataset comprises 5 files. The core of the dataset is a MySQL database with all the milking metadata including when the malware was collected, from where the malware was collected, the malware classification, and exploit server information. In addition, there is a figure that captures the database schema, a tarball with the malware binaries, another tarball with the icons extracted from those malware binaries, and a signature file for the Snort IDS produced by our FIRMA tool.

Database. The database comprises 8 tables. The most important table in the DB is the MILK table, which contains a row for every time a malware was milked from a exploit server. Every row contains the timestamp of when the malware was milked, the landing URL, as well as identifiers linking to the other DB tables. Other two important tables are the FILES and LANDING_IP tables. The FILES table contains a row for each unique malware binary (identified by SHA1 hash) and its classification information. The LANDING_IP table contains a row for each exploit server (identified by the landing IP), the exploit kit installed in the server, the server's autonomous system number, and its country code.

Malware. The malware tarball contains 11,363 samples (.exe and .dll files). For each malware binary, its network traffic, icon, and screenshot labels, as well as the final family label, can be found in the FILES table in the DB.

Icons. The icons tarball contains 5,777 icons extracted from the executables. We provide the icons for convenience, since they can be extracted from the provided malware.

4.8.2 Release 1.1

In the next release of the Malicia dataset we plan to include the exploits extracted from the network captures of our honeyclients (Section 4.4.4) and additional malware classification information.

Exploits. This release will include an exploits tarball with the 172 exploits collected, named using their SHA1 hash. The database will contain an additional table where each row represents one exploit file and contains its hash, size, file type, and CVE label.

Updated malware classification. The 1.0 release includes one month of unclassified malware samples, which were collected after the DIMVA 2013 paper was written, but before the dataset was released. The 1.1 release will update the malware classification so that it covers all samples and corresponds to the classification data used in this manuscript.

4.9 Discussion

In this section we discuss implications of our work, avenues for improvement, and suggest other applications.

Takedown challenges. Setting up a exploit server in the cloud is a simple and cheap process, while taking down a exploit server can be a complicated one and may cost little to the exploit server owner, which can simply move to another provider. We need to raise the cost for the exploit server managers of one of their exploit servers being taken down and make it less anonymous to rent them. Hosting providers should incorporate processes to verify the identity of the renter, assign reputation to payment accounts, and closely monitor short leases, as these are more likely to be abused. More emphasis is also needed on the attribution of the organizations running drive-by operations and those providing the specialized underground services supporting them. We believe that this work is a step in that direction.

Criminal investigations. Our clustering is designed to be used by law enforcement during the pre-warrant (plain view) phase of a criminal investigation [150]. During this phase, criminal activity is monitored and targets of importance are selected among suspects. Our clustering can identify large operations among all reported exploit servers, satisfying this requirement. The goal of the plain view phase is gathering enough evidence to obtain a magistrate-issued warrant for the ISPs and hosting providers for the servers in the operation.

Additional features. Other features can be incorporated to our clustering to further identify servers with shared configuration. For example, we could incorporate the web server version running in the exploit server and the registrant for DNS domains pointing to the servers.

Improving coverage. In this work we show that even with a small number of drive-by download feeds we can identify exploit servers in the same operation. Adding more feeds would improve our coverage, identifying more operations. Acquiring feeds is challenging because many security vendors are careful about sharing them since they consider them a competitive advantage.

Other applications. The problem of distinguishing from a pool of servers running the same software which servers are managed by the same organization is not exclusive to exploit servers. Malware kits pose similar challenges because they provide configurable bot and C&C server software, shared among all organizations buying the kit. Our technique could be applied to this scenario to identify C&C servers in the same operation.

4.10 Conclusion

We have proposed a technique to identify drive-by download operations by clustering exploit servers under the same management based on their configuration, the exploits they serve, and the malware they distribute. Our analysis reveals that to sustain long-lived operations miscreants are turning to the cloud. We find that 60% of the exploit servers are hosted by specialized cloud hosting services. We have performed what we believe is the first quantitative analysis of exploit polymorphism. We observe that different types of exploits are repacked differently; repacking can be integrated in the exploit kit to be performed automatically, invoked manually using external tools, and some exploit types (e.g., fonts) may not be repacked at all. We have also analyzed the abuse reporting procedure with discouraging results: most abuse reports go unanswered and even when reported, it still takes several days to take down an exploit server.

5

CyberProbe: Towards Internet-Scale Active Detection of Malicious Servers

5.1 Preamble

This chapter reproduces the work "CyberProbe: Towards Internet-Scale Active Detection of Malicios Servers" published at NDSS 2014. This paper shows a new and general active probing technique to detect malicious servers. This work was realized together with people from the IMDEA Software Institue and Texas A&M university. Antonio Nappa has been the leading author of this work.

5.2 Introduction

Cybercrime is one of the largest threats to the Internet. At its core is the use of malware by miscreants to monetize infected computers through illicit activities such as spam, clickfraud, ransomware, and information theft. To distribute the malware, control it, and monetize it, miscreants leverage remotely-accessible servers distributed throughout the Internet. Such malicious servers include, among many others, exploit servers to distribute the malware through drive-by downloads, C&C servers to control the malware, web servers to monitor the operation, and redirectors for leading fake clicks to advertisements. Even P2P botnets require "server-like" remotely accessible peers for newly infected hosts to join the botnet.

Identifying the server infrastructure used by an operation is fundamental in the fight against cybercrime. It enables take-downs that can disrupt the operation [181], sinking C&C servers to identify the infected hosts controlled by the operation [182], and is a critical step to identify the miscreants running the operation, by following their money-trail [150].

Most current defenses identify malicious servers by passively monitoring for attacks launched against protected computers, either at the host (e.g., AV in-

stallations, HIDS) or at the network (e.g., NIDS, spamtraps, honeypots), or by running malware in a contained environment monitoring their network communication [65, 66]. These passive approaches achieve limited coverage, as they only observe servers involved in the attacks suffered by the protected hosts, or contacted by the malware samples run. To increase coverage, security companies aggregate information from multiple distributed sensors or execute more malware samples, but this requires a large investment or a large user base, and still does not achieve Internet-scale coverage. These approaches are also slow; malicious servers are detected asynchronously, when attacks happen to target the protected hosts. This is problematic because miscreants often use dynamic infrastructures, frequently moving their servers to make detection difficult, as well as in reaction to individual server takedowns [44].

By the time a new server is detected, a previously known one may already be dead.

A prevalent active approach for identifying malicious servers is using honeyclient farms, which visit URLs, typically found through crawling, looking for exploit servers performing drive-by downloads [54, 57]. Such farms are at the core of widely deployed browser defenses such as Google's SafeBrowsing and Microsoft's Forefront. However, honeyclients focus on exploit servers and do not cover other malicious server types. In addition, achieving coverage is expensive, requiring large investments in server farms to run the crawlers and honeyclients. Thus, they are often deployed only by large corporations.

In this paper, we propose a novel *active probing* approach for detecting malicious servers and compromised hosts that listen for (and react to) incoming network requests. Our approach sends probes to remote hosts and examines their responses, determining whether the remote hosts are malicious or not. The probes are sent from a small set of *scanner* hosts to a large set of *target* hosts. The targets may belong to the same network (e.g., a hosting facility), different networks across the Internet (e.g., all hosting facilities of the same provider), or correspond to all remotely accessible Internet hosts. Our approach is general and can identify different malicious server types including C&C servers, exploit servers, web front-ends, and redirect servers; as well as malware that listens for incoming traffic such as P2P bots.

Compared with existing defenses, our active probing approach is fast, cheap, easy to deploy, and achieves Internet scale. It does not require a sensor to be hosted in every network. Using 3 scanners, it can probe the Internet in 24 hours searching for a specific family of malicious servers, e.g., C&C servers of the same malware family or exploit servers of a specific operation. The scanners can be geographically distributed and rate-limited to respect bandwidth constraints on the networks hosting them. To reduce the probing time we can simply add more scanners. Given its speed, it can be used to understand the size of the server infrastructure used by an operation at a small window of time.

Furthermore, it enables tracking (dynamic) malicious infrastructures over

time, by periodically scanning for the servers of the same operation.

We have implemented our approach in a tool called CyberProbe, which comprises two components: *adversarial fingerprint generation* and *scanning*. CyberProbe implements a novel adversarial fingerprint generation technique, which assumes that the servers to be fingerprinted belong to an adversary who does not want them to be fingerprinted. Adversarial fingerprint generation takes as input network traces capturing dialogs with servers of a malicious *family of interest*, and builds a fingerprint, which captures what probes to send and how to determine from a target's response if it is malicious. The fingerprint generation process is designed to minimize the traffic sent to malicious servers and to produce inconspicuous probes to minimize the chance of detection by the adversary. The scanning component takes as input a fingerprint and a set of target ranges and probes those targets to check if they belong to the family of interest.

We have used CyberProbe to build 23 fingerprints for 13 malicious families (10 malware families and 3 drive-by download operations). Using CyberProbe and those fingerprints, we perform 24 scans (12 of them Internet-wide). The scans identify 7,881 P2P bots and 151 distinct malicious servers including C&C servers, exploit servers, payment servers, and click redirectors. Of those servers, 75% are unknown to 4 public databases of malicious infrastructure: VirusTotal [25], URLQuery [157], Malware Domain List [156], and VxVault [183]. This demonstrates that for some families CyberProbe can achieve up to 4 times better coverage than existing techniques. CyberProbe is also fast; in some cases it can even identify malicious servers *before* they start being used by the miscreants, when they are simply on stand-by.

Our results uncover an important *provider locality* property. A malicious operation hosts an average of 3.2 servers on the same provider to amortize the cost of setting up a relationship with the provider. As malicious servers are often hosted in cloud hosting providers [44], these providers need to be aware of provider locality. When they receive an abuse report for a malicious server, chances are more servers of the same family are being hosted on their networks.

This work makes the following contributions:

- We propose a novel active probing approach for Internet-scale detection of malicious servers. Our approach sends probes to remote target hosts and classifies those targets as belonging to a malicious family or not. Compared to current solutions our active probing approach is fast, scalable, easy to deploy, and achieves large coverage.

- We implement our approach into CyberProbe, a tool that implements a novel adversarial fingerprint generation technique, and three network scanners. CyberProbe builds fingerprints from a set of network traces for a malicious family, under the assumption that the adversary does not want its servers to be fingerprinted, and probes target networks or the Internet using those fingerprints.

- We use CyberProbe to conduct 24 localized and Internet-wide scans for malicious servers. CyberProbe identifies 151 malicious servers, 75% of them unknown to existing databases of malicious activity. It also uncovers an important provider locality property of the malicious servers hosting infrastructure.

5.3 Overview and Problem Definition

CyberProbe uses an active probing (or network fingerprinting) approach that sends probes to a set of remote hosts and examines their responses, determining whether each remote host belongs to a malicious family or not. Network fingerprinting has been a popular security tool for nearly two decades [76]. A fingerprint identifies the type, version, or configuration of some networking software installed on a remote host. It captures the differences in the responses to the same probes sent by hosts that have the target software installed and those that have not. A fingerprint can identify software at different layers of the networking stack. Tools like Nmap [121] use it to identify the OS version of remote hosts, and other tools like fpdns [79] or Nessus [78] use it for identifying application-layer software such as DNS or Web servers.

Our fingerprints target application-layer software and its configuration. Each fingerprint targets a specific malicious family. For C&C servers and P2P bots, a fingerprint identifies the C&C software used by a malware family. For exploit servers, a fingerprint can identify the exploit kit software or a specific configuration of the exploit kit. For example, a fingerprint could be used to identify all BlackHole exploit servers on the Internet, and a different fingerprint could be used to identify only BlackHole exploit servers belonging to a specific operation. For the latter, we leverage the intuition that exploit servers belonging to the same operation are managed by the same individuals, and therefore have similarities in their (exploit kit) configuration [44]. Since an exploit kit is typically a set of web pages and PHP scripts installed on an off-the-shelf web server (e.g., Apache or Nginx), the fingerprint needs to capture characteristics of the exploit kit independent of the underlying web server.

A malicious family may have multiple fingerprints. For example, a malware family may use different C&C protocols, or different messages in the same C&C protocol. A different fingerprint can be generated for each of those protocols or message types, but all of them identify the same family. Similarly, an exploit kit stores a number of files on a web server (e.g., PHP, PDF, JAR), and a fingerprint could capture a probe (and its corresponding response) for each of those files.

Our active probing approach takes as input network traces capturing traffic involving a few *seed servers* that belong to the family of interest, often only one. The fingerprints CyberProbe generates enable finding not only the seed servers, but also other previously unknown servers from the same family. Thus, active

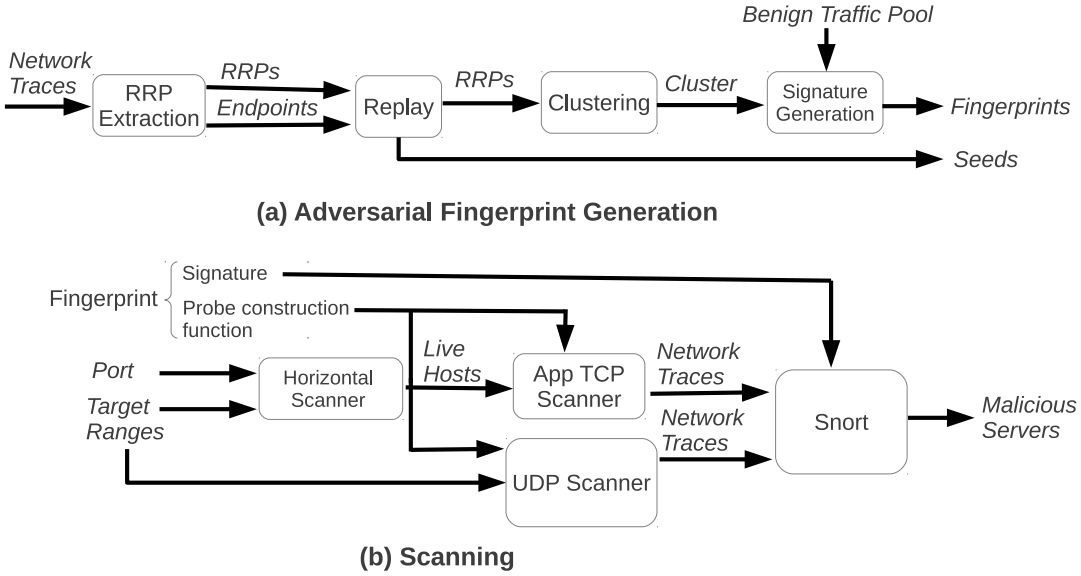


Figure 5.1: Architecture overview.

probing provides a way of amplifying the number of servers known to be part of the infrastructure of a malicious operation.

5.3.1 Problem Definition

The problem of *active probing* is to classify each host h in a set of remote target hosts H as belonging to a target family x or not. Active probing comprises two phases: *fingerprint generation* and *scanning*. The goal of fingerprint generation is to produce one or more fingerprints for a *family of interest* x, where each fingerprint $FG^x = \langle P, f_P \rangle$ comprises a *probe construction function* P and a *classification function* f_P. The probe construction function returns, for a given target host $h \in H$, the sequence of probes to be sent to the target host. The classification function is a boolean function such that when we send the probes $P(h)$ to host h and collect the responses R_P from h, $f_P(R_P)$ outputs true if h belongs to the family of interest and false otherwise. The goal of scanning is given a fingerprint, a port number, and a set of target hosts, to send the probes, collect the responses, and determine whether each target host belongs to the family of interest (i.e., matches the fingerprint).

5.3.2 Adversarial Fingerprint Generation Overview

In this work we introduce the concept of *adversarial fingerprint generation*, i.e., how to generate fingerprints for servers owned by an adversary who may not want them to be fingerprinted.

The challenge in traditional fingerprint generation is to find probes that trigger *distinctive* responses from servers in the family of interest, i.e., responses that can be differentiated from those by servers not in the family. A general framework for fingerprint generation is proposed in FiG [88]. It generates candidate probes, sends them to a set of training hosts comprising hosts in the family of interest and outside of it, and applies learning algorithms on the responses to capture what makes the responses from hosts in the family of interest distinctive.

Our adversarial fingerprint generation approach follows that framework, but has two important differences. First, we consider an adversarial scenario where the set of training hosts from the family of interest are malicious servers. We do not control them and they may be tightly monitored by their owners. In this scenario, it is critical to minimize the amount of traffic sent to those malicious seed servers and to produce probes that look inconspicuous, i.e. that resemble valid messages. As FiG generates random candidate probes, a huge number of such candidates needs to be sent before finding a distinctive response, as most random probes do not have proper protocol structure and will be ignored or incite a generic error response. Instead, CyberProbe replays previously observed requests to the seed servers. These requests come from valid interactions with the malicious servers and thus are well-formed and inconspicuous. We obtain such requests by executing malware in a contained environment (Section 5.3.4), by monitoring a honeyclient as it is exploited in a drive-by download, or from external analysis [184].

Second, our approach differs in the process used to build the classification function. FiG's classification functions have two main problems: they operate on the raw response, ignoring any protocol structure, and they need a specific matching engine.

Instead, a key intuition in this work is that the classification function can be implemented by using a network signature on the responses from the targets. Network signatures typically capture requests sent by malware infected hosts, but can similarly capture responses from remote endpoints. This relationship between fingerprint generation and signature generation enables prior and future advances on either field to be applied to the other. CyberProbe generates protocol-aware network signatures compatible with Snort [34] and Suricata [162], two efficient signature-matching open source IDSes. Figure 5.2 shows example fingerprints for a clickfraud operation and a drive-by download operation.

Figure 5.1a shows the adversarial fingerprint generation architecture. It takes as input a set of network traces capturing interactions with servers from the family of interest. First, it extracts the unique request-response pairs (RRPs) in the traces. Then, it replays the requests to the servers in the traces, keeping only replayed RRPs with distinctive responses. Next, it clusters similar requests. Finally, it generates signatures for the responses in a cluster. It outputs one or more fingerprints for the family of interest, each comprising a probe construction

clickpayz1

probe: GET /td?aid=e9xmkgg5h6&said=26427
signature: alert tcp any any -> any any (msg:"clickpayz1"; content: "302"; http_stat_code; content: "|0d0a0d0a|Loading..."; sid:1; rev:1;)

bh2-ngen

probe: GET /ngen/shrift.php
signature: alert tcp any any -> any any (msg:"bh2-ngen"; content: "200"; http_stat_code; content: "|0d0a|Content-Disposition: attachment|3b| filename=font.eot|0d0a|"; sid: 2, rev:1;)

Figure 5.2: Example fingerprints.

function and a signature.

5.3.3 Scanning Overview

We use two types of scans based on the target ranges: *Internet-wide* and *localized*. Internet-wide scans probe the entire IPv4 address space while localized scans probe selected ranges. Our localized scans explore the *provider locality* of the malicious servers. That is, whether the managers of a malicious family select a small number of hosting and ISP providers and install multiple servers in each, to amortize the cost of setting up a relationship with the provider (e.g., registering with a fake identity, setting up working VMs).

Using the seed servers as a starting point, a localized scan probes only the set of IP ranges belonging to the same providers that host the seed servers. Localized scans do not allow identifying the full infrastructure of a malicious family. However, they require sending only a very small number of probes, and quite frequently they still identify previously unknown servers.

We envision two different application scenarios for our active probing approach. Some entities like antivirus vendors, police, or national security agencies may want to use Internet-wide scans to identify all malicious servers of a family on the Internet. However, other entities like hosting providers or ISPs may want to simply scan their own IP ranges to identify malicious servers installed by their clients.

Scanners. Figure 5.1b shows the architecture of CyberProbe's scanning component. It comprises three scanners: a horizontal TCP scanner, a UDP scanner, and an application-layer TCP scanner (app-TCP). The horizontal TCP scanner performs a SYN scan on a given port, and outputs a list of hosts listening on that port. The UDP and app-TCP scanners send the fingerprint probes and collect or analyze the responses. For TCP fingerprints, CyberProbe first runs the horizontal scanner and then the app-TCP scanner on the live hosts found by the horizontal scanner. This allows reusing the results of the horizontal scanner for multiple scans on the same port. All 3 scanners can be distributed across multiple

scanner hosts. The receiver component of the UDP and appTCP scanners can output a network trace containing all responses or run Snort on the received traffic to output the set of hosts matching the fingerprint. Saving the network trace requires significant disk space (e.g., 50 GB for an Internet-wide HTTP scan), but enables further analysis of the responses.

Currently, our UDP and appTCP scanners probe one fingerprint at a time since different fingerprints, even if for the same family, may use different transport protocols and require scanning on different ports. The scanners can be easily modified to scan with multiple fingerprints if the target port and target hosts are the same and the fingerprints use the same transport protocol. However, an important goal of the scanning is to spread the traffic received by a target over time and each additional fingerprint makes the scan more noisy.

5.3.4 Malware Execution

Executing malware in a contained environment is a widely studied problem [185, 65, 66]. For active probing, the main goals are acquiring the malicious endpoints known to the malware sample (e.g., C&C servers and P2P peers) and collecting instances of the network traffic between the sample and the malicious endpoints. Since C&C servers are highly dynamic it is important to run the malware soon after collection to maximize the probability that at least one of the C&C servers is alive.

We use two containment policies for running the malware: *endpoint failure* and *restricted access*. The endpoint failure policy aborts any outgoing communication from the malware by sending error responses to DNS requests, resets to SYN packets, and sinking outgoing UDP traffic. This policy is designed to trick the malware into revealing all endpoints it knows, as it tries to find a working endpoint. The *restricted access* policy allows C&C traffic to and from the Internet, but blocks other malicious activities such sending spam, launching attacks, or clickfraud. This policy also resets any connection with a payload larger than 4 KB to prevent the malware to download and install other executables.

The malware is first run with the endpoint failure containment policy and a default configuration. If it fails to send any traffic, it is rerun with different configurations. For example, it is queued to be rerun on a different VM (e.g., on QEMU if originally run on VMWare) and for an extended period of time (e.g., doubling the execution timer). This helps to address malware samples that use evasion techniques for specific VM platforms, and to account for malware samples that may take longer to start its network communication.

5.4 Adversarial Fingerprint Generation

This section explains the process of generating fingerprints for a malicious family of interest starting from a set of network traces. Fingerprint generation comprises 4 steps. First, it extracts from the network traces the set of request-response pairs (RRPs) (Section 5.4.1). Then, it replays the requests to the live servers collecting their responses (Section 5.4.2). Next, it clusters RRPs with similar requests (Section 5.4.3). Finally, it generates signatures for each cluster (Section 5.4.4).

Benign traffic pool. Adversarial fingerprint generation also takes as input a pool of benign traffic used to identify which parts of the responses from servers in the family are distinctive, i.e., do not appear in benign traffic. This pool comprises three traces: two of HTTP and HTTPS traffic produced by visiting the top Alexa sites [186] and a 2-day long trace comprising all external traffic from a network with 50 users, captured at the network's border. We scan the traces with two IDS signature sets, verifying that they do not contain malicious traffic.

5.4.1 RRP Feature Extraction

From the network traces, CyberProbe first extracts the RRPs, i.e., TCP connections and UDP flows initiated by the malware or honeyclient towards a remote responder, and for which some data is sent back by the responder. Here, a UDP flow is the sequence of UDP packets with the same endpoints and ports that times out if no communication is seen for a minute. For each RRP, CyberProbe extracts the following feature vector:

$$\langle proto, sport, dport, sip, dip, endpoint, request, response \rangle$$

where *proto* is the protocol, *sport*, *dport*, *sip*, *dip* are the ports and IP addresses, and *endpoint* is the domain name used to resolve the destination IP. The *request* and *response* features represent the raw content of the request and response.

To extract the protocol feature CyberProbe uses protocol signatures to identify standard protocols commonly used by malware such as HTTP. Protocol signatures capture keywords present in the early parts of a message (e.g., GET or POST in HTTP) [187, 188]. They are able to identify the protocol even if it uses a non-standard port, and can also identify non-standard protocols on standard ports. Both situations are common with malware. For unknown application protocols, the protocol feature is the transport protocol.

RRPs for which the request endpoint is one of the top 100,000 Alexa domains [186] are discarded. This removes traffic to benign sites, used by malware to test connectivity and by exploit servers to download vulnerable software or redirect the user after exploitation.

In addition, it removes RRPs that have identical requests (excluding fields known to have dynamic data such as the HTTP Host header), to avoid replaying the same request many times. From the remaining RRPs CyberProbe builds an initial list of malicious endpoints. For this, it resolves each domain in the *endpoint* feature to obtain the current IP addresses the domain resolves to. It returns the union of the destination IP addresses and the resolved IPs.

5.4.2 Replay

The next step is to replay the requests in the RRPs extracted from the network traces to the known malicious endpoints.

The goal is to identify requests that lack replay protection, i.e., requests that if replayed to the same server at a later time or to another server of the family still incite a distinctive response. CyberProbe replays each unique request in the RRPs to every entry in the initial list of malicious endpoints, collecting the responses from endpoints that are alive.

The replay uses a commercial Virtual Private Network (VPN) that offers exit points in more than 50 countries, each with a pool of IP addresses, totaling more than 45,000 IPs. Using a VPN is important for two reasons. First, while the requests CyberProbe replays have a valid protocol syntax, there is still a small chance that they are replayed in an incorrect order or are no longer valid. If so, the managers of the malicious family could notice it and block the sender's IP address. In addition, we are interested in requests that generate a response without requiring any prior communication with the malicious server. Since CyberProbe replays all requests to each endpoint, it is important that the request being replayed is not influenced by any state that a previously replayed request may have set in the server. To achieve independence between replays, the replayer changes the VPN exit node (an thus its observable IP address) for each request sent to the same endpoint. Intuitively, a server keeps a separate state machine for each client that connects to it. Thus, by employing a previously unused IP address, the server will be in its initial state when it receives the replayed request.

Filtering benign servers. A common situation when replaying is that the IP address of a malicious server in the input network traces may have been reassigned to a benign server. Responses from benign servers need to be removed before building a signature to avoid false positives. To filter responses from benign servers, CyberProbe leverages the intuition that a benign server will not understand the replayed request and typically will ignore it (e.g., for a binary C&C request) or return an error (e.g., HTTP 404). Thus, as a first step CyberProbe removes from the replayed RRPs those with no response or where the response is an error (e.g., HTTP 4xx). However, a surprisingly large number of benign HTTP servers reply with a successful response (i.e., HTTP 200 OK)

to *any* request, possibly including a custom error message in the body of the response. Thus, a technique is needed to identify custom error messages without a priori knowledge of how they may look.

To address this challenge, CyberProbe also sends an HTTP request for a random resource to each potentially malicious HTTP server, leveraging the insight that if the responses from a server to the replayed request and to the random request are similar, most likely the server did not understand either request and the response is an error message.

CyberProbe considers two HTTP responses similar if they have the same result code, the same Content-Type header value, and similar content. Two non-HTML contents are similar if their MIME type as returned by the UNIX `file` tool is the same. For HTML documents, it uses an off-the-shelf similarity package [189], which serializes the HTML trees of the pages as arrays and finds the longest common sequence between the arrays. It measures similarity as:

$$d(a,b) = \frac{2 * length(LCS(array(a), array(b)))}{length(array(a)) + length(array(b))}$$

.

After removing errors and responses from benign servers the remaining RRPs are replayed twice more to the endpoints that responded, so that variations in the responses, e.g., changes in the HTTP Date and Cookie headers, are captured. The output of the replay phase are the remaining replayed RRPs. The original RRPs extracted from the network traces are not part of the output, i.e., only RRPs for which the request successfully replays are used to build the fingerprint. The unique endpoints in the output RRPs are the seed servers.

5.4.3 Clustering RRPs by Request Similarity

Next, CyberProbe clusters the RRPs by request similarity to identify instances of the same type of request across the network traces. This step prevents generating multiple fingerprints of the same type and enables producing more general fingerprints. We use two separate clusterings, a protocol-aware clustering for HTTP and a transport clustering for other protocols.

For HTTP, CyberProbe groups RRPs for which the requests have the same method (e.g., GET or POST) and satisfy the following conditions:

- **Same path.** The path in both URLs is the same and does not correspond to the root page.
- **Similar parameters.** The Jaccard index of the sets of URL parameters is larger than an experimentally selected threshold of 0.7. Parameter values are not included.

For other protocols, CyberProbe groups packets from the same transport protocol, with the same size and content, and sent to the same destination port. The output of the request clustering is the union of the traffic clusters output by the two clusterings. Each cluster contains the RRP feature vectors and the clusters do not overlap.

Probe construction function. From the requests in each cluster, CyberProbe produces a probe construction function. The probe construction function is basically one of the probes in the cluster where the value of a field may be replaced by the special TARGET and SET macros. The TARGET macro represents that the field needs to be updated with the value of the target endpoint during scanning, e.g., the HTTP Host header. The SET macro is used for fields that have different values in the cluster's requests. It represents that the value of the field can be chosen from this set when generating a new probe during scanning.

5.4.4 Signature Generation

For each cluster, signature generation produces signatures that capture parts of the responses that are unique to the family of interest, i.e., that are uncommon in the benign traffic pool. CyberProbe builds token-set payload signatures, which are supported by both Snort and Suricata.

A token set is an unordered set of binary strings (i.e., tokens) that matches the content of a buffer if all tokens in the signature appear in the buffer, in any order. The more tokens and the longer each token the more specific the signature.

Algorithm 1 describes the signature generation. Its salient characteristics are that when the protocol is known (e.g,. HTTP) the tokenization is performed on fields and that multiple signatures may be generated for each cluster. For each field in the responses in the cluster, it identifies distinctive tokens i.e., tokens with high coverage and low false positives. We define the *false positive rate* of a token in a field to be the fraction of responses in the benign pool that contain the token in the field, over the total number of responses in the benign pool. The *coverage* is the fraction of responses in the cluster with the token in that field, over the total number of responses in the cluster. A token is distinctive if it has a file coverage larger than 0.4 and a false positive rate below 10^{-9}.

Algorithm 1 can generate multiple signatures because distinctive tokens do not need to appear in all responses in the traffic cluster. This is important to handle noise in the cluster, e.g., from incorrectly labeled malware in the input traces. The get_distinct_fields function returns all fields in the response (or a single field if the protocol is unknown), except fields that contain dynamically generated data (e.g., the Date and Set-Cookie HTTP headers), as those fields should not be part of the signature. The tokenize function uses a suffix array [190] to extract tokens larger than 5 bytes that appear in the set of unique field values.

Algorithm 1 Signature Generation Algorithm

```
1  def tokenize_responses(cluster) {
2    tokens_info = []
3    # Get unique fields for responses in cluster
4    unique_fields = get_distinct_fields(cluster)
5    for field in unique_fields
6      # Get unique values for field
7      unique_values = get_distinct_fields_values(field)
8      # Tokenize unique field values
9      tokens = tokenize(unique_values)
10     for token in tokens
11       # Get feature vectors for responses with the token
12       vectors = get_responses(token)
13       # Add token
14       tokens_info.add(field,token,vectors)
15   return tokens_info
16
17 def refine_signature(tokens_info,curr_sig)
18   tinfo,rem_tokens_info =
19     get_token_max_overlap(tokens_info,curr_sig)
20   rsig = add_token(curr_sig,tinfo)
21   if cov(rsig) = cov(curr_sig)
22     refine_signature(rem_tokens_info,rsig)
23   if fp(curr_sig) < thres_fp
24     return curr_sig
25   refine_signature(rem_tokens_info,rsig)
26
27 def generate_signatures(cluster) {
28   signatures = []
29   tokens_info = tokenize_responses(cluster)
30   while true
31     # Find token that maximizes coverage
32     tinfo,rem_tokens_info,cov_increase =
33       get_max_coverage_token(signatures,tokens_info)
34     if cov_increase < thres_cov break
35     else
36       initial_sig = add_token(Ø,token_info)
37       refined_sig = refine(rem_tokens_info,initial_sig)
38       if refined_sig
39         signatures.add(refined_sig)
40
41   return signatures
42 }
```

5.5 Scanning

This section first describes general characteristics of our scanning such as the target ranges to scan, scan rate, scan order, and scanner placement. Then, it details the implementation of our horizontal, UDP, and appTCP scanners.

5.5.1 General Scanning Characteristics

Scan ranges. We perform 3 types of scans based on the ranges to be probed: *localized-reduced*, *localized-extended*, and *Internet-wide*. For Internet-wide scans, prior work has used different ranges that qualify as "Internet-wide" [191, 99, 101, 100]. These studies do not scan the full Internet IPv4 space (F), but rather

Full (F)	**Unreserved** (U)	**Allocated** (I)	**BGP** (B)
4.3B (100%)	3.7B (86%)	3.7B (86%)	2.6B (60%)

Table 5.1: Number of IPv4 addresses (in billions) for different Internet-wide target sets.

the non-reserved IPv4 ranges ($U \subseteq F$) [191], the IANA-allocated blocks ($I \subseteq U$) [99, 100], or the set of advertised BGP ranges ($B \subseteq I$) [101]. These ranges differ in their sizes, which are shown in billions of IP addresses in Table 5.1. The U and I ranges are nowadays the same as all the non-reserved IPv4 space has been allocated by IANA. In this work, for Internet-wide horizontal and UDP scans, we first collect the BGP ranges advertised the day of the scan from the RouteViews site [192]. Then, we union those ranges removing any route overlaps. The table shows that using the BGP information to exclude non-routable ranges reduces the scan range up to 40%.

Localized scans focus on IP ranges belonging to providers that have been observed in the past to host a server of the malicious family. To select the target ranges for localized scans we use the IP addresses of the seed servers and the BGP route information. For localized-reduced scans, we obtain the most specific BGP route that contains each seed's IP address, and output the union of those routes. For localized-extended scans, for each seed server we first obtain the most specific route containing the seed's IP. From each of those routes, we extract the route description, which typically identifies the provider that the route belongs to. Then, we query again the BGP information for the list of all other routes with the same description (i.e., from the same provider) and make their union our target set.

Scan rate. Nowadays, a well-designed scanner running on commodity hardware can send fast enough to saturate a 1 Gbps link (i.e., 1.4 Mpps) [104] and some work enables commodity hardware to saturate even 10 Gbps links [103]. Thus, a scanner often needs to be rate-limited to avoid saturating its uplink, disconnecting other hosts in the same network. In this work, for good citizenship we limit each horizontal and UDP scanner host to a maximum of 60,000 packets per second (26 Mbps), and each appTCP scanner host to a rate of 400 connections per second.

Scan order. Our horizontal and UDP scanners select which target to probe next using a random permutation of the target address space. Drawing targets uniformly at random from the target ranges mixes probes to different subnets over time, avoiding the overload of specific subnets [102]. To scan in random order, without needing to keep state about what addresses have already been scanned or are left to be scanned, our horizontal and UDP scanners use a linear congruential generator (LCG) [193]. Since the IP addresses output by the horizontal scanner are not sequential, the appTCP scanner does not use a LCG but

simply randomizes the order of the target IP addresses.

Whitelisting. The LCG iterates over a single consecutive address range. However, the BGP ranges to be scanned may not be consecutive. Also, we may need to exclude certain ranges, e.g., those whose owners request so. To address these issues, before probing a target, the horizontal and UDP scanners check if the target's IP is in a whitelist of IP addresses to scan, otherwise they skip it. The whitelist is implemented using a 512 MB bit array, where each bit indicates if the corresponding IP address needs to be probed. This ensures that checks are done in $O(1)$. Since most commodity hardware has a few GBs of memory this is a good tradeoff of memory for efficiency. For the appTCP scanner, which does not use an LCG, we simply remove IP addresses that should not be probed from the input target list.

Scanner placement. Multiple scanners can be used to distribute a scan. Since a single scanner may be able to saturate its uplink it is typically not needed to use multiple scanners on the same network. It is preferable to add them in separate networks with independent uplinks. All scanners use the same generator for the LCG. To split the target hosts between scanners, we assign each scanner a unique index from zero to the number of scanners minus one. All scanners iterate over the targets in the same order, but at each iteration only the scanner whose index matches the target IP modulo the number of scanners sends the probe.

5.5.2 Horizontal Scanner

An efficient horizontal scanner is fundamental to perform fast and resource-efficient scans because the large majority of IP addresses (97%–99% depending on the port) do not send responses to probes. Two important characteristics of our horizontal scanner are the lack of scanning state and the asynchronous sending of probes and receiving of responses.

Our horizontal scanner performs TCP SYN (or half-open) scans. While there exists different types of TCP scans [121], TCP SYN scans are arguably the most popular one because they can efficiently determine if a target host is listening on a port. They are also called half-open scans because they never complete a full TCP handshake. A SYN packet is sent to a target and if a SYNACK response is received, the scanner marks the target as alive and sends it a RST packet, which avoids creating state on the scanner or the target. A single SYN packet is sent to each target without retransmissions, which prior work has shown as a good tradeoff between accuracy (low packet loss on the backbone) and efficiency (avoiding doubling or tripling the number of probes) [101]. The horizontal scanner is implemented using 1,200 lines of C code and runs on Linux. It comprises a sender and a receiver module. Both modules are independent and can be run on the same or different hosts. We describe them next.

Sender. The sender uses raw sockets to send the probes. Raw sockets bypass the kernel network stack so that no state is kept for a probe. They prevent the kernel from doing route and ARP lookups, and bypass the firewall. When a SYNACK packet is received, the kernel automatically sends a RST packet since it is unaware of the connection. On initialization the sender creates a buffer for a raw Ethernet request. It fills all fields in the Ethernet, IP, and TCP headers except the destination IP address, source port, sequence number, and TCP and IP checksums.

Using a single buffer and caching most field values reduces memory accesses, increasing performance. The source IP is the IP address of the receiver. If the receiver runs on a separate host the sender spoofs the receiver's IP address. To enable the receiver to identify valid responses, the sequence number is filled with the XOR of the target IP and a secret shared between the sender and the receiver. The checksums can be computed on software or outsourced to the network card if it supports checksums on raw sockets.

The sender implements rate limiting by enforcing an inter-probe sleeping time. The Linux kernel does not provide fine-grained timers by default, so OS functions like *usleep* or *nanosleep* are too coarse for microsecond sleeps. Instead, the scanner deactivates CPU scaling, computes the sleeping time in ticks, and then busy-waits using the `rdtsc` instruction until it is time to send the next probe.

Receiver. The receiver is implemented using libpcap [194] and set to sniff all SYNACK packets. Note that the number of received packets is much smaller than the number of probes, e.g., only 2.6% of the advertised IPs listen on 80/tcp. Thus performance is less critical in the receiver than in the sender. Once the sender completes, the receiver keeps listening for a predefined time of 5 minutes to capture delayed responses. The receiver uses the shared secret, the acknowledgment number, and the source IP to check if the SYNACK corresponds to a valid probe. If so, it outputs the source IP to a log file of live hosts. There is no need to keep state about which IPs have already responded. Once the scan completes, duplicated entries due to multiple SYNACKs are removed from the log.

5.5.3 AppTCP & UDP Scanners

The appTCP and UDP scanners need to be able to send probes from different fingerprints, which may capture different application-layer protocols and message types. The probe construction function in a fingerprint abstracts the specificities of probe building from the scanner. Each probe construction function comprises two C functions. The first function is called during initialization and builds a default probe. Then, for each target host the appTCP or UDP scanner passes the target IP to the second function, which returns the TCP or UDP payload for the probe (e.g., updating the default probe with target-specific field values).

Type	Source	Families	Pcaps	RRPs	RRPs Replayed	Seeds	Fingerprints
Malware	VirusShare	152	918	1,639	193	19	18
Malware	MALICIA	9	1,059	764	602	2	2
Honeyclient	MALICIA	6	1,400	42,160	9,497	5	2
Honeyclient	UrlQuery	1	4	11	11	1	1

Table 5.2: Adversarial fingerprint generation results

Both scanners can apply the fingerprint by running Snort on the received traffic. In addition, they can collect the responses into a network trace and then run Snort offline on the trace. In our experiments we store the responses to enable post-mortem analysis and for collecting benign responses to enhance the benign traffic pool.

AppTCP scanner. The appTCP scanner is implemented using the libevent [195] library for asynchronous events, which is able to handle thousands of simultaneous non-blocking connections. It comprises 600 lines of C code plus the code that implements the probe construction functions for each fingerprint. It takes as input the list of live hosts identified by the horizontal scanner. To limit the connection rate the appTCP scanner operates on batches and the batch size limits the maximum number of simultaneously open connections. Reception is asynchronous, i.e., each received packet triggers a callback that reads the content from the socket. It sets a maximum size for a response since most classification functions operate on the early parts of a response. The default is 1MB but can be modified for any fingerprint. This limit is needed for servers that respond to any request with a large stream of data. For example, SHOUTCast [196] radio streaming servers may send a 1GB stream in response to an HTTP request for a random file.

UDP scanner. The UDP scanner uses the same architecture as the horizontal scanner, but builds instead UDP probes using the fingerprint's probe construction function. It comprises 800 lines of C code. The sender component also uses raw sockets, but embeds the secret in the source port instead of the sequence number. Similar to the appTCP scanner, the receiver component sets the maximum size of a response to 1MB.

5.6 Evaluation

This section presents the evaluation results for adversarial fingerprint generation (Section 5.6.1), our scanning setup (Section 5.6.2), scanning results (Sections 5.6.3 to 5.6.5), and detailed analysis of selected operations (Section 5.6.6).

5.6.1 Adversarial Fingerprint Generation Results

We obtain malware from two different sources: VirusShare [197] and the MALICIA dataset [46]. We run the malware on our infrastructure to produce the network traces used as input to the fingerprint generation. VirusShare malware is not classified, so we use a traffic clustering algorithm to split the executables into families [40]. The MALICIA dataset contains malware distributed through drive-by downloads, already classified into families, so clustering is not needed. For the exploit servers, we use network traces of the honeyclients collecting the malware in the MALICIA dataset. In addition, we add another exploit server family not present in MALICIA that we identify in URLQuery [157] and use a honeyclient to collect the network traces.

Table 5.2 summarizes the results of adversarial fingerprint generation. It shows the type and source of the network traces, the number of malicious families, the number of network traces processed, the RRPs in those traces, the RRPs replayed after filtering, and the number of seeds and fingerprints output. Overall, CyberProbe produces 23 fingerprints for 13 families: 3 exploit server families and 10 malware families. Of those, one fingerprint uses UDP and the rest use HTTP. The number of generated fingerprints is low compared to the number of network traces processed because some families have many traces (e.g., 700 for winwebsec) and because much malware connects to dead servers, which have likely been replaced by newer ones.

5.6.2 Scanning Setup

For the localized horizontal and UDP scans we use a single scanner at one of our institutions. This machine has 4 CPU cores running at 3.30GHz, a network connection at 1Gbps and 4GB of RAM. To distribute the Internet-wide horizontal and UDP scans across different hosts and locations we also rent 4 large instances in a cloud hosting provider. For the HTTP scans, we rent smaller virtual machines on virtual private server (VPS) providers and also use two dedicated hosts installed at one of our institutions. For the VPSes we select the cheapest instance type offered by each provider that satisfies the following minimum requirements: 1GHz, 512RAM, 100Mbps link, and 15GB hard drive.

Different providers may offer different virtualization technologies (e.g., XEN, OpenVZ, VMWare). The cheapest ones are often based on OpenVZ technology (starting at $4/month). In total we spent close to $600 on cloud hosting for the experiments in this paper.

The selected instances on different providers have different resources and those resources are sometimes not well specified. For example, some providers only specify the maximum amount of network traffic the instance can send over the rental period (e.g., 1TB/month), but do not specify the network interface bandwidth and whether they perform some additional rate-limiting of the VMs. To

HID	Type	Start Date	Port	Targets	SC	Rate(pps)	Time	Live Hosts
1	E	2013-03-12	8080	13,783,920	1	300	14.3h	193,667 (1.4%)
2	R	2013-03-26	80	4,096	1	60	1.2m	2,053 (50.1%)
3	E	2013-04-08	80	7,723,776	1	125	19.1h	316,935 (4.1%)
4	R	2013-04-14	80	24,576	1	200	2.4m	14,134 (57.5%)
5	E	2013-04-15	80	32,768	1	200	3.6m	14,869 (45.3%)
6	E	2013-04-17	80	1,779,965	1	125	3.9h	751,531 (42.2%)
7	E	2013-04-20	8080	19,018,496	1	900	6.0h	301,758 (1.6%)
8	R	2013-04-23	80	105,472	1	250	7.2m	8,269 (7.8%)
9	R	2013-04-28	80	668,672	1	5,000	2.4m	36,148 (5.4%)
10	I	2013-04-30	80	2,612,160,768	4	50,000	3.5h	67,727,671 (2.6%)
11	I	2013-04-30	8080	2,612,160,768	4	50,000	3.5h	239,517 (0.01%)
12	I	2013-07-01	80	2,510,340,631	5	50,000	2.9h	65,633,678 (2.6%)
13	I	2013-08-05	80	2,459,631,240	4	60,000	2.9h	63,534,118 (2.6%)

Table 5.3: Horizontal scanning results.

address the resource differences across VMs we split the target addresses proportionally to the hard drive and network bandwidth (when known) of the rented instances. This may result in some scanners being assigned larger ranges than others, e.g., 3 scanner hosts being used one with 50% and each of the other two with 25% of the total target addresses.

5.6.3 Horizontal Scanning

For TCP fingerprints, CyberProbe first performs a horizontal scan of the desired target ranges to identify hosts listening on the target port. Table 5.3 summarizes our horizontal scans. It shows the scan type, i.e., localized-reduced (R), localized-extended (E), or Internet-wide (I); the date of the scan; the target port; the number of target IP addresses scanned; the number of scanners used (SC); the sending rate for each scanner; the duration of the scan; and the number (and percentage) of live hosts found.

The first 9 scans are localized scans, targeting small ranges from 4,096 to 19 million IP addresses, and performed at very low scan rates. The goal of these localized scans was to test our scanning infrastructure, and to perform an initial evaluation of whether our hosting provider locality hypothesis holds (next subsection). The last 4 are Internet-wide scans, three on 80/tcp and one on 8080/tcp. Using 5 scanner hosts at a rate of 50,000 packets per second (pps), or 4 scanners at 60,000pps, it takes less than 3 hours for CyberProbe to perform an Internet-wide horizontal scan.

The Internet-wide scans found 2.6% of all advertised IP addresses listening on 80/tcp and 0.01% on port 8080. The 80/tcp scan on April 30th found 67.7 million hosts, the scan on July 1st 65.5 million, and the August 5th scan 63.5 million. The 8080/tcp scan found 239K live hosts. The difference on live hosts found between the 80/tcp scans is due to changes on the total size of the BGP advertised routes on the scan days. The live hosts intersection between the April 30th and July 1st 80/tcp scans is 43.9 million IPs (67%). That is, two thirds

ID	Start Date	Port	Fingerprint	Targets	HID	SC	Time	Resp.	Found	Known	New	VT	UQ	MD	VV
1	2013-01-08	8080	doubleighty	4K	-	1	62h	92%	5	4	1	0	3	1	0
2	2013-03-03	8080	doubleighty	193K	1	1	79m	91%	11	2	9	0	1	0	0
3	2013-03-26	80	winwebsec	2K	2	1	3m	96%	2	1	1	0	1	0	0
4	2013-04-08	80	winwebsec	316K	3	1	5.3h	22%	2	2	0	0	1	0	0
5	2013-04-15	80	blackrev	14K	4	1	18m	94%	1	1	0	0	0	0	0
6	2013-04-16	80	blackrev	14K	5	1	19m	94%	2	1	1	0	0	0	0
7	2013-04-17	80	bh2-adobe	751K	6	1	9.9h	55%	3	1	2	1	1	0	0
8	2013-04-17	8080	doubleighty	301K	7	1	5.1h	22%	4	2	2	0	0	0	0
9	2013-04-23	80	kovter-links	8K	8	1	8m	36%	2	1	1	1	0	0	0
10	2013-04-23	80	clickpayz1	8K	8	1	8m	31%	17	2	15	0	0	0	0
11	2013-04-28	80	clickpayz1	36K	9	1	35m	38%	17	15	2	1	0	0	0
12	2013-07-06	80	bh2-adobe	65.6M	12	3	24.7h	75%	10	1	9	3	1	0	0
13	2013-07-11	80	clickpayz1	65.6M	12	3	26.5h	74%	22	17	5	7	0	0	0
14	2013-07-16	80	clickpayz2	65.6M	12	3	26.6h	76%	25	12	13	5	1	0	0
15	2013-07-20	80	kovter-pixel	65.6M	12	3	26.5h	72%	7	1	6	4	0	0	0
16	2013-07-22	80	bh2-ngen	65.6M	12	3	24.6h	72%	2	1	1	0	0	0	0
17	2013-07-25	80	optinstaller	65.6M	12	3	24.5h	71%	18	1	17	3	2	0	1
18	2013-07-27	80	bestav-pay	65.6M	12	4	15.6h	70%	16	2	14	6	5	0	0
19	2013-07-29	80	bestav-front	65.6M	12	4	13.2h	*62%	2	1	1	1	1	0	0
20	2013-07-31	80	ironsource	65.6M	12	4	13.1h	*59%	7	1	6	5	5	0	0
21	2013-08-05	80	soft196	65.6M	12	2	23.8h	71%	8	1	7	6	5	0	0
22	2013-08-06	80	winwebsec	63.5M	13	3	15.6h	85%	11	0	11	7	3	0	0
							TOTALS:		194	70	124	50	30	1	1

Table 5.4: HTTP scan results.

of the 80/tcp live hosts are stable for over 2 months. The others change due to servers being added and removed, and IP assignments changing over time. This indicates that we can trade coverage for politeness by reusing the results of a horizontal scan for multiple application-layer scans. This slowly decreases coverage over time, but minimizes the number of horizontal scans needed.

The results show that the 80/tcp localized scans find from 4.1% up to 57.5% of live hosts on the targeted ranges, well above the 2.6% Internet average. This happens because most seeds are located on cloud hosting providers, which are being abused to install malicious servers. Thus, localized scans focus on hosting services that house significantly more web servers than other residential or enterprise networks.

5.6.4 HTTP scanning

Table 5.4 summarizes our HTTP scans, which probe the set of live hosts found by the horizontal scans, identifying malicious servers matching a fingerprint. The left part of the table shows the scan configuration: the scan date, the target port, the fingerprint used, the number of hosts scanned, the horizontal scan that found them (HID), and the number of scanners used (SC). We have used CyberProbe to perform 22 scans using 14 fingerprints. Note that we have yet to scan for the remaining fingerprints.

The middle part of Table 5.4 shows the results: the scan duration; the response rate (Resp.), i.e., the percentage of targets that replied to the probe; the number of malicious servers found; the number of malicious servers found previously known to us, i.e., seeds and servers found by a prior scan for the same family; and the

number of found servers previously unknown. CyberProbe takes on average 14 hours to perform an Internet-wide HTTP scan using 4 scanners and 24 hours using 3 scanners.

The results show that the 22 scans identified 194 servers, of which 151 are unique. Starting from 15 seeds CyberProbe identified 151 unique malicious servers, achieving a 10x amplification factor. Of the 22 scans, 91% (20) find previously unknown malicious servers, the exception being two localized scans for winwebsec and blackrev. The 11 localized scans find 66 servers (34 new), an average of 6 servers found per scan. The 11 Internet-wide scans find 128 servers (72 new), an average of 11.6 servers found per scan. While Internet-wide scans find more servers per scan, if we normalize by the number of targets scanned, localized scans find an abnormally high number of malicious servers. This verifies our provider locality hypothesis: cybercriminals are installing multiple servers on the same providers. Once they establish a relationship with a hosting provider they are likely to reuse it, minimizing the effort for locating new providers, learn their procedure to install new servers, and create fake identities for registration (e.g., Paypal accounts).

Coverage. The right part of Table 5.4 shows the number of servers found by CyberProbe that were already known to 4 popular anti-malware cloud services: VirusTotal (VT) [25], URLQuery (UQ) [157], Malware Domain List (MD) [156], and VxVault (VV) [183]. All these cloud services use crowd-sourcing to collect potentially malicious executables and URLs. Their coverage depends on the number and volume of their contributors. Some of them have infrastructures to automatically visit submitted URLs (VirusTotal and URLQuery) and execute the submitted malware to collect behavioral information (VirusTotal). The collected information is dumped into databases and public interfaces are provided to query them. As far as we know, Malware Domain List and VxVault follow a similar process to populate their databases from submissions, but the process is manually performed by volunteers. We select these specific databases because they are popular and allow querying by IP address, while other public databases, e.g., Google Safe Browsing [37], only enable URL queries.

The best coverage is achieved by VirusTotal, which knows 25.7% of the servers found by CyberProbe (50/194), followed by URL Query with 15.5% (30/194). Malware Domain List and VxVault only know one of the servers each, an abysmal 1.1%. Overall, CyberProbe finds 4 times more malicious servers than the best of these services. The best coverage among those 4 services is achieved by those using automatic processing (VirusTotal and URLQuery). Although those 2 services have huge URL and malware collections, they still achieve limited coverage. Those services could be combined with our active probing approach so that when they discover new seed servers and families, fingerprints are automatically generated and scanned to identify other family servers. This would significantly increase their coverage and bring them closer to Internet scale.

Type	Start Date	Port	Fingerprint	Targets	SC	Rate(pps)	Time	Found
R	2013-03-19	16471	zeroaccess	40,448	1	10	1.2h	55 (0.13%)
I	2013-05-03	16471	zeroaccess	2,612,160,768	4	50,000	3.6h	7,884 (0.0003%)

Table 5.5: C&C UDP scanning results.

Our results show that CyberProbe achieves 4 times better coverage than current approaches for identifying some malicious server families. However, there exist some implementation and deployment trade-offs that limit CyberProbe's coverage, which could be even higher. For example, we reuse the results of horizontal scans over time to minimize the number of horizontal scans.

In particular, scans 12–21 target the live hosts found by horizontal scan 12 in Table 5.3. As expected, the response rate of these HTTP scans decreases over time as those results become stale. However, we find that the response rate decreases slowly, from 75% to 70% 3 weeks later. Scans 19–20 show a lower response rate because they include 2 instances that (unaware to us) were rate-limited by the provider. Removing the instances from that provider the response rate was 70% for scans 19–20. This slow decrease justifies the trade-off of coverage for politeness. However, in other situations it may be possible or better to perform more aggressive scanning. We further discuss scan completeness in Section 5.7.

False positives. The majority of the fingerprints do not produce false positives. However, the bh2-adobe fingerprint, which captures a fake Adobe webpage (further explained in the next section) produces one false positive. It corresponds to a web server with a page that contains license keys for popular software from Adobe. The authors seem to have copied parts of the Adobe webpage that are part of our signature. We have not verified if the license keys work.

5.6.5 UDP scans.

One of the fingerprints produced by CyberProbe was for the UDP-based P2P protocol used by the ZeroAccess botnet.

According to an analysis by Sophos [31], this P2P protocol has two types of peers: remotely reachable supernodes with public IP addresses and normal nodes behind NATs. There are two distinct ZeroAccess botnets, each using two ports for the P2P C&C (for 32-bit and 64-bit infected hosts). The executed malware was from one of the 32-bit botnets operating on port 16471/udp. The fingerprint captures a `getL` command in which a peer requests from a supernode the list of other supernodes it knows about, and the corresponding `retL` response where the supernode returns a subset of its peers.

Table 5.5 summarizes the UDP scans. A localized-restricted scan on 40,488 IPs belonging to a residential US provider was first used to test the fingerprint. It identified 55 supernodes, a response rate of 0.13%. A later Internet-wide scan

Operation	Finger prints	Seeds	Servers	Prov.	Provider Locality
bestav	3	4	23	7	3.3
bh2-adobe	1	1	13	7	1.8
bh2-ngen	1	1	2	2	1.0
blackrev	1	1	2	2	1.0
clickpayz	2	2	51	6	8.5
doubleighty	1	1	18	9	2.0
kovter	2	2	9	4	2.2
ironsource	1	1	7	4	1.7
optinstaller	1	1	18	2	9.0
soft196	1	1	8	4	2.0
TOTAL	14	15	151	47	3.2 (avg.)

Table 5.6: Server operations summary.

found 7,884 supernodes (0.0003% response rate). Since the response comprises a list of advertised supernodes, we extract their addresses from the responses and compute their union across all 7,884 responses. There were 15,943 supernodes advertised at the time of the Internet-wide scan. Of those, 6,257 (39%) were found by our scan and 9,686 (61%) were not reachable. The unreachable hosts could have been cleaned, be offline, or have changed IP address (e.g., mobile devices, DHCP). Our scan also finds 1,627 supernodes alive but not advertised. This could be due to supernodes only responding with a partial list of peers and due to nodes that have changed IP since advertised. One day after the Internet-wide scan only 19% of the 15,943 advertised supernodes were alive. This high variability has previously been observed to make it easy to overestimate the size of a botnet using IP addresses [182]. However, the speed of active probing makes IP variability a smaller issue, enabling an easy and quite accurate method for estimating the size of P2P botnets.

5.6.6 Server Operations

Table 5.6 summarizes the 10 server operations analyzed. It shows the number of fingerprints from the operation used in the scans, the seeds used to generate the fingerprints, the number of unique servers found, the number of providers hosting the servers found, and the ratio of servers per provider of the operation. Overall, these operations host an average of 3.2 servers per provider. The remainder of this section details selected operations.

BestAV. Best AV is an affiliate pay-per-install program that has been operating since at least August 2010 distributing the `winwebsec` family, which encompasses multiple fake AV brands [198]. Nowadays, BestAV manages 3 programs: the

winwebsec fake antivirus, the `Urausy` ransomware, and another unknown family [199]. We have 3 fingerprints related to the BestAV operation. Two of the fingerprints were generated by running winwebsec malware. They capture C&C servers (winwebsec) and payment servers (bestav-pay). The Internet-wide scans reveal 16 payment servers and 11 C&C servers. There is strong provider locality as they use 4 cloud hosting providers for the 27 servers. Provider A hosts 6 payment and 5 C&C servers, provider B 9 payment and 4 C&C servers, provider C 2 C&C servers, and provider D the remaining payment server. The 3 providers used for payment servers provide only dedicated server hosting, which indicates that the managers do not want external services colocated with their payment infrastructure. The third fingerprint captures web servers used by the affiliates for checking statistics and collecting their installers. We manually generated this fingerprint after reading an external analysis, which identified 2 live web servers [198]. One of them was alive and we use it as seed server. An Internet-wide scan reveals a second server for the affiliates that we have not seen mentioned anywhere else. This server does not show any domain associated on different passive DNS databases, so we believe it is kept as backup in case the main one is taken offline.

Blackhole2-adobe. The bh2-adobe fingerprint captures a malware distribution operation through drive-by downloads that has been ongoing since at least October 2012 [200]. This operation configures their Blackhole 2 exploit servers to redirect users to a fake Adobe webpage if exploitation fails, which prompts the user to install a malicious Flash player update. The webpage has been copied from Adobe but resides on a different resource. Our fingerprint captures that an Adobe server will reply to that resource with a 404 error, but the exploit servers will successfully return an Adobe download page. Our Internet-wide scan on July 6 found 10 live servers, all in cloud hosting services. This supports recent results that the majority of exploit servers are abusing cloud hosting services [44]. Of the 10 servers, 3 were already known to VirusTotal. Another 2 were identified by VirusTotal four days later, and a third one 13 days after CyberProbe detected it. This shows how CyberProbe can find servers earlier than other approaches.

Blackhole2-ngen. The bh2-ngen fingerprint captures another drive-by download operation, distinguishable because their URLs contain the `/ngen/` directory. The Internet-wide scan reveals only 2 servers. To verify that CyberProbe does not miss servers we examine the URLQuery database. It contains 10 exploit servers with the /ngen/ string since October 2012. Since then, the database contains at most three servers operating on the same period of time. None of those 10 servers are located in known hosting providers, which makes us think they are using their own hosting. The new server CyberProbe found on July 7 is still not in URLQuery. It is the only server hosted on a known dedicated server hosting provider. We hypothesize it is either a test server or has not yet been set to receive traffic.

Doubleighty. The doubleighty family uses an unknown exploit server with a fixed resource in the landing URL: `/forum/links/column.php`. CyberProbe identifies 18 distinct servers in 3 localized scans with strong provider locality as two cloud hosting providers host 61% of the servers. After the March 3 scan, we used a honeyclient to visit the 9 new servers found. Seven of them exploited the honeyclient but two did not. We set the honeyclient to periodically visit those 2 servers. One month later (April 4) one of them started distributing malware. This shows that the server was installed much earlier than it started being used. It also shows that active probing can sometimes identify stand-by servers, before they exhibit their malicious behavior.

Kovter. Kovter is a ransomware family that blocks the infected computer and displays a police warning on the screen telling the user it needs to pay a fine to have it unlocked.

CyberProbe produced two fingerprints for different C&C messages used by the malware. We performed one localized scan using the kovter-links fingerprint that found 2 servers and an Internet-wide scan 3 months later using the newer kovter-pixel fingerprint that found 7. Thus, the C&C infrastructure has a high level of redundancy. One of the servers appears in both scans so it has been alive for at least 3 months. It is located in a German cloud hosting provider. Overall, the 8 distinct servers are distributed among 4 cloud hosting providers.

Clickpayz Clickpayz[1] is an online service that sells clicks. Quoting them: "clickPAYZ has a network of search sites with 10s of millions of people searching for everything under the sun". Some files in our malware datasets send clicks to their servers and the two fingerprints produced by CyberProbe seem to correspond to traffic sent by two different affiliates. The 39 unique servers identified by both fingerprints are click redirectors belonging to Clickpayz. They are located on 6 cloud hosting providers. Clickpayz managers are either unaware that their affiliates send them clicks via files flagged as malicious by different antivirus, or simply do not care. However, their claim of having tens of millions of people searching their sites is dubious and their site only provides an email address as contact information, typically a sign of dark objectives.

5.7 Discussion

5.7.1 Ethical Considerations

Internet scanning has been carried out many times for different research goals [98, 99, 101, 100, 104]. Still, the unsolicited nature of the probes makes some targets consider it offensive. We take ethical considerations seriously in our study. For

[1] https://www.clickpayz.com/

our horizontal scanning, we follow the recommendations of prior work, notably those by Leonard and Loguinov [101] who study how to perform maximally polite horizontal scans. We adopt their proposals of mixing the scanner IP addresses, setting up forward and backward DNS records for the scanners, running a web server on the scanners with a page explaining that the probing is part of a research project, and removing from the scan whitelist the ranges of owners that complain about our probing and are willing to share their IP ranges. Overall, we have removed from the whitelist 106K IP addresses. In addition, we limit the probing rate of our horizontal scanners to 60,000pps, well below their maximum rate. We also manually vet the generated fingerprints before scanning to make sure they do not contain attacks and will not compromise any host. Furthermore, we work with our system administrators to minimize the impact on the local network (e.g., bypass the firewall / IDS) and to quickly answer any complaints.

No prior literature details how to perform application-layer probing of malicious servers. Our HTTP probing is not malicious, it simply sends a request, which we have manually vetted first, and collects a response from a target server. However, our requests mimic those of malicious families, and often request inexistent resources from web servers. Thus, they may be considered suspicious or malicious by server owners, or may trigger IDSes loosely configured to match traffic on both directions. Overall, out of 11 VMs that we use for HTTP probing, 2 of them got suspended for "malicious" behavior. We did not get a warning from those providers, but found out when trying to access the instances. In addition, we received 3 emails warning us that our VMs may have been compromised. The communications from the providers and those received by the system administrators of our institutions show that the majority of the complaints come from web honeypots that do not advertise their IP addresses and consider any received traffic malicious. A less frequent reason are owners thinking our scanner hosts have been infected or are attacking them. Similar to the horizontal scanning, when the providers let us know their IP ranges, we avoid further probing.

Finally, it is worth noting that our scanning does not collect and publicize any sensitive information on remote networks.

5.7.2 Future Improvements

Completeness. Our current implementation is not guaranteed to find all servers forming the infrastructure of a family. There are two reasons for this. First, there are some families for which we cannot generate a fingerprint (e.g., their traffic cannot be replayed) or for which we may only be able to generate fingerprints for some of the server types they use (e.g., for the C&C server but not for their web servers). Second, our implementation has some limitations that limit our coverage. In particular, we have limited scanning capacity and are not able to run all fingerprints for a family simultaneously. In addition, we reuse the results of horizontal scans. This makes our probing more polite but reduces coverage

slowly over time.

Complex protocol semantics. One limitation of our fingerprint generation approach is that a replayed request may fail to incite a response from a remote server, e.g., if a field in the request should be a checksum of the sender's IP address or if the request is encrypted using the IP address as initialization vector. Such semantic information cannot be easily obtained from the network traffic, but prior work extracts it from a binary that implements the protocol [201, 202]. For cases where a binary is available, e.g., with malware, we plan to integrate binary analysis techniques into our approach.

Shared hosting. Some types of web hosting such as shared hosting and content delivery networks (CDNs) involve installing multiple domains on the same web server under the same IP. Here, the web server requires the presence of a domain name in the Host header to route the request, as two domains on the same server may define the same resource (e.g,. index.html). This is problematic for our scanning as we do not know the domains hosted on each probed IP address. However, malicious servers rarely use shared hosting services because those services are *managed*, i.e., the web server owner installs the content for the clients, which is problematic if the content is C&C software or an exploit kit. We could leverage passive DNS databases to identify domains hosted on an IP address to be probed. Some challenges that we foresee are the current limited coverage of such databases and the large amount of queries needed to complete a scan.

Making the probes identifiable to selected parties. Whenever we get a complaint on our probing we ask the reporters for the IP ranges they own and we remove them from the whitelist. However, some reporters may not want to disclose their IP ranges, e.g., if they run web honeypots whose addresses should remain secret. For those cases, we could embed a secret in our probes and disclose it to selected parties. For example, we could fix the secret used to compute the sequence number of our TCP probes and reveal it to reporters so that they can check if a received probe was sent by CyberProbe. Whenever the secret is updated we would need to notify all reporters.

5.8 Conclusion

In this paper, we have proposed a novel active probing approach for detecting malicious servers and compromised hosts that listen for (and react to) incoming network requests. Our active probing approach sends probes to remote hosts and examines their responses, determining whether the remote hosts are malicious or not. Compared with existing defenses, it is fast, cheap, easy to deploy, and

achieves Internet scale. It identifies different malicious server types such as C&C servers, exploit servers, payment servers, and click redirectors, as well as malware that listens for incoming traffic such as P2P bots.

We have implemented our active probing approach in a tool called CyberProbe, which implements a novel adversarial fingerprint generation technique, and 3 scanners. We have used CyberProbe to build fingerprints for 13 malicious families. Using those fingerprints, CyberProbe identifies 151 malicious servers and 7,881 P2P bots through 24 localized and Internet-wide scans. Of those servers 75% are unknown to 4 databases of malicious servers, indicating that for some families CyberProbe can achieve up to 4 times better coverage than existing techniques. Our results also reveal an important *provider locality* property: cybercriminals host an average of 3.2 servers on the same hosting provider to amortize the cost of setting up a relationship with a provider.

6 RevProbe: Detecting Silent Reverse Proxies in Malicious Server Infrastructures

6.1 Preamble

In this chapter we reproduce the content of the paper "RevProbe: Detecting Silent Reverse Proxies in Malicious Server Infrastructures". This work presents a tool codenamed RevProbe that by using actibe probing techniques detects silent reverse proxies. At the time of the writing this work has been submitted to NDSS 2016, Antonio Nappa has been the leading author of this paper.

6.2 Introduction

Attackers are constantly looking for mechanisms to protect their server infrastructure. A popular approach is to introduce indirection by adding intermediate layers of servers that simply forward traffic between the clients (e.g., bots, victims) and the final servers (e.g., C&C, exploit servers). Those intermediate servers are most exposed and hide the final servers that attackers closely manage and that store the most sensitive data. They can be hosted on infected machines and cheap cloud VMs and thus are easy to replace. And, they can distribute traffic across multiple final servers, making the malicious server infrastructure more resilient to take downs.

Different types of intermediate servers can be used to introduce indirection in malicious infrastructures. Botnet operators leverage publicly reachable computers in the botnet to serve as proxies [203, 204], HTTP redirectors are widely used in drive-by downloads [57], and traffic delivery systems (TDSes) aggregate traffic towards exploit servers [155]. Another type of intermediate servers are *reverse proxies*, which are set up by Web service administrators to interpose the communication between clients and the Web service servers. They differ from forward and ISP proxies [106] in that they are specific to one Web service and act as

the Web service endpoints. Reverse proxies are used in benign Web services and content delivery networks (CDNs) for load balancing traffic across servers [205], caching content [205], and filtering attacks [206].

An instance of reverse proxies are *silent reverse proxies* that do not reveal their proxy role. Silent reverse proxies hide the existence of other Web servers behind them. There have been reports of silent reverse proxies being used in malicious server infrastructures [207], but their prevalence is yet unknown. Currently, there is no effective tool to identify silent reverse proxies. While some tools exist (e.g., [112, 109, 108]) they fail to detect silent reverse proxies in common configurations. Furthermore, no tool details the hierarchy of servers hiding behind a silent reverse proxy, e.g., the number of servers behind a load balancer.

In this work we present RevProbe, a state-of-the-art tool for automatically detecting silent reverse proxies and identifying the server infrastructure behind them. RevProbe uses an active probing approach that sends requests to a remote target IP address and analyzes the responses for discrepancies and leaks indicating that the IP address does not correspond to a single server but to a reverse proxy with other servers behind. When it detects a reverse proxy, it outputs a *reverse proxy tree* capturing the hierarchy of servers it detected behind the reverse proxy. When possible, the identified servers are tagged with their software package, version, and IP address.

We design novel techniques for detecting silent reverse proxies based on discrepancies they introduce in the traffic such as extracting time sequences from the HTTP Date header and using the structure of default error pages to identify discrepancies in Web server software. We have also performed a comprehensive study of existing tools for reverse proxy detection (and Web server fingerprinting) and the techniques they use. We incorporate those techniques as well as our novel techniques into RevProbe.

RevProbe can be used by security analysts for taking down malicious infrastructures, counterintelligence, and attribution. It can be combined with recent active probing tools for detecting malicious servers [208, 87] to determine if detected servers are reverse proxies or final servers. RevProbe has also important applications on benign server infrastructures. It can be used during penetration testing to identify vulnerable servers hiding behind a reverse proxy and vulnerabilities introduced by the reverse proxy; for asset management; for auditing Web application firewall rules; for security compliance testing (as reverse proxies may be forbidden by the hosting network policy); for measuring and optimizing performance; and for cloud cartography [209].

We evaluate RevProbe on both controlled configurations and live benign and malicious websites. To measure its accuracy in comparison with other reverse proxy detection tools, we first apply RevProbe and 6 other tools on 36 silent reverse proxy configurations. RevProbe perfectly recovers the reverse proxy tree in 77% of the configurations, and the presence of a reverse proxy in the remaining 23% configurations. No other tool is a close competitor. The clos-

est tool is lbmap [108], which recovers the correct reverse proxy tree in 33% of the configurations and a reverse proxy in another 17%, but it only detects the less popular HAProxy [210] and Pound [211] reverse proxies, missing completely Apache [212] and Nginx [213] acting as reverse proxies.

Then, we use RevProbe to perform the first study on the usage of silent reverse proxies in both benign and malicious Web services. We apply RevProbe on the top 10,000 Alexa domains [186] and 8,512 malicious domains from MalwareDomains [214].

Our results show that 16% of active IPs in malicious Web infrastructures and 20% in benign infrastructures correspond to reverse proxies. The vast majority of malicious reverse proxies (92%) are silent to hide the existence of servers behind them, compared to 55% of benign reverse proxies. Reverse proxies are predominantly used to load balance traffic among multiple servers in both malicious (83%) and benign (87%) infrastructures. The dominant reverse proxy tree (78% of malicious trees and 86% of benign) is a Web load balancer with 2–6 servers behind in malicious infrastructures, and up to 30 servers behind in benign infrastructures, followed by a reverse proxy with one server behind (17% malicious trees, 13% benign), but more complex hierarchies also exist.

Contributions:

- We present RevProbe, a state-of-the-art active probing tool for detecting silent reverse proxies and identifying the server infrastructure hiding behind them.
- We design novel techniques for detecting reverse proxies based on discrepancies they introduce in the traffic such as extracting time sequences from the HTTP Date header and using the structure of default error pages to identify discrepancies in Web server software.
- We perform a comprehensive study of existing tools for reverse proxy detection (and Web server fingerprinting) and the techniques they use. We incorporate those techniques as well as our novel techniques into RevProbe.
- We compare the accuracy of RevProbe with 6 other reverse proxy detection tools on 36 silent reverse proxy configurations, showing that RevProbe outperforms existing tools.
- We apply RevProbe to perform the first study on the use of silent reverse proxies in malicious (and benign) infrastructures. Our results shows that 16% of malicious active IPs correspond to reverse proxies, that 92% of those are silent, and that they are predominantly used to load balance connections across multiple servers.

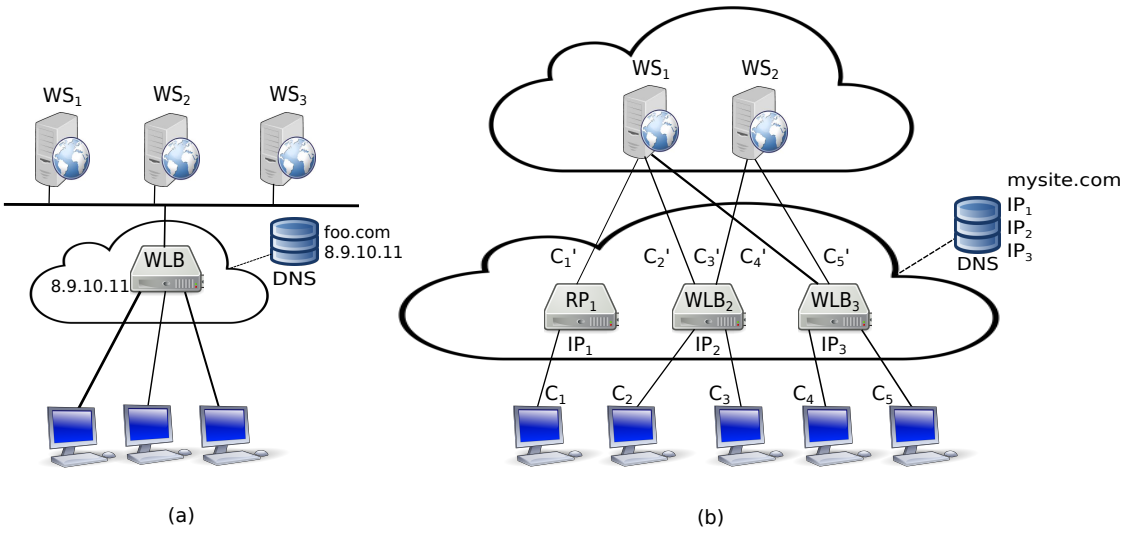


Figure 6.1: Reverse proxy usage examples.

6.3 Overview and Problem Definition

A proxy interposes on the communication between a client and a server, splitting the communication into two TCP connections: client to proxy and proxy to server. A proxy is *explicit* if it requires the IP address and port of the proxy to be specified in the client application (e.g., browser) and *transparent* otherwise.

A *reverse proxy* is purposefully setup by the owner of a Web service to mediate the HTTP communication from clients to the Web service servers. Thus, it is specific to a Web service as opposed to a *forward proxy* that is often setup in a local network and mediates communication of local clients to any remote service. It is also different from an *ISP proxy* [106], which intercepts all HTTP communication from ISP clients to any Web service. Reverse proxies are by definition transparent because they do not require client configuration. They are typically located across the Internet from the clients and can be hosted in the same subnet as the final servers as well as in a remote subnet across the Internet.

Reverse proxies are advertised as the service's endpoints. When a client arrives at a reverse proxy it looks like it is arriving at the server because the IP address it uses to contact the service is the IP address of the reverse proxy. If a domain name is used to advertise the service, the domain will resolve to the IP address of the reverse proxy. Thus, reverse proxies require the collaboration of the service owner to be installed.

There exist specialized types of reverse proxies. A *Web load balancer* (WLB) is a reverse proxy that load balances requests across multiple servers according to some policy (e.g., round-robin, least-loaded); a *Web application firewall* (WAF) filters requests to remove potential attacks; and a *Web reverse cache* (WRC)

accelerates communication by temporarily storing content from the servers responses, and later serving requests from the cached copy.

A *silent reverse proxy* tries to hide the presence of servers behind it. It does not purposefully signal its role as reverse proxy, e.g., by introducing a Via header in the HTTP response [215], thus hiding the existence of servers behind it. Without an explicit signal, the silent reverse proxy looks like the server.

Reverse proxies are used in benign Web services and content delivery networks (CDNs). But, (silent) reverse proxies can also be used in malicious infrastructures for hiding the final servers. In malicious infrastructures, the reverse proxies are the most exposed components because their IP addresses are visible, but their value is lower compared to final servers as they may not store any essential information and simply forward traffic. If the reverse proxy is taken down, little sensitive information may be obtained by the investigators. The attackers can replace the reverse proxy with another one and use the early warning to move their final servers. In addition, reverse proxies can use cheaper hosting (e.g., infected machines, cloud VMs) while attackers can make a larger investment to protect the final servers (e.g., bullet-proof hosting).

Figure 6.1 shows two usage scenarios for reverse proxies. In Figure 6.1a one reverse proxy is used as a WLB to distribute connections across 3 final servers on the same local network. Figure 6.1b captures a more robust server infrastructure with a line of reverse proxies and WLBs forwarding traffic to final servers that may be hosted somewhere else on the Internet. Some reports (e.g., [207]) link this kind of infrastructure to exploitation-as-a-service models [28] where the final servers would be exploit servers and the reverse proxies would hide them from the victims.

There exists other middleware that malicious Web infrastructures can use for introducing indirection and load balancing traffic across malicious servers. For example, drive-by downloads heavily rely on HTTP redirectors to route traffic to exploit servers [57]. A special case of HTTP redirectors are traffic delivery systems (TDSes) that use a number of configurable rules to decide where to route requests [155]. HTTP redirectors (and TDSes) differ from reverse proxies in that once the redirection happens the rest of the communication does not go through the redirector. They are also not silent as their IP address is visible to a client.

6.3.1 Problem Definition

In this work we develop a tool for detecting silent reverse proxies. We consider two alternative definitions of our problem: strict (simpler) and generalized (harder).

Strict problem definition. Given the IP address and port of a remote Web server $\langle ip, port \rangle$, output true if $\langle ip, port \rangle$ corresponds to a reverse proxy, false

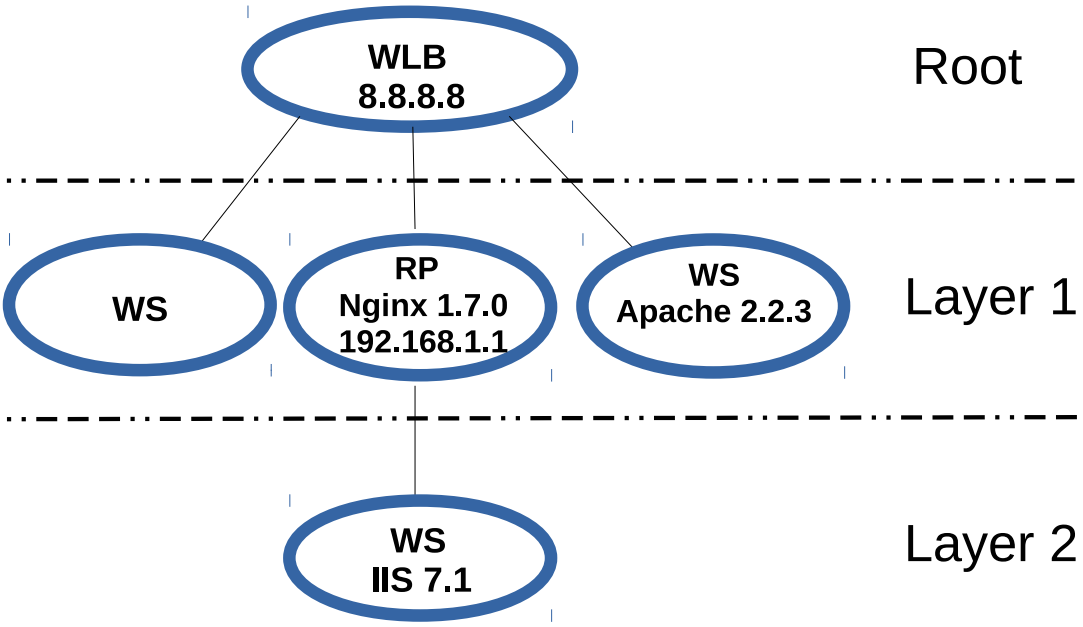


Figure 6.2: Reverse proxy tree example.

otherwise. This definition only identifies the existence of a reverse proxy; it does not attempt to identify the server infrastructure hiding behind it.

Generalized problem definition. Given the IP address and port of a remote Web server $\langle ip, port \rangle$, determine the server infrastructure behind it. The goal is to output a *reverse proxy tree* where each node corresponds to a Web server. The root node corresponds to the input pair $\langle ip, port \rangle$, internal nodes correspond to other reverse proxies, and leaf nodes correspond to final servers. A node with children is a reverse proxy and a node with multiple children is a Web load balancer. Each node is annotated with its type: reverse proxy (RP), Web load balancer (WLB), or final Web server (WS). A node can have 3 optional attributes: Web server software (e.g., Apache, Nginx), Web server version (e.g., 2.2.3 for Apache), and IP address.

Figure 6.2 shows an example reverse proxy tree. It shows that the target IP address (8.8.8.8) corresponds to a WLB that load balances traffic across 3 servers in layer 1. Of those, two are final servers and another is a reverse proxy to another final server at layer 2. Some nodes have the optional attributes software package, software version, and IP address.

The goal of RevProbe is identifying whether a given $\langle ip, port \rangle$ corresponds to a reverse proxy (strict definition) and, if so, recover the reverse proxy tree (generalized definition). The goal is not to recover the optional node attributes (software package, version, IP address). However, RevProbe will annotate the tree nodes with software, version, and IP address if it happens to recover that information during its processing.

Both problem definitions take as input an IP address. If the input is instead a domain, our approach first resolves the domain into a list of IP addresses. If the input is a URL, it follows all HTTP redirections and then resolves the final domain in the redirection chain to obtain a list of IP addresses. Our reverse proxy detection is then applied separately for each IP address.

A related problem is Web server fingerprinting [115, 116, 117] (WSF), which aims to recover the software package and version running at a given $\langle ip, port \rangle$ endpoint. Current WSF approaches assume the endpoint corresponds to a single server, which is not true with reverse proxies. For example, existing WSF tools will incorrectly label endpoints that correspond to Web load balancers where the received responses may come from multiple servers potentially running different software.

6.3.2 Approach Overview

We assume only black box access is available to the remote Web service identified by $\langle ip, port \rangle$. The code of the Web service is not available in any form. Thus, we use active probing techniques that send requests to $\langle ip, port \rangle$, collect the responses, and infer from those responses the presence of a reverse proxy and the server infrastructure behind it. RevProbe acts as a client that interacts with the Web service over the network.

There exist two general approaches to identify proxies through active probing: *timing-based* and *discrepancy-based*. Timing-based techniques leverage the property that every reverse proxy introduces an additional hop in the communication, which in turn introduces unnatural additional latency from the client's perspective [111]. Discrepancy-based techniques focus on identifying discrepancies that the proxies may introduce in the traffic [106].

We use (mostly) a discrepancy-based approach because timing-based approaches cannot identify multiple servers hiding behind the reverse proxy, e.g., a WLB. They also have problems detecting reverse proxies when the delay they introduce is too small to be detectable across the Internet, e.g., when the reverse proxy sits in the same subnet as the final server.

In contrast, discrepancy-based techniques can identify multiple servers behind a reverse proxy. However, in some situations they may miss some servers. For example, in the presence of a WLB that balances traffic across multiple servers, if some of those final servers are configured identically and perfectly synchronized, RevProbe may not be able to distinguish them and underestimate the number of final servers. With discrepancy-based approaches, the more servers hide behind a reverse proxy the easier the strict problem definition gets, since each new server may introduce discrepancies, but the generalized problem definition gets harder as more information needs to be recovered.

It is worth noting that some of our detection module (e.g., Max-Forwards and phpinfo, detailed in Section 6.5) are not based on discrepancies and thus can also

	Reverse Proxy Detection									
	Explicit			Silent			WSF			
Tool	**RP**	**WLB**	**WAF**	**RP**	**WLB**	**WAF**	**Exp.**	**Imp.**	**Classes**	**# Req.**
Our tool	✓	✓	-	✓	✓	-	✓	✓	*	45
Halberd [112]	-	✓	-	-	✓	-	✓	-	*	25,570
Htrosbif [109]	-	✓	-	-	✓	-	✓	✓	75	20
http_trace.nasl [107]	✓	-	-	-	-	-	-	-	-	1
lbmap [108]	✓	✓	✓	✓	✓	✓	✓	✓	6	37
TLHS [110]	✓	-	-	✓	-	-	-	-	-	3
WAFW00f [114]	-	-	✓	-	-	✓	-	-	-	13
ErrorMint [122]	-	-	-	-	-	-	✓	-	*	9
HMAP [120]	-	-	-	-	-	-	✓	✓	28	178
HTTPRecon [117]	-	-	-	-	-	-	✓	✓	460	9
HTTPrint [115]	-	-	-	-	-	-	✓	✓	118	22
http_version.nasl [216]	-	-	-	-	-	-	✓	-	*	2
nikto.nasl [217]	-	-	-	-	-	-	✓	✓	1,250	6,297
Nmap [121]	-	-	-	-	-	-	✓	-	*	1

Table 6.1: Summary of existing tools for reverse proxy detection and Web server fingerprinting (WSF).

identify difficult configurations such as a reverse proxy running the same software as the final server behind.

6.4 State of the Art

We split the state of the art discussion into related work (Section 6.4.1) and existing tools for reverse proxy detection, also part of the related work but better described together (Section 6.4.2).

6.4.1 Related Work

Weaver et al. [106] propose a technique to detect ISP and forward proxies by installing an application in client hosts and have it communicate with a Web server under their control. Discrepancies between the response sent by their server and the response received by the client application indicate the presence of a proxy. Our work focuses instead on detecting reverse proxies that serve as endpoints of a Web service. In this scenario we do not control the server the client connects to.

In his M.Sc. thesis, Weant [111] proposes to detect reverse proxies through timing analysis of TCP and HTTP round trip times. Timing-based approaches focus on the strict problem definition and cannot identify multiple servers hiding behind the reverse proxy. They also fail to detect reverse proxies when the delay they introduce is too small to be detectable across the Internet. Furthermore, Weant evaluates his technique on a single ground truth configuration and does not measure the prevalence of reverse proxies in malicious and benign infrastructures.

Web server fingerprinting. A number of tools exist to fingerprint the program and version run by a remote Web server [115, 116, 117, 118, 119, 120]. Among these, some tools like Nmap [121] or ErrorMint [122] fingerprint the program version exclusively by examining explicit program version information provided by the server, e.g., in the Server header and error pages. Other tools like HMAP [120], HTTPPrint [115], and HTTPRecon [117] use fingerprints that capture differences between how different Web server versions construct their responses. These type of fingerprints do not rely on the program version information explicitly provided by the server.

Vulnerability scanners such as Nessus [78] and OpenVAS [218] detect vulnerabilities in a variety of software, including Web servers. They first fingerprint the software running at a given endpoint and then look up those software versions in vulnerability databases. Both Nessus and OpenVAS run NASL scripts [219] and there exist NASL scripts for Web server fingerprinting such as http_version.nasl [216] and nikto.nasl [217].

A common limitation of all Web server fingerprinting (WSF) tools is that they assume the $\langle ip, port \rangle$ endpoint to be fingerprinted corresponds to a single Web server, which is not true with reverse proxies. When faced with a reverse proxy, they will often recover the software version of the final server behind the reverse proxy, but they can be confused if the reverse proxy manipulates the responses, or acts as a load balancer to servers with different software. One of the conclusions of this work is that reverse proxy detection and Web server fingerprinting are best done together.

Automatic fingerprint generation. There exist approaches to automatically build program version fingerprints [88, 118]. Currently, RevProbe uses manually generated fingerprints and could benefit from such approaches. Recent work builds fingerprints for malicious server programs (e.g., C&C, exploit kits) scanning the Internet to locate servers that run them [208, 87]. These tools cannot distinguish between reverse proxies and final servers in their results and could leverage our approach to this end.

6.4.2 Reverse Proxy Detection Tools

Table 6.1 summarizes existing reverse proxy detection tools. For completeness, it also includes Web server fingerprinting tools described in the previous section. The table only includes publicly available tools, commercial tools are not included. The table is broken into 3 blocks: reverse proxy detection, Web server fingerprinting (WSF), and number of requests sent by default. Tools with suffix *.nasl* are NASL scripts [219] for the Nessus [78] and OpenVAS [218] vulnerability scanners. All tools take as input a target IP address or domain and focus on the strict problem definition. They do not output a reverse proxy tree or recover the

number of servers behind a WLB, but in some cases they may identify a reverse proxy and a final server behind.

The reverse proxy detection block distinguishes between detection of explicit and silent reverse proxies and also between generic reverse proxy detection (RP), Web load balancer detection (WLB) and Web application firewall (WAF) detection.

The WSF block captures if the software information comes exclusively from explicit version information provided by the Web server (e.g., Server header and versions in error pages) or if it is detected without trusting the explicit version information (e.g., using fingerprints). The final column in this block captures the number of classes (i.e., program versions) the tool can identify. An asterisk indicates the tool has no predefined classes, but rather outputs any explicit program version information. The rightmost column captures the average number of requests that the tool sends in default configuration to a target IP.

Next we detail each of the tools, the detection approaches they use, and the comparison with RevProbe.

Halberd. This tool focuses exclusively on detecting Web load balancers. It sends the same request for a period of 15 seconds to the target IP. Differences in some response headers (e.g., E-Tag, Server) indicate a load balancer. On our tests, it sends on average 25,570 requests, the most of all tools. RevProbe incorporates this technique but it also adds a novel second technique, based on extracting time sequences from the HTTP Date header, to detect WLBs and estimate the number of servers behind.

Htrosbif and lbmap. These tools are similar. They send a number (20 and 37 respectively) of abnormal requests to the target trying to force a response from a reverse proxy. They use a database of signatures on the responses to identify some specific reverse proxy software (i.e., HAProxy, Pound, Vanquish). They differ in the database of requests and signatures. Both fail to detect generic Web server software (e.g., Apache, Nginx) running as reverse proxy. RevProbe also incorporates a module for forcing proxy responses, but uses a novel method to extract fine-grained information from the error pages that also works with Apache and Nginx.

TLHS. Gregoire [110] presents the HTTP traceroute tool (TLHS), which leverages the Max-Forwards HTTP header that limits the maximum number of times a request is forwarded. RevProbe also incorporates this technique, but it cannot be used in isolation because popular reverse proxy software (e.g., Nginx) ignores the header. In our experiments, it only works well with Apache reverse proxies.

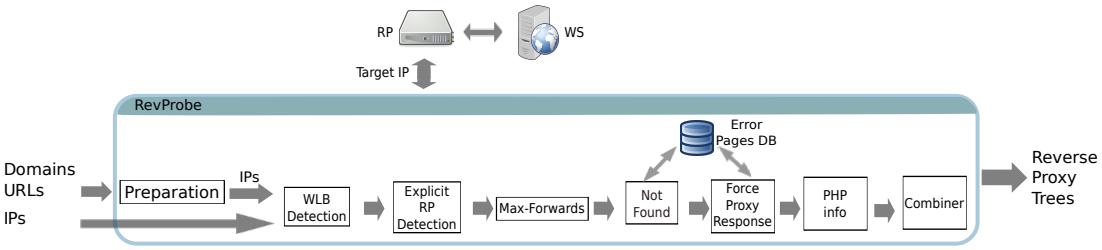


Figure 6.3: Approach overview.

http_trace.nasl. This NASL plugin only detects reverse proxies by examining the HTTP Via header, thus it cannot detect silent reverse proxies. RevProbe also includes an explicit RP detection module for completeness, but its goal is detecting silent reverse proxies.

WAFW00f. This tool exclusively detects Web application firewalls. It sends a normal request and a malicious request to the same URL. Discrepancies between both responses flag a WAF. This tool incorrectly flags any RP as a WAF and produces false negatives if the WAF does not return an error, but rather a 200 OK response with some error message in the body. Currently, RevProbe does not differentiate WAFs from other RPs because their fine-grained classification requires sending attacks, or at least attack-resembling requests to third parties, which can be considered offensive.

In addition to the methods used by the tools above, RevProbe also implements two novel detection techniques, and provides another module that implements a previously known technique not implemented by these tools (i.e., phpinfo [220]).

6.5 Approach

Figure 6.3 provides an overview of our approach. The preparation module first resolves domains and URLs into final IPs. For each IP address, RevProbe examines whether it corresponds to a reverse proxy by sending probes and examining the responses. If it finds a reverse proxy, it outputs a reverse proxy tree for that IP address.

For each IP address, RevProbe runs a number of detection modules. Each module may send requests to the remote IP or simply examine the responses to requests sent by other modules. Each module outputs a, possibly partial, reverse proxy tree. Those trees are combined at the end into a reverse proxy tree for each target IP address. The remainder of this section details the modules in Figure 6.3.

6.5.1 Preparation

A user may want to run RevProbe on a DNS domain or URL, rather than an IP address. The preparation module obtains a set of IP addresses from those domains and URLs and feeds them to RevProbe. Then RevProbe examines each IP address independently.

For domains, the preparation module resolves the domain and extracts the list of IP addresses it points to. For URLs, the preparation module fetches the URL and follows all HTTP redirections the URL may produce. From the final URL in the redirection chain, it extracts the domain name and resolves it as above.

After the preparation step, any HTTP connection from RevProbe uses the IP address to connect (i.e., without resolving the domain again) and provides the domain name if known in the Host header, otherwise the IP address.

6.5.2 Web Load Balancer Detection

The goal of the WLB detector is to detect the presence of a WLB and to provide a *lower bound* in the number of servers that hide behind it. The WLB detector uses three different tests: *same request*, *datetime sequences*, and *load balancer cookie*. These tests are detailed next.

Same request test. The first test is to send the same request multiple times to a given target IP address, monitoring for changes in the responses (e.g., different HTTP Server header values), which may indicate the existence of multiple Web servers and thus the presence of a WLB that forwards requests across them. The challenge in building this test is that not all changes in the responses indicate the presence of a WLB as some parts of the response (e.g., some headers, content) are volatile and change regardless of the request being kept constant, without this indicating the presence of a WLB. Thus, the test checks for differences in the responses related to a server's configuration. As a consequence, it will fail to detect a WLB, or provide only a lower bound on the real number of servers, if the servers behind the WLB are configured identically, e.g., run the same Web server software version, have the same configuration, and deliver the same content. In practice, running a perfectly homogeneous server infrastructure and updating the servers' content simultaneously is challenging and such differences in configuration manifest often.

The tests sends sequentially the same request n times to the target IP and collects the sequence of responses $\{r_1, \ldots, r_n\}$. It also records the time each request and response was received. The larger n the more confidence in the test results, but the noisier the test is. By default, it sends 30 requests.

For each response, it first extracts the features in Table 6.2. The first feature is the hash of the sequence of header names (without value) in the order they appear in the response. All other features correspond to the value of an

Feature	Type
Header name hash	string
Accept-Ranges	string
Allow	string
Connection	string
Content-Disposition	string
Content-Encoding	string
Content-Language	string
Content-Type	string
Date	datetime
P3P	string
Server	string
Status-Line	string
Transfer-Encoding	string
Upgrade	string
X-Powered-By	string

Table 6.2: WLB detection module features.

HTTP header. These features have been selected because their content is related to the server's configuration. The features do not include headers that are not standard [215] (except the popular X-Powered-By), volatile headers that change often (e.g., Set-Cookie), content-related headers (e.g., Content-Length, Etag, Last-Modified) since the content can be dynamic, caching-related headers (e.g., Cache-Control), and proxy-related headers (e.g., Via) that are examined in the explicit detection module (Section 6.5.3). These features can be of 2 types: string or datetime. This test focuses on the string features, the Date header is handled by the second test.

The intuition behind the string features is that their value should be deterministic for the same request and same server. Thus, observing different values in responses to the same request indicates the presence of multiple servers, and thus a WLB. This test outputs the maximum number of distinct values for a feature c. If $c = 1$, no WLB was observed for this test. If $c > 1$, there is a WLB with c servers behind it. Note that if the configuration of a server is changed during the test RevProbe will count the server as two differently-configured servers. The probability of this event can be minimized by reducing the time between requests, at the expense of increasing the load of the target IP, or by repeating the test at a later time.

Datetime sequences test. The second test proposes a novel technique to extract time sequences from the HTTP Date header of the received responses. It does not require to send requests, but examines the responses to the same request test. The motivation behind this test is that timestamps are a good

Algorithm 2 Datetime sequence identification

Input: response sequence, R = $[r_1, \ldots, r_n]$
Output: num_servers $\in [1, \infty)$

```
1:  procedure DATETIME–SEQUENCE
2:      S = {}
3:      for i ∈ [1, n] do
4:          currTime = r_i.getDatetime()
5:          Found = False
6:          for s ∈ S do
7:              th = updateThreshold(s, currTime)
8:              if 0 ≤ (currTime − s.lastTime()) ≤ th then
9:                  s.append(currTime)
10:                 found = True
11:                 break
12:             end if
13:         end for
14:         if Found = False then
15:             S.newSeq(currTime)
16:         end if
17:     end for
18: end procedure
```

source of information for identifying servers behind a WLB whose clocks are not synchronized. Multiple interleaved time sequences manifest the presence of a WLB and the number of servers behind can be approximated by the number of distinct interleaved sequences observed.

Algorithm 2 describes the datetime sequence identification algorithm. It iterates over the sequence of responses (line 3). For each response, it first obtains the datetime in the Date header (line 4). Then, it checks if the datetime is past, but within a threshold th of the end of any existing sequence (lines 6–12). If so, it adds the response to the current sequence (line 8). Otherwise, it creates a new sequence for the response (lines 13–14). Note that if the date in the current request is in the past of the examined sequence, it cannot belong to that sequence as the Date header should monotonically increase for the same server. The threshold value th is dynamically updated (line 7) based on the time difference between this request and the last entry in the sequence that updated the Date header. The new threshold value is $th = \lceil (\sum_i^j RTT_i) * c \rceil$ where i is the index of the last response in the sequence that updated the Date header value, j the index of the current request, RTT_i the round trip time of request-response pair i, and c a constant factor (by default 2).

Load balancer cookie test. The third test leverages that some Web load balancer software introduce cookies in the responses sent by the server for persistence, i.e., to make sure that different HTTP connections of the same Web session are forwarded by the WLB to the same server. The Set-Cookie header from each response obtained from the prior test is checked against the subset of the OWASP cookie database [221]. The OWASP database contains signatures for the cookies produced by different Web applications, including 5 signatures for cookies introduced by commercial load balancers such as F5 BIG-IP [222] and KempLB [223]. This test outputs 1 if no WLB is detected and 2 if a WLB is detected indicating that at least two final servers are identified behind a WLB.

The WLB detection module outputs a tree with a root WLB node and in layer 1 the maximum of the number of servers found by the same request, date sequences, and load balancer cookie tests. If no WLB is found, it outputs a singleton tree with a root WS node.

6.5.3 Explicit Reverse Proxy Detection

While detecting silent reverse proxies is its main goal, RevProbe also detects explicit reverse proxies, which announce their presence. The explicit RP module does not produce traffic, but examines a number of headers in all responses received from a target IP.

The Via header must be used by proxies to indicate their presence[1] [215]. Each proxy that forwards an HTTP message (i.e., request or response) must append a comma-separated entry to the Via header specifying the protocol version of the message received and the proxy's hostname (or pseudonym). Some explicit RPs use the X-Via [224] header instead. The explicit RP module parses the Via and X-Via headers if they exist to retrieve the list of explicit proxies. In practice, many reverse proxies are silent; they do not append themselves to the Via header and strip these headers. This module also examines the following cache-related headers, useful for detecting caching RPs: X-Served-By [225], X-Cache [225], X-Cache-Lookup [226], X-Varnish [227], and X-Cache-Hits [225].

This module outputs a tree with a node for each explicit proxy identified, or a singleton WS node if no RP is found.

6.5.4 Max-Forwards

The Max-Forwards header in an HTTP request can be used to limit the number of proxies that can forward the request [215]. It is set by the source to a maximum number of hops. Each proxy must check its value: if the value is zero the proxy must not forward the request, but must respond as the final recipient; if greater

[1]The X-Forwarded-For header is analogous but only included in requests to track the client's IP address.

than zero it must decrease it and forward the request with the updated Max-Forwards value. Max-Forwards is only required to be supported with the TRACE and OPTIONS methods, but may be supported with other methods such as GET.

This module sends HTTP requests to the target IP each time increasing the value in the Max-Forwards header, from zero to a maximum number of hops (by default 3). We compare each response (starting with value 1) with the prior response. If the two responses have identical values in the string headers in Table 6.2, no reverse proxy is found and the test exits. If there is a difference, then a reverse proxy is found, e.g., value zero returns a 400 proxy error and value one returns 200 OK. In this case, the Max-Forward value is incremented and the test repeated to check if there may be multiple reverse proxies chained. A limitation of this technique is that some Web servers such as Nginx [213], HAProxy [210], and Pound [211], do not support the Max-Forwards header and always forward the request. Our test uses the GET method as we have experimentally observed that TRACE is often not supported.

This module outputs a tree with a node for each RP identified, or a singleton WS node if no RP is found.

6.5.5 Error Pages Database

This section describes the error pages database, which is an auxiliary module used by other detection modules, rather than a detection module itself.

The HTML error page in an HTTP error response may explicitly leak information about the server version and even the server hostname (or IP address). For example, some default pages for Apache report the Web server version in the <*address*> tag. Furthermore, if the server uses the default error page from the Web server software (rather than a customized error page), the structure of the error page implicitly leaks the server's software.

We have built a database of 51 default error pages for popular Web servers. Each entry in the database is indexed by the hash of the sequence of HTML tag and attribute names in the error page. Each entry contains information about the server software the error page corresponds to, and whether some tag in the HTML content stores version or endpoint information.

Every time an HTTP error response is received by any module, RevProbe hashes the error page structure and looks up the hash in the database. If found, it tags the error page with the server software and extracts the explicit version and endpoint information, if any.

6.5.6 Not Found Module

This module sends a request for a non-existing resource, which triggers a not found error. The request will typically be forwarded by the RP and answered by

the final server since only the server knows what content exists. Caching RPs will not find the content in the cache and forward the request as well.

This module identifies a reverse proxy using two methods. The first method uses the error page database to extract (explicit or implicit) software information about the server that returns the error page. Then, it compares this information with the Server header in the HTTP response. If it finds a discrepancy it flags a reverse proxy. For example, if the error page contains an <*address*> tag with an Apache version, while the Server header corresponds to Nginx, this indicates the presence of an Nginx RP in front of an Apache server. This method works because some Web servers like Nginx overwrite the Server header of a response coming from the final server, even if the HTTP specification mandates they should not [215]. The second method checks if the returned error page contains an explicit hostname (or public IP address), which does not correspond to the target IP. If so, it also flags a reverse proxy and the hostname (or IP address) in the error page identifies the final server.

This module outputs a tree with a root RP node and one WS node at layer 1 if a RP is found, otherwise a singleton WS node.

6.5.7 Force Proxy Response

A perfectly silent reverse proxy would forward all requests to the final server(s). In reality, RP implementations will often parse the requests and do some basic checks on them. Based on those checks they will forward the request, or reply to it themselves with an error. When an RP is present, most requests will be answered by the final server, but incorrect requests may be answered by the reverse proxy itself.

This module sends a number of requests to the target IP. Some of the requests are normal and the others are formatted to trigger an error from popular Web servers used as RP. Then, it uses two methods to check for discrepancies between the responses to both types of requests.

The first method uses a fingerprint on the response to an incorrect request. In some cases that response is so specific to a particular Web server that it can be used as a fingerprint for that Web server. RevProbe has such fingerprints for 3 programs that can only act as RP but not as WS: HAProxy [210], Pound [211], and Varnish [227].

The second method looks for Web server software discrepancies. It compares the Web server software information extracted from the response to a proper request, with the same information extracted from the error response to an incorrect request. For the latter, it leverages the error page database. If it finds a discrepancy it flags a reverse proxy. This method is similar to the first method of the not found test in Section 6.5.5. The difference is that here it operates on software discrepancies found across multiple responses, while in Section 6.5.5 it focuses on discrepancies within the same response (Server header and HTML

content).

This module outputs a tree with a root RP node and one WS node at layer 1 if a RP is found, otherwise a singleton WS node.

6.5.8 PHPinfo

Web servers that support PHP may have the phpinfo.php file, which administrators may have forgotten to remove. This file executes the phpinfo function [220], that returns a wealth of data about the server, and pipes this data over the network to the client. The returned data may include the server's IP address in the SERVER_ADDR field. For each target IP, RevProbe tries to fetch this file from 4 common paths. If it finds it, then it checks if the SERVER_ADDR field contains a public IP address that differs from the target IP. If so, this reveals the presence of a reverse proxy, and also deanonymizes the server behind it.

This module outputs a tree with a root RP node and one WS node at layer 1 if a RP is found, otherwise a singleton WS node.

6.5.9 Combiner

The combiner takes as input the possibly partial reverse proxy trees produced by each of the detection modules and merges them to produce the final tree for each target IP address. In addition to merging the tree nodes, it annotates the nodes with the software package, version, and IP information that may have been recovered by the different tests.

The output of RevProbe is a reverse proxy tree for each target IP that has been flagged as a RP, as well as the list of target IPs for which no RP has been found.

6.6 Evaluation

This section evaluates our approach. First, we compare the accuracy of RevProbe with other prior RP detection tools using 36 silent proxy configurations for which we have ground truth (Section 6.6.1). Then, we test RevProbe on live websites, namely on the top 10,000 Alexa domains and on 8,512 malicious domains (Section 6.6.2).

6.6.1 Tool Comparison

To compare the accuracy of RevProbe to other tools, we test them on 36 silent reverse proxy configurations for which we have ground truth. We test 3 different reverse proxy trees depicted in Figure 6.4. Type 1 corresponds to a reverse proxy with one server behind. Type 2 is a WLB that distributes connections to 2 servers.

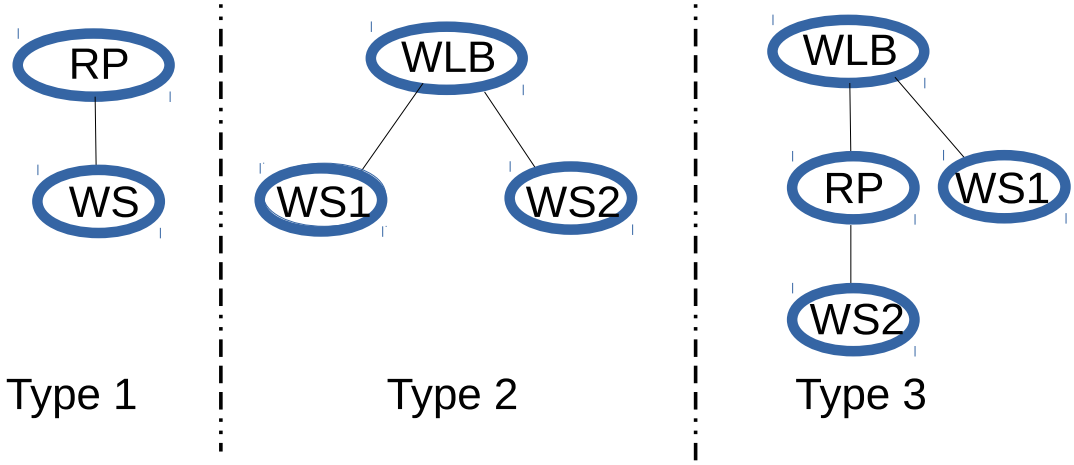


Figure 6.4: Server trees used for tool comparison.

Program	RP	WS	Versions
Apache	✓	✓	2.2.22, 2.4.7
Nginx	✓	✓	1.1.19, 1.6.2
IIS	-	✓	7.5
HAProxy	✓	-	1.4.24
Pound	✓	-	2.6

Table 6.3: Programs used in tool comparison.

Type 3 is a WLB balancing between one final server and a reverse proxy that hides another final server. For each tree, we test multiple software configurations. We use 7 versions of 5 programs, summarized in Table 6.3. Apache and Nginx can operate as final server or reverse proxy, IIS only as final server, and HAProxy and Pound only as reverse proxies. We configure all RPs and WLBs silently. The WLBs use a round-robin policy.

Table 6.4 summarizes the results. For each of the 36 configurations, it shows the type of tree tested, the server versions used at each layer, and the test results for each tool. For configurations of Type 3 the server used as reverse proxy in layer 1 is marked with an asterisk. For each tool test result, a ✓symbol means that the tool perfectly recovered the shape of the reverse proxy tree; *wlb* that it detected a WLB at the root node but not the servers behind it; *rp* that it detected a RP at the root but not the servers behind it; and a - symbol that it did not detect a reverse proxy.

RevProbe results. In all 24 configurations of type 1 and 2 RevProbe perfectly recovers the reverse proxy tree. In addition, it also recovers the software package and version of all servers in those trees. Trees of type 3 have mixed results. In 4 out of 12 configurations for trees of type 3 it recovers the perfect

Tree	Root		Layer 1		Layer 2	our tool	Halberd	Htrosbif	http_trace.nasl	lbmap	TLHS	WAFW00f
T1	Nginx	1.6.2	Apache/2.4.7			✓	-	-	-	-	-	-
T1	Nginx	1.6.2	Nginx/1.1.19			✓	-	-	-	-	-	-
T1	Nginx	1.6.2	IIS/7.5			✓	-	-	-	-	-	-
T1	Apache	2.4.7	Apache/2.2.22			✓	-	-	-	-	rp	rp
T1	Apache	2.4.7	Nginx/1.6.2			✓	-	-	-	-	rp	rp
T1	Apache	2.4.7	IIS/7.5			✓	-	-	-	-	rp	rp
T1	HAProxy	1.4.24	Apache/2.4.7			✓	-	✓	-	✓	-	-
T1	HAProxy	1.4.24	Nginx/1.6.2			✓	-	✓	-	✓	-	-
T1	HAProxy	1.4.24	IIS/7.5			✓	-	rp	-	✓	-	rp
T1	Pound	2.6	Apache/2.4.7			✓	-	✓	-	✓	-	-
T1	Pound	2.6	Nginx/1.6.2			✓	-	✓	-	✓	-	-
T1	Pound	2.6	IIS/7.5			✓	-	rp	-	✓	-	rp
T2	Nginx	1.6.2	Apache/2.4.7	Apache/2.2.22		✓	wlb	-	-	-	-	-
T2	Nginx	1.6.2	Nginx/1.1.19	Nginx/1.6.2		✓	wlb	-	-	-	-	-
T2	Nginx	1.6.2	IIS/7.5	Apache/2.4.7		✓	wlb	-	-	-	-	-
T2	Apache	2.4.7	Apache/2.4.7	Apache/2.2.22		✓	wlb	-	-	-	rp	rp
T2	Apache	2.4.7	Nginx/1.1.19	Nginx/1.6.2		✓	wlb	-	-	-	rp	rp
T2	Apache	2.4.7	IIS/7.5	Apache/2.4.7		✓	wlb	-	-	-	rp	rp
T2	HAProxy	1.4.24	Apache/2.4.7	Apache/2.2.22		✓	wlb	rp	-	✓	rp	rp
T2	HAProxy	1.4.24	Nginx/1.1.19	Nginx/1.6.2		✓	wlb	rp	-	✓	rp	rp
T2	HAProxy	1.4.24	IIS/7.5	Apache/2.4.7		✓	wlb	rp	-	✓	rp	rp
T2	Pound	2.6	Apache/2.4.7	Apache/2.2.22		✓	wlb	rp	-	✓	rp	rp
T2	Pound	2.6	Nginx/1.1.19	Nginx/1.6.2		✓	wlb	rp	-	✓	rp	rp
T2	Pound	2.6	IIS/7.5	Apache/2.4.7		✓	wlb	rp	-	✓	rp	rp
T3	Nginx	1.6.2	*Apache/2.4.7	IIS/7.5	IIS/7.5	wlb	wlb	-	-	-	-	-
T3	Nginx	1.6.2	*Apache/2.4.7	Apache/2.4.7	Nginx/1.6.2	✓	wlb	-	-	-	rp	-
T3	Nginx	1.6.2	*Nginx/1.6.2	Apache/2.4.7	Apache/2.4.7	✓	wlb	-	-	-	-	-
T3	Apache	2.4.7	*Apache/2.4.7	Apache/2.2.22	Nginx/1.6.2	✓	wlb	-	-	-	rp	rp
T3	Apache	2.4.7	*Apache/2.4.7	IIS/7.5	Nginx/1.6.2	wlb	wlb	-	-	-	rp	rp
T3	Apache	2.4.7	*Apache/2.4.7	Apache/2.4.7	Nginx/1.6.2	✓	wlb	-	-	-	rp	rp
T3	HAProxy	1.4.24	*Apache/2.4.7	Apache/2.4.7	Nginx/1.6.2	wlb	wlb	rp	-	wlb	rp	rp
T3	HAProxy	1.4.24	*Nginx/1.6.2	Nginx/1.6.2	Apache/2.4.7	wlb	wlb	rp	-	wlb	rp	-
T3	HAProxy	1.4.24	*Nginx/1.6.2	IIS/7.5	IIS/7.5	wlb	wlb	rp	-	wlb	-	-
T3	Pound	2.6	*Nginx/1.6.2	IIS/7.5	IIS/7.5	wlb	wlb	rp	-	wlb	rp	rp
T3	Pound	2.6	*Nginx/1.6.2	Nginx/1.6.2	Apache/2.4.7	wlb	wlb	rp	-	wlb	rp	-
T3	Pound	2.6	*Nginx/1.6.2	Apache/2.4.7	Apache/2.4.7	wlb	wlb	rp	-	wlb	rp	rp

Table 6.4: Tool comparison. For each tool test, a ✓symbol means the reverse proxy tree was perfectly recovered; *wlb* that a Web load balancer was detected but not the rest of the tree; *rp* that a reverse proxy was detected but not the rest of the tree; and a - symbol that no reverse proxy was detected. An asterisk before a server at layer 1 means that server was used as a reverse proxy.

	Domains		IPs		Root			
Source	**All**	**Active**	**All**	**Active**	**RP**	**Explicit**	**Silent**	**WLB**
Alexa	10,000	9,626	13,067	11,879	2,361 (19.9%)	1,066 (45.1%)	1,295(54.9%)	2,045 (86.6%)
MD	8,512	6,172	4,097	3,528	578 (16.4%)	44 (7.6%)	534 (92.4%)	479 (82.9%)

Table 6.5: our tool Results on benign and malicious domains.

reverse proxy tree. For the remaining 8 type 3 configurations it identifies that the target IP address corresponds to a WLB (strict problem definition) but it is not able to recover the internal tree structure. One issue with trees of Type 3 is that only the Max-Forwards and Explicit RP detection modules can recover multiple RP layers. And, some programs ignore the Max-Forwards header completely, so silent RPs at intermediate layers are challenging to recover.

Other tools. Among the other tools, Halberd identifies a WLB in the 24 T2 and T3 trees. However, it fails to identify a RP in all 8 T1 trees because there is only a Web server and similar to our WLB detection module it requires more than one WS to identify a RP. Htrosbif fails to completely detect Apache and Nginx as RP, it only detects HAProxy and Pound for which it has fingerprints. It does not detect any WLB and fully recovers 4 T1 trees. http_trace.nasl fails all tests because it only detects explicit reverse proxies and our configurations only use silent reverse proxies. lbmap also fails to detect Apache and Nginx as a reverse proxy in all configurations. It perfectly recovers the reverse proxy tree for T1 and T2 trees with HAProxy and Pound at the root, and the presence of a WLB in T3 trees with HAProxy and Pound at the root. TLHS uses the Max-Forwards header for detection, which works well with Apache acting as RP. Surprisingly, it also detects a RP in some T2 and T3 configurations where Apache is not used as RP/WLB (or even used at all). This happens because the error page titles are different. WAFW00f focuses on detecting WAFs. None of our configurations has a WAF installed, so these detections could be considered false positives. WAFW00f uses discrepancies to identify WAFs but that technique it uses identifies any RP as a WAF.

Summary. The tool comparison shows that RevProbe outperforms prior RP detection tools. RevProbe perfectly recovers the reverse proxy tree in 77% of the configurations, and the presence of a reverse proxy (strict problem definition) in the remaining 23%. None of the prior tools is a close competitor or a clear winner over the others. lbmap perfectly recovers 12 trees and a WLB in other 6 configurations, but it only detects HAProxy and Pound reverse proxies. Htrosbif perfectly recovers 4 T1 trees and a RP in 14 other configurations, but it has no support for WLBs. And, Halberd only detect WLBs.

Source	RPs	Type 1	Type 2	Type 3	Oth.
Alexa	2,361	311 (13%)	2,045 (86%)	3 (1%)	2
MD	578	98 (17%)	450 (78%)	29 (5%)	1

Table 6.6: Reverse proxy tree types identified in benign and malicious domains.

6.6.2 Live Websites

We test RevProbe on both benign and malicious live websites to compare how they use reverse proxies. As representative of likely benign websites we use the Alexa top 10,000 domains [186]. For malicious websites we use the immortal domains list from malwaredomains.com [214] (MD). Immortal domains are domains alive for more than 90 and less than 360 days and that Google Safebrowsing [37] still reports as malicious. In all cases, RevProbe requests the root page of the website. For malicious websites RevProbe uses a virtual private network (VPN) for anonymity. The VPN has a large pool of exit IP addresses, making it difficult to identify RevProbe's probing.

Table 6.5 summarizes the results. For each dataset, it first shows the number of domains tested and those that resolved to at least one IP address. Then, the number of distinct IPs that those live domains resolved to (possibly a larger number) and the number of those IP addresses that were active, i.e., responded to at least one probe. Next, it shows the number of active IP addresses where RevProbe detected a reverse proxy, how many of those root reverse proxies were explicit and silent, and how many were load balancers.

Overall, RevProbe tests 11,879 distinct active IP addresses from Alexa and 3,528 from MD. Among the active IP addresses, RevProbe identifies 2,361 reverse proxies in Alexa and 578 in MD. Thus, 16% of malicious and 20% of benign active IPs correspond to a reverse proxy. Of the malicious reverse proxies, 92% are silent, compared to 55% for benign RPs. This shows how the vast majority of malicious reverse proxies are silent and used to hide the servers behind them. The fraction is significantly smaller among benign RPs, but still more than half of benign reverse proxies are silent. This may be due to benign services also wanting to hide their server infrastructure and to popular Web server software (e.g., Apache, Nginx) to be silent by default when running as RP.

Of the malicious RPs, 83% are load balancers, similar to the 87% for benign RPs. Thus, the vast majority of RPs in both malicious and benign infrastructures are used to distribute traffic among multiple servers. Figure 6.5 shows the distribution of the number of servers that RevProbe identifies behind WLBs. Of the malicious WLBs 89% have two servers behind, 7% have three servers, and 3% have more than 3. The maximum number of servers behind a WLB is 6. For benign WLBs those percentages are 75% (2), 11% (3), 14% (>3), and the maximum 30. As expected, the WLBs of highly ranked benign Websites distribute their traffic among a larger number of servers than WLBs in malicious infrastructures.

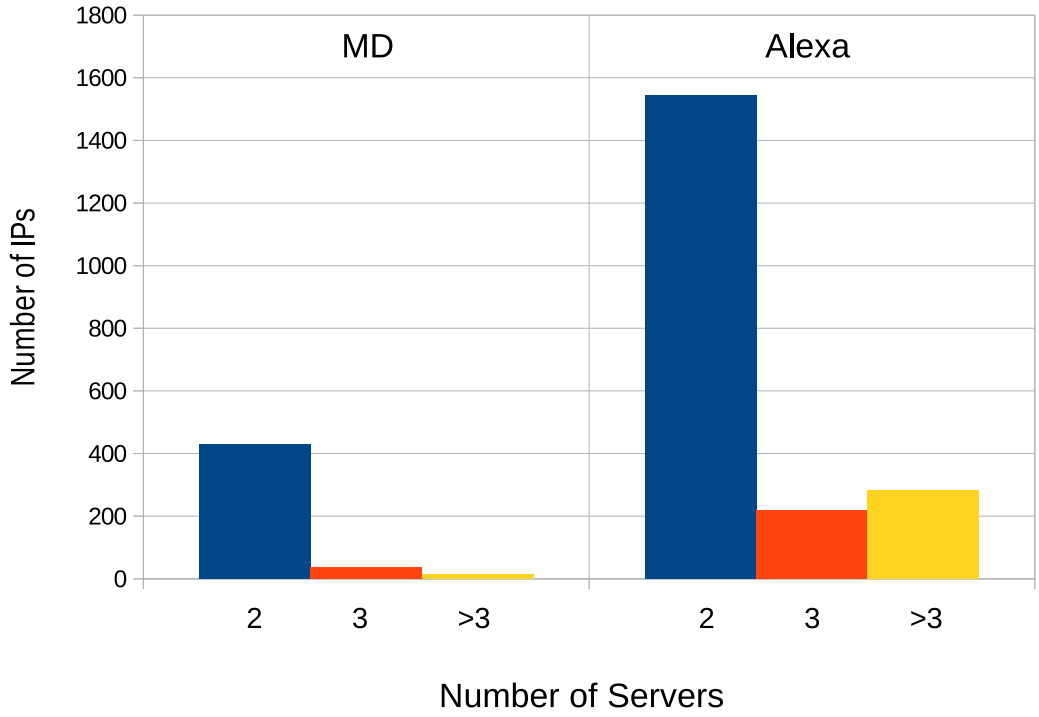


Figure 6.5: Number of final servers found behind WLBs.

Tree types. Table 6.6 summarizes the type of reverse proxy trees RevProbe identifies behind the active IP addresses. The results are quite similar for Alexa and MD domains. The most common tree is a WLB with a variable number of servers behind (Type 2 in Figure 6.4 with varying number of servers in layer 1) detected for 78% of the malicious and 86% of the benign active IPs. The Type 1 tree in Figure 6.4 (one reverse proxy with one server) is detected for 17% of the malicious and 13% of the benign active IPs. The Type 3 tree in Figure 6.4 (one WLB and a RP at layer 1) occurs in 1% of malicious and 5% of benign active IPs. The remaining 3 trees have a root RP, a WLB in layer 1, and 2 servers in layer 2. These results point to most RPs being the root of simple hierarchies, predominantly T2 and T1 trees.

Web server software. Table 6.7 summarizes the Web server software that RevProbe identifies in the nodes of the recovered trees. Overall, RevProbe recovers the Web server type for 41% of the nodes in malicious trees and 39% of nodes in benign trees. The most common Web server software in malicious infrastructures is Squid found on 40% of the nodes tagged with Web server software, followed by Apache (20%), and Nginx (12%). In the Alexa domains, Varnish (27%) is most common, followed by Squid (21%), and Nginx (20%).

The vast majority of servers use open source software. The most popular

Program	Alexa	MD
Apache	358 (5%)	133 (8%)
IIS	51 (1%)	31 (2%)
Nginx	513 (8%)	78 (5%)
Squid	540 (8%)	269 (17%)
Varnish	699 (10%)	55 (3%)
Others	452 (7%)	98 (6%)
Null	4,090 (61%)	958 (59%)
Total	6,703	1,622

Table 6.7: Software identified running on the nodes in the reverse proxy trees recovered by our tool.

reverse proxy software correspond to caching proxies (Squid and Varnish). This is expected in benign infrastructures where performance is an important reason of using reverse proxies, but surprising in malicious infrastructures where the RP could cache sensitive information. It could happen that different types of malicious domains behave differently in this regard, e.g., that phishing domains are cached but exploit kit data is not.

Deanonymizations. The phpinfo module recovers the public IP address of a final Web server of a malicious domain. It also recovers a private IP address for 12 malicious servers. For the benign domains, it recovers the public IP address for 29 final servers and a private IP address for another 41. To confirm the deanonymizations we connect to the 30 public IP addresses and fetch the root page using the corresponding domain in the HTTP Host header. If the content served is similar to the content served through the RP, the deanonymization is confirmed. Of the 30 IPs, 13 are confirmed, 4 did not respond, 10 show an error page, and 3 show different content. Overall, RevProbe deanonymizes 1 malicious server and 12 Alexa servers hiding behind silent reverse proxies.

Summary. Our results show that reverse proxies are common in malicious Web infrastructures (16% of active IPs). Those reverse proxies are predominantly silent to hide the existence of servers behind (92%). Reverse proxies are also common in benign infrastructures (20% of active IPs) but are less often silent (55%). In both malicious and benign infrastructures RPs are predominantly used to load balance traffic among multiple servers (83%–87%). The vast majority of RPs are root to simple server hierarchies, predominantly Type 2 and Type 1 trees (95%–99%).

6.7 Discussion

Ethical considerations. Active probing approaches send traffic to targets, which may correspond to third-party services that have not solicited it. Thus, some targets may consider the probes undesirable or even offensive. In addition, the probes can potentially place a burden on the target. We take this issue seriously and seek a careful balance between the amount of interaction and the merits of the results. Since Web services are remote and only available as a black box, we believe active probing approaches are required to detect reverse proxies and the server infrastructure behind them.

We have designed RevProbe to send a small number of requests (45 by default) to a remote target, which we believe to be a manageable load even for small websites. The requests sent by RevProbe are all well-formed HTTP requests. Only the not found and force proxy response modules send requests designed to trigger an error. For our experiments with third-party Web services we limit the force proxy error module to use a single type of incorrect request, an incorrect HTTP method such as "GOT / HTTP/1.1" rather than "GET / HTTP/1.1".

Shared servers. RevProbe identifies the server infrastructure hiding behind an IP address. In some scenarios multiple reverse proxies, at different IP addresses, may proxy to the same final servers. This can happen, among others, with domains that resolve to multiple IP addresses and with Web services pointed by multiple domains. Currently, RevProbe only detects whether servers identified behind different reverse proxies (i.e., target IPs) are the same if it recovers their public IP address. We leave as future work exploring other avenues to combine reverse proxy trees from different target IP addresses.

Incomplete trees. Our evaluation on controlled silent reverse proxy configurations shows that Type 3 configurations with 2 layers of proxy servers are challenging to fully recover. Only two of our modules specifically recover sequences of reverse proxies. As next step, we plan to explore other techniques that may be able to detect sequences of reverse proxies, including timing-based approaches. In general, RevProbe cannot always recover a perfect reverse proxy tree, but it is a significant step forward from prior tools that do not specifically address the generalized problem definition.

Combining fingerprinting. Current Web server fingerprinting tools have problems with reverse proxies as they assume an IP address corresponds to a single server. One conclusion of this work is that Web server fingerprinting and reverse proxy detection are best combined together. RevProbe takes a step towards this goal, being able to recover software package information for 40% of all servers.

However, we have not yet built a large database of program version fingerprints. We plan to further explore this combination next.

Other protocols. In this work we have focused on HTTP communication, but our techniques should be equally applicable to HTTPS. Active probing approaches that look at discrepancies in the traffic are also applicable to other protocols, but require the protocol grammar. For proprietary protocols (e.g., C&C) the protocol grammar can be recovered by analyzing the network traffic [228] or program executions [201].

Evasion. A malicious Web service can change its response based on parameters of the client such as geographical location, User-Agent, and Referer [229, 230]. To address such cloaking, RevProbe uses a VPN when connecting to malicious Web services, and can be configured to change its HTTP parameters including User-Agent and Referer. To avoid detection, attackers may attempt to remove discrepancies introduced by the reverse proxy. However, complete removal requires deep understanding of the reverse proxy code and configurable options, as well as careful configuration of the final servers. The difficulty to get all of these perfectly right is a key component of RevProbe's detection.

6.8 Conclusion

In this paper we have presented RevProbe, a state-of-the-art tool for automatically detecting silent reverse proxies and identifying the server infrastructure behind them. RevProbe uses an active probing approach that sends requests to a remote target IP address and analyzes the responses for indications that the IP address corresponds to a reverse proxy with other servers behind. When it detects a reverse proxy, it outputs a *reverse proxy tree* capturing the hierarchy of servers it detected behind the reverse proxy. When possible, the identified servers are tagged with their software package, version, and IP address.

We have compared RevProbe with existing tools on 36 silent reverse proxy configurations showing that RevProbe significantly outperforms them. We have also applied RevProbe to perform the first study on the usage of silent reverse proxies in both benign and malicious Web services. RevProbe identifies that 16% of malicious and 20% benign active IP addresses correspond to reverse proxies, that 92% of those are silent compared to 55% for benign reverse proxies, and that reverse proxies in both malicious and benign server infrastructures are predominantly used to load balance connections across multiple servers.

Part III

Conclusion

7

Conclusion and Future Directions

In the last years cybercrime has become more relevant in the wild because its revenues have become significant, due to this fact new techniques and analysis methodologies are needed to understand this phenomenon. In this thesis we proposed a number of significant advances for understanding cybercrime. We have tackled the problem from two complementary points of view: the clients that get infected and are used to perpetrate malicious actions that lead to illict gains and the servers that are used to control and manage the infrastructure of infected machines. Our techniques and methodologies help to understand cybercrime has a whole, giving insights for improving a fundamental step of the software development process, that is patching, and also support fast and precise take-down efforts of malicious servers.

In the first part of this thesis, we have developed a new technique to associate and analyze data from different dataset for investigating the lifecycle of client-side vulnerabilities. We have analyzed field data passively collected on 8.4 million hosts over an observation period of 5 years, examining a total of 1,593 vulnerabilities from 10 client-side applications.

Furthermore, we have discovered two novel attacks made possible by the fact that different programs or different versions of the same program may ship the same copy of a piece of code (e.g., a .dll file on windows) that is vulnerable. In this scenario the two applications have two different patching programs and policies. Hence an attacker can leverage the window of time that opens between the patching of the two applications to exploit the application that is not patched yet.

We have also performed a statistical analysis of the vulnerability lifecycle. For most vulnerabilities patching starts around the disclosure date, but the patching mechanism has an important impact on the rate of patch deployment, auto-updated software generally patches 1.5 times faster than the manually updated software. However, only 28% of the patches in our study reach 95% of the vulnerable hosts during our observation period. We also found that the median fraction

of vulnerable hosts patched when exploits are released is at most 14%.

This suggests that there are additional factors that influence the patch deployment process, in particular user education and knowledge play a key role in this process. We divide the users into different categories (professionals, developers and security analysts) based on the applications they have installed on their systems. Our analysis shows that developers and security analysts patch faster than other users. Our findings will help system administrators and security analysts to asses the risk associated with vulnerabilities. Moreover understanding how vulnerabilities evolve over time can give some insights to cyber-insurance companies that can tailor their policies for example to different user categories.

The analysys of the vulnerability lifecycle has some limitations, for example we had to limit our analysis to 10 application and to a manageable number of vulnerabilities. Analyzing more vulnerabilities, more applications, and understand whether software vendors improve their policies and solve issues like shared code would give an updated picture of the software deployment ecosystem. This evolutionary picture will help to develop new countermeasures and security policies to protect against victim infection.

The second part of this thesis presented novel defences against malicious infrastructures, we have investigated more than 500 alive exploits servers over a period of more than one year, we have milked and classified 11,363 unique malware samples from these servers. By using the classification data of the malware and the servers configuration data we have successfully identified different cybercrime operations. Furthermore we have analyzed the abuse report procedure showing that our community needs to improve this procedure because at the moment it does not work as it is supposed to be. With the data that we have collected over more than one year of observations of exploit servers operation we have created the Malicia dataset, that has been released to 64 international institutions worldwide, including companies and universities. This dataset includes the malware that we have collected and classified (11,363 unique samples) and all the metadata associated with the servers that were distributing those malwares.

Nonetheless, malicious infrastructures include differente typologies of servers, not only exploit servers, for this reason we have proposed CyberProbe, a system that is able to fingerprint and detect any kind of malicious server in the wild. We have generated 23 fingerprints and scanned the internet looking for malicious servers. We have identified 151 malicious servers and 7,881 P2P bots through 24 localized and internet-wide scans. Of the identified servers 75% are unknown to existing public databases of malicious servers, CyberProbe achieves 4 times better coverage than existing systems.

However, tools like CyberProbe and its evolution AutoProbe [87] have a fundamental limitation, they cannot understand if the malicious server they are probing through HTTP is a silent reverse proxy that hides the location of the final server. To this end we have developed a new system called RevProbe a state-of-the-art tool for automatically detecting silent reverse proxies and identify the

server hierarchy behind them. Also RevProbe uses an active probing approach and can easily be integrated with CyberProbe and AutoProbe to make their detection more accurate. We have compared RevProbe with existing tools on 36 silent reverse proxy configurations showing that RevProbe outperforms them. With RevProbe we have performed the first measurement of reverse proxy prevalence in the wild. RevProbe identifies that 16% of malicious and 20% of benign active IP addresses correspond to reverse proxies, 92% of the malicious IPs are silent, compared with 55% of benign IPs. We have also observed that reverse proxies are predominatly used as load balancers, to balance connections across multiple servers. Our contributions to identify and enumerate malicious infrastructures will help law-enforcement agencies, governments and researchers to better understand cybercrime operations and to counter them with advanced and accurate tool.

The main goals that we have pursued in the second part of this thesis are attribution (i.e., associate malicious servers to their operation) and enumeration (i.e., enumerate all the servers of an operation). For the first goal (i.e., attribution) a natural evolution would be to investigate which are the indivituals or the organizations behind an operation. With respect to the second goal (i.e., enumeration) a future direction would be to deploy CyberProbe and RevProbe together into a provider network in which a malicious server has already been reported, in this way, by exploiting the *provider locality* property mentioned in Chapter 5 it would be possible to eradicate all the malicious servers of a give family from the provider network.

7.1 Funding Acknowledgments

This research was partially supported by the Regional Government of Madrid through the N-GREENS SoftwareCM project S2013/ICE-2731, and by the Spanish Government through Grant TIN2012-39391-C04-01, a Juan de la Cierva Fellowship for Juan Caballero. This work was supported in part by the European Union through the FP7 network of excellence NESSoS (Grant FP7- ICT No. 256980). Partial support was also provided by the EIT-ICT Labs CADENCE project.

Bibliography

[1] B. Krebs, "Spam nation," in *Spam Nation* (Sourcebooks, ed.), 2014.

[2] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage, "Spamalytics: An empirical analysis of spam marketing conversion," in *ACM Conference on Computer and Communications Security*, (Alexandria, VA), October 2008.

[3] C. Kanich, N. Weaver, D. McCoy, T. Halvorson, C. Kreibich, K. Levchenko, V. Paxson, G. M. Voelker, and S. Savage, "Show me the money: Characterizing spam-advertised revenue.," in *USENIX Security Symposium*, USENIX Association, 2001.

[4] Wikipedia, "Bittorrent https://en.wikipedia.org/wiki/Bittorrent."

[5] K. Onarlioglu, U. O. Yilmaz, D. Balzarotti, and E. Kirda, "Insights into user behavior in dealing with internet attacks," in *19th Annual Network and Distributed System Security Symposium (NDSS)*, NDSS 12, January 2012.

[6] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *International World Wide Web Conference*, 2010.

[7] M. P. Center, "What is ransomware?." https://www.microsoft.com/security/portal/mmpc/shared/ransomware.aspx.

[8] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *IEEE Symposium on Security and Privacy*, (Oakland, CA), May 2008.

[9] wikipedia, "Intrusion detection system." https://en.wikipedia.org/wiki/Intrusion_detection_system.

[10] wikipedia, "Stack canaries." https://en.wikipedia.org/wiki/Stack_buffer_overflow#Stack_canaries.

[11] wikipedia, "Address space layout randomization." https://en.wikipedia.org/wiki/Address_space_layout_randomization.

[12] wikipedia, "Position independet executable." https://en.wikipedia.org/wiki/Position-independent_code.

[13] W. A. Arbaugh, W. L. Fithen, and J. McHugh, "Windows of vulnerability: A case study analysis," *IEEE Computer*, vol. 33, December 2000.

[14] H. Okhravi and D. Nicol, "Evaluation of patch management strategies," *International Journal of Computational Intelligence: Theory and Practice*, vol. 3, no. 2, pp. 109–117, 2008.

[15] S. Frei, *Security Econometrics: The Dynamics of (In)Security*. PhD thesis, ETH Zürich, 2009.

[16] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *International Conference on Software Engineering*, 2012.

[17] L. Bilge and T. Dumitraş, "Before we knew it: an empirical study of zero-day attacks in the real world," in *ACM Conference on Computer and Communications Security*, pp. 833–844, ACM, 2012.

[18] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson, "The matter of Heartbleed," in *Internet Measurement Conference*, (Vancouver, Canada), Nov 2014.

[19] L. Zhang, D. Choffnes, D. Levin, T. Dumitras, A. Mislove, A. Schulman, and C. Wilson, "Analysis of ssl certificate reissues and revocations in the wake of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, (New York, NY, USA), pp. 489–502, ACM, 2014.

[20] T. Dübendorfer and S. Frei, "Web browser security update effectiveness," in *CRITIS Workshop*, September 2009.

[21] T. Dumitraş and D. Shou, "Toward a standard benchmark for computer security research: The Worldwide Intelligence Network Environment (WINE)," in *EuroSys BADGERS Workshop*, (Salzburg, Austria), Apr 2011.

[22] "U.s. national vulnerability database." http://nvd.nist.gov/.

[23] "Osvdb: Open sourced vulnerability database." http://osvdb.org/.

[24] "Exploit database." http://exploit-db.com/.

[25] "Virustotal." http://www.virustotal.com/.

[26] J. Caballero, C. Grier, C. Kreibich, and V. Paxson, "Measuring pay-per-install: The commoditization of malware distribution," in *USENIX Security Symposium*, (San Francisco, CA), August 2011.

[27] D. Dittrich and S. Dietrich, "P2p as botnet command and control: A deeper insight," in *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*, pp. 41–48, Oct 2008.

[28] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G. M. Voelker, "Manufacturing compromise: The emergence of exploit-as-a-service," in *ACM Conference on Computer and Communications Security*, 2012.

[29] Wikipedia, "Bulletproof hosting https://en.wikipedia.org/wiki/Bulletproof_hosting."

[30] E. Kartaltepe, J. Morales, S. Xu, and R. Sandhu, *Social Network-Based Botnet Command-and-Control: Emerging Threats and Countermeasures*, vol. 6123 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010.

[31] J. Wyke, "The zeroaccess botnet: Mining and fraud for massive financial gain," September 2012. http://www.sophos.com/en-us/why-sophos/our-people/technical-papers/zeroaccess-botnet.asp:x.

[32] C. Kreibich and J. Crowcroft, "Honeycomb: Creating intrusion detection signatures using honeypots," *SIGCOMM Computer Communications Review*, vol. 34, no. 1, 2004.

[33] Wikipedia, "Spamtrap https://en.wikipedia.org/wiki/Spamtrap."

[34] "Snort." http://www.snort.org/.

[35] V. Paxson, "bro: A system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23–24, 1999.

[36] P. Amini, "kraken botnet infiltration," April 2008. http://dvlabs.tippingpoint.com/blog/2008/04/28/kraken-botnet-infiltration.

[37] "Google safe browsing." https://developers.google.com/safe-browsing/.

[38] "Dorkbot takedown 2015." Dorkbot Takedown 2015 http://www.theregister.co.uk/2015/12/04/dorkbot_botnet_takedown/.

[39] N. Krawetz, "Average perceptual hash," May 2011. http://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html.

[40] M. Z. Rafique and J. Caballero, "Firma: Malware clustering and network signature generation with mixed network behaviors," in *International Symposium on Recent Advances in Intrusion Detection*, 2013.

[41] J. Caballero, M. G. Kang, S. Venkataraman, D. Song, P. Poosankam, and A. Blum, "FiG: Automatic Fingerprint Generation," in *NDSS*, February 2007.

[42] M. S. Weant, "Fingerprinting Reverse Proxies using Timing Analysis of TCP Flows," Master's thesis, Computer Science Department, Naval Postgraduate School, Monterey, CA, September 2013.

[43] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dimitras, "The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, (San Jose, CA), May 2015.

[44] A. Nappa, M. Z. Rafique, and J. Caballero, "Driving in the cloud: An analysis of drive-by download operations and abuse reporting," in *SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, (Berlin, Germany), July 2013.

[45] A. Nappa, M. Z. Rafique, and J. Caballero, "The MALICIA Dataset: Identification and Analysis of Drive-by Download Operations," *International Journal of Information Security*, vol. 14, pp. 15–33, February 2015.

[46] "The malicia project." http://malicia-project.com/.

[47] D. Moore, C. Shannon, and K. C. Claffy, "Code-red: a case study on the spread and victims of an internet worm," in *Internet Measurement Workshop*, pp. 273–284, ACM, 2002.

[48] E. Rescorla, "Security holes... who cares," in *Proceedings of the 12th USENIX Security Symposium*, pp. 75–90, 2003.

[49] T. Ramos, "The laws of vulnerabilities," in *RSA Conference*, 2006.

[50] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage, "When private keys are public: results from the 2008 debian openssl vulnerability," in *Internet Measurement Conference*, pp. 15–27, ACM, 2009.

[51] C. Gkantsidis, T. Karagiannis, P. Rodriguez, and M. Vojnovic, "Planet scale software updates," in *SIGCOMM*, pp. 423–434, ACM, 2006.

[52] T. Dübendorfer and S. Frei, "Web browser security update effectiveness," in *CRITIS Workshop*, September 2009.

[53] S. Frei and T. Kristensen, "The security exposure of software portfolios," in *RSA Conference*, March 2010.

[54] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King, "Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities," in *Network and Distributed System Security Symposium*, (San Diego, CA), February 2006.

[55] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy, "A crawler-based study of spyware on the web," in *Network and Distributed System Security Symposium*, (San Diego, CA), February 2006.

[56] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, "The ghost in the browser: Analysis of web-based malware," in *USENIX Workshop on Hot Topics on Understanding Botnets*, (Cambridge, United Kingdom), April 2007.

[57] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All Your iFRAMEs Point to Us," in *USENIX Security Symposium*, 2008.

[58] M. Polychronakis, P. Mavrommatis, and N. Provos, "Ghost turns zombie: Exploring the life cycle of web-based malware," in *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, (San Francisco, CA), April 2008.

[59] J. Zhang, C. Seifert, J. W. Stokes, and W. Lee, "Arrow: Generating signatures to detect drive-by downloads," in *International World Wide Web Conference*, (Hyderabad, India), April 2011.

[60] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Low-overhead mostly static javascript malware detection," in *USENIX Security Symposium*, 2011.

[61] C. Y. Cho, J. Caballero, C. Grier, V. Paxson, and D. Song, "Insights from the inside: A view of botnet management from infiltration," in *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, (San Jose, CA), April 2010.

[62] D. Canali, D. Balzarotti, and A. Francillon, "The role of web hosting providers in detecting compromised websites," in *www*, 2013.

[63] B. Stone-Gross, Christopher, Kruegel, K. Almeroth, A. Moser, and E. Kirda, "Fire: Finding rogue networks," in *Annual Computer Security Applications Conference*, 2009.

[64] C. Shue, A. J. Kalafut, and M. Gupta, "Abnormally malicious autonomous systems and their internet connectivity," *IEEE/ACM Transactions of Networking*, vol. 20, February 2012.

[65] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy, "Studying spamming botnets using botlab," in *Symposium on Networked System Design and Implementation*, 2009.

[66] C. Kreibich, N. Weaver, C. Kanich, W. Cui, and V. Paxson, "gq: Practical containment for measuring modern malware systems," in *Internet Measurement Conference*, 2011.

[67] C. Rossow, C. J. Dietrich, H. Bos, L. Cavallaro, M. van Steen, F. C. Freiling, and N. Pohlmann, "Sandnet: Network traffic analysis of malicious software," in *Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, (Salzburg), April 2011.

[68] M. Bailey, J. Oberheide, J. Andersen, Z. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in *International Symposium on Recent Advances in Intrusion Detection*, (Queensland), September 2007.

[69] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, (Paris, France), July 2008.

[70] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Network and Distributed System Security Symposium*, (San Diego, CA), February 2009.

[71] R. Perdisci, W. Lee, and N. Feamster, "Behavioral clustering of http-based malware and signature generation using malicious network traces," in *Symposium on Networked System Design and Implementation*, 2010.

[72] D. S. Anderson, C. Fleizach, S. Savage, and G. M. Voelker, "Spamscatter: Characterizing internet scam hosting infrastructure," in *USENIX Security Symposium*, (Boston, MA), August 2007.

[73] J. Caballero, C. Grier, C. Kreibich, and V. Paxson, "Measuring pay-per-install: The commoditization of malware distribution," in *USENIX Security Symposium*, 2011.

[74] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: Feature hashing malware for scalable triage and semantic analysis," in *ACM Conference on Computer and Communications Security*, 2011.

[75] R. Perdisci and M. U, "Vamo: Towards a fully automated malware clustering validity analysis," in *Annual Computer Security Applications Conference*, 2012.

[76] D. E. Comer and J. C. Lin, "Probing tcp implementations," in *USENIX Summer Technical Conference*, (Boston, MA), June 1994.

[77] Fyodor, "Remote os detection via tcp/ip stack fingerprinting," December 1998. http://www.phrack.com/phrack/51/P51-11.

[78] Tenable, "Nessus." http://www.tenable.com/products/nessus; [Online; accessed 18-Dec-2014].

[79] "Fpdns." http://www.rfc.se/fpdns/.

[80] T. Kohno, A. Broido, and K. Claffy, "Remote physical device fingerprinting," in *IEEE Symposium on Security and Privacy*, (Oakland, CA), May 2005.

[81] P. Eckersley, "How unique is your web browser?," in *pets*, 2010.

[82] G. Gu, V. Yegneswaran, P. Porras, J. Stoll, and W. Lee, "Active botnet probing to identify obscure command and control channels," in *Annual Computer Security Applications Conference*, 2009.

[83] Z. Xu, L. Chen, G. Gu, and C. Kruegel, "Peerpress: Utilizing enemies' p2p strength against them," in *ACM Conference on Computer and Communications Security*, 2012.

[84] M. Marquis-Boire, B. Marczak, C. Guarnieri, and J. Scott-Railton, "For their eyes only: The commercialization of digital spying." https://citizenlab.org/2013/04/for-their-eyes-only-2/.

[85] M. Smart, G. R. Malan, and F. Jahanian, "Defeating tcp/ip stack fingerprinting," in *USENIX Security Symposium*, (Washington, D.C.), August 2000.

[86] G. R. Malan, D. Watson, and F. Jahanian, "Transport and application protocol scrubbing," in *IEEE International Conference on Computer Communications*, (Tel Aviv, Israel), March 2000.

[87] Z. Xu, A. Nappa, R. Baykov, G. Yang, J. Caballero, and G. Gu, "Autoprobe: Towards automatic active malicious server probing using dynamic binary analysis," in *Proceedings of the 21st ACM Conference on Computer and Communications Security(CCS14)*, November 2014.

[88] J. Caballero, M. G. Kang, S. Venkataraman, D. Song, P. Poosankam, and A. Blum, "FiG: Automatic Fingerprint Generation," in *NDSS*, February 2007.

[89] C. Kreibich and J. Crowcroft, "Honeycomb - creating intrusion detection signatures using honeypots," in *Workshop on Hot Topics in Networks*, (Boston, MA), November 2003.

[90] H.-A. Kim and B. Karp, "Autograph: Toward automated, distributed worm signature detection," in *USENIX Security Symposium*, (San Diego, CA), August 2004.

[91] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," in *Symposium on Operating System Design and Implementation*, (San Francisco, CA), December 2004.

[92] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *IEEE Symposium on Security and Privacy*, (Oakland, CA), May 2005.

[93] K. Wang, G. Cretu, and S. J. Stolfo, "Anomalous payload-based worm detection and signature generation," in *International Symposium on Recent Advances in Intrusion Detection*, (Seattle, WA), September 2005.

[94] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha, "An architecture for generating semantics-aware signatures," in *USENIX Security Symposium*, (Baltimore, MD), July 2005.

[95] Z. Li, M. Sanghi, B. Chavez, Y. Chen, and M.-Y. Kao, "Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience," in *IEEE Symposium on Security and Privacy*, (Oakland, CA), May 2006.

[96] K. Rieck, G. Schwenk, T. Limmer, T. Holz, and P. Laskov, "Botzilla: Detecting the phoning home of malicious software," in *ACM Symposium on Applied Computing*, 2010.

[97] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation," *ACM SIGOPS Operating Systems Review*, vol. 40, October 2006.

[98] N. Provos and P. Honeyman, "Scanssh - scanning the internet for ssh servers," Tech. Rep. CITI TR 01-13, University of Michigan, October 2001.

[99] D. Dagon, C. Lee, W. Lee, and N. Provos, "Corrupted dns resolution paths: The rise of a malicious resolution authority," in *Network and Distributed System Security Symposium*, 2008.

[100] N. Heninger, Z. Durumeric, E. Wustrow, and J. Halderman, "Mining your ps and qs: Detection of widespread weak keys in network devices," in *USENIX Security Symposium*, 2012.

[101] D. Leonard and D. Loguinov, "Demystifying service discovery: Implementing an internet-wide scanner," in *Internet Measurement Conference*, 2010.

[102] S. Staniford, V. Paxson, and N. Weaver, "How to 0wn the internet in your spare time," in *USENIX Security Symposium*, (San Francisco, CA), August 2002.

[103] L. Rizzo, "Netmap: A novel framework for fast packet i/o," in *usenixatc*, 2012.

[104] Z. Durumeric, E. Wustrow, and J. A. Halderman, "Zmap: Fast internet-wide scanning and its security applications," in *USENIX Security Symposium*, 2013.

[105] L. Wang, F. Douglis, , and M. Rabinovich, "Forwarding requests among reverse proxies," in *International Web Caching and Content Delivery Workshop*, May 2000.

[106] N. Weaver, C. Kreibich, M. Dam, and V. Paxson, "Here Be Web Proxies," in *PAM*, 2014.

[107] "http_trace.nasl." http://plugins.openvas.org/nasl.php?oid=11040.

[108] E. Marcussen, "Http fingerprinting - the next generation," 2012. https://www.owasp.org/images/f/f8/AppsecAPAC2012-HTTP-fingerprinting-TNG.pdf.

[109] E. I. Bols, "htrosbif - freecode." http://mac.freecode.com/projects/htrosbif.

[110] N. Gregoire, "Traceroute-like http scanner," November 2011. http://www.agarri.fr/kom/archives/2011/11/12/traceroute-like_http_scanner/index.html.

[111] M. S. Weant, "Fingerprinting Reverse Proxies using Timing Analysis of TCP Flows," Master's thesis, Computer Science Department, Naval Postgraduate School, Monterey, CA, September 2013.

[112] J. M. Bello-Rivas, "Halberd," August 2010. https://github.com/jmbr/halberd.

[113] C. Shaffer, "Identifying Load Balancers in Penetration Testing," tech. rep., SAN, March 2010.

[114] S. Gauci and W. G. Henrique, "wafw00f." https://github.com/sandrogauci/wafw00f; [Online; accessed 18-Dec-2014].

[115] S. Shah, "An introduction to http fingerprinting," May 2014. http://www.net-square.com/httprint_paper.html.

[116] L. S. Online, "Http fingerprint." http://www.carnal0wnage.com/papers/LSO-HTTP-Fingerprinting.pdf; [Online; accessed 19-Nov-2014].

[117] M. Raef, "w3dt.net - httprecon (server fingerprint)." https://w3dt.net/tools/httprecon; [Online; accessed 19-Nov-2014].

[118] T. Book, M. Witick, and D. S.Wallach, "Automated generation of web server fingerprints," May 2013. http://arxiv.org/abs/1305.0245.

[119] S. Shah, "Http fingerprinting and advanced assessment techniques," 2003. http://www.blackhat.com/presentations/bh-usa-03/bh-us-03-shah/bh-us-03-shah.ppt.

[120] D. W. Lee, "HMAP: A Technique and Tool For Remote Identification of HTTP Servers," Master's thesis, University of California at Davis, 2001.

[121] "Nmap." http://www.insecure.org.

[122] ErrorMint, "Errormint." http://sourceforge.net/projects/errormint/; [Online; accessed 18-Dec-2014].

[123] C. Grier *et al.*, "Manufacturing compromise: the emergence of exploit-as-a-service," in *ACM Conference on Computer and Communications Security*, (Raleigh, NC), Oct 2012.

[124] L. Allodi and F. Massacci, "A preliminary analysis of vulnerability scores for attacks in wild," in *CCS BADGERS Workshop*, (Raleigh, NC), Oct 2012.

[125] W. R. Marczak, J. Scott-Railton, M. Marquis-Boire, and V. Paxson, "When governments hack opponents: A look at actors and technology," in *USENIX Security Symposium*, 2014.

[126] S. Hardy, M. Crete-Nishihata, K. Kleemola, A. Senft, B. Sonne, G. Wiseman, P. Gill, and R. J. Deibert, "Targeted threat index: Characterizing and quantifying politically-motivated targeted malware," in *USENIX Security Symposium*, 2014.

[127] S. L. Blond, A. Uritesc, C. Gilbert, Z. L. Chua, P. Saxena, and E. Kirda, "A Look at Targeted Attacks through the Lense of an NGO," in *USENIX Security Symposium*, August 2014.

[128] D. G. Kleinbaum and M. Klein, *Survival Analysis: A Self-Learning Text*. Springer, third ed., 2011.

[129] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *IEEE Symposium on Security and Privacy*, (Oakland, CA), pp. 143–157, May 2008.

[130] "The WebKit open source project." http://webkit.org.

[131] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "Vex: Vetting browser extensions for security vulnerabilities.," in *USENIX Security Symposium*, USENIX Association, 2010.

[132] T. Dumitras and P. Efstathopoulos, "The provenance of wine," in *EDCC 2012*, May 2012.

[133] Google, "Chrome releases." http://googlechromereleases.blogspot.com.es/.

[134] "Safari version history." http://en.wikipedia.org/wiki/Safari_version_history.

[135] T. M. Therneau, *A Package for Survival Analysis in S*, 2014. R package version 2.37-7.

[136] E. E. Papalexakis, T. Dumitraş, D. H. P. Chau, B. A. Prakash, and C. Faloutsos, "Spatio-temporal mining of software adoption & penetration," in *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, (Niagara Falls, CA), Aug 2103.

[137] "Heartbleed bug healt report." http://zmap.io/heartbleed/.

[138] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford-Chen, and N. Weaver, "Inside the slammer worm," *IEEE Security & Privacy*, vol. 1, no. 4, pp. 33–39, 2003.

[139] C. Shannon and D. Moore, "The spread of the witty worm," *IEEE Security & Privacy*, vol. 2, no. 4, pp. 46–50, 2004.

[140] Symantec Corporation, "Symantec Internet security threat report, volume 16," April 2011.

[141] Symantec Corporation, "Symantec Internet security threat report, volume 17." http://www.symantec.com/threatreport/, April 2012.

[142] Microsoft, "Zeroing in on malware propagation methods." Microsoft Security Intelligence Report, 2013.

[143] "Private communication from leading security vendor.."

[144] K. Nayak, D. Marino, P. Efstathopoulos, and T. Dumitraş, "Some vulnerabilities are different than others: Studying vulnerabilities and attack surfaces in the wild," in *International Symposium on Research in Attacks, Intrusions and Defenses*, (Gothenburg, Sweeden), Sep 2014.

[145] A. Lineberry, T. Strazzere, and T. Wyatt, "Don't hate the player, hate the game: Inside the Android security patch lifecycle," in *Blackhat*, 2011.

[146] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire OS distributions," in *IEEE Symposium on Security and Privacy,May 2012, San Francisco, California, USA*, pp. 48–62, IEEE Computer Society, 2012.

[147] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel, "Staged deployment in Mirage, an integrated software upgrade testing and distribution system," in *ACM Symposium on Operating Systems Principles, Stevenson, Washington, USA*, pp. 221–236, ACM, 2007.

[148] S. Quinn, K. Scarfone, M. Barrett, and C. Johnson, "Guide to adopting and using the security content automation protocol (SCAP) version 1.0." NIST Special Publication 800-117, Jul 2010.

[149] S. Dery, "Using whitelisting to combat malware attacks at fannie mae," *IEEE Security & Privacy*, vol. 11, August 2013.

[150] R. J. Walls, B. N. Levine, M. Liberatore, and C. Shields, "Effective digital forensics research is investigator-centric," in *USENIX Workshop on Hot Topics in Security*, (San Francisco, CA), August 2011.

[151] "Love vps." http://www.lovevps.com/.

[152] T. Morrison, "How hosting providers can battle fraudulent sign-ups," October 2012. http://www.spamhaus.org/news/article/687/how-hosting-providers-can-battle-fraudulent-sign-ups.

[153] "Malicia project." http://malicia-project.com/.

[154] Xylitol, "Blackhole exploit kits update to v2.0," September 2011. http://malware.dontneedcoffee.com/2012/09/blackhole2.0.html.

[155] Z. Li, S. Alrwais, Y. Xie, F. Yu, and X. Wang, "Finding the Linchpins of the Dark Web: A Study on Topologically Dedicated Hosts on Malicious Web Infrastructures," in *IEEE Symposium on Security and Privacy*, 2013.

[156] "Malware domain list." http://www.malwaredomainlist.com/.

[157] "Urlquery." http://urlquery.net/.

[158] "Ssdsandbox." http://xml.ssdsandbox.net/dnslookup-dnsdb.

[159] "Bfk: Passive dns replication." http://www.bfk.de/bfk_dnslogger.html.

[160] M. Z. Rafique, C. Huygens, and J. Caballero, "Network dialog minimization and network dialog diffing: Two novel primitives for network security applications," Tech. Rep. TR-IMDEA-SW-2014-001, IMDEA Software Institute, Madrid, Spain, March 2014. https://software.imdea.org/~juanca/papers/TR-IMDEA-SW-2014-001.pdf.

[161] C. Zauner, "Implementation and benchmarking of perceptual image hash functions," Master's thesis, Upper Austria University of Applied Sciences, Hagenberg, Austria, July 2010.

[162] "Suricata." http://suricata-ids.org/.

[163] C. Rossow and C. J. Dietrich, "Provex: Detecting botnets with encrypted command and control channels," in *SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2013.

[164] "An overview of exploit packs (update 20) jan 2014." http://contagiodump.blogspot.com.es/2010/06/overview-of-exploit-packs-update.html.

[165] "Wepawet." https://wepawet.iseclab.org/.

[166] "Allatori java obfuscator." http://www.allatori.com/.

[167] "Zelix klassmaster heavy duty protection." http://www.zelix.com/klassmaster/.

[168] L. Kaufman and P. J. Rousseeuw, *Finding Groups In Data: An Introduction To Cluster Analysis*, vol. 4. Wiley-Interscience, 1990.

[169] J. C. Dunn, "Well-separated clusters and optimal fuzzy partitions," *Journal of Cybernetics*, vol. 4, no. 1, 1974.

[170] L. Daigle, "Whois protocol specification." RFC 3912 (Draft Standard), September 2004.

[171] Y. Shafranovich, J. Levine, and M. Kucherawy, "An extensible format for email feedback reports." RFC 5965 (Proposed Standard), August 2010. Updated by RFC 6650.

[172] J. Falk, "Complaint feedback loop operational recommendations." RFC 6449 (Informational), November 2011.

[173] J. Falk and M. Kucherawy, "Creation and use of email feedback reports: An applicability statement for the abuse reporting format (arf)." RFC 6650 (Proposed Standard), June 2012.

[174] "X-arf: Network abuse reporting 2.0." http://x-arf.org/.

[175] D. Crocker, "Mailbox names for common services, roles and functions." RFC 2142 (Proposed Standard), May 1997.

[176] "The spamhaus project," October 2012. http://www.spamhaus.org/.

[177] Caida, "As ranking," October 2012. http://as-rank.caida.org.

[178] "Cool exploit kit - a new browser exploit pack." http://malware.dontneedcoffee.com/2012/10/newcoolek.html/.

[179] Xylitol, "Tracking cyber crime: Hands up affiliate (ransomware)," December 2011. http://www.xylibox.com/2011/12/tracking-cyber-crime-affiliate.html.

[180] "New dutch notice-and-take-down code raises questions," October 2008. http://www.edri.org/book/export/html/1619.

[181] D. Dittrich, "So you want to take over a botnet...," in *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, (San Jose, CA), April 2012.

[182] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, "Your botnet is my botnet: Analysis of a botnet takeover," in *ACM Conference on Computer and Communications Security*, 2009.

[183] "Vxvault." http://vxvault.siri-urz.net.

[184] "Collection of pcap files from malware analysis." http://contagiodump.blogspot.com.es/2013/04/collection-of-pcap-files-from-malware.html/.

[185] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage, "Scalability, fidelity, and containment in the potemkin virtual honeyfarm," in *Symposium on Operating Systems Principles*, (Brighton, United Kingdom), October 2005.

[186] "Alexa - the web information company." http://www.alexa.com/.

[187] P. Haffner, S. Sen, O. Spatscheck, and D. Wang, "acas: Automated construction of application signatures," in *ACM Workshop on Mining network data*, (Philadelphia, PA), October 2005.

[188] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer, "Dynamic application-layer protocol analysis for network intrusion detection," in *USENIX Security Symposium*, (Vancouver, Canada), July 2006.

[189] "Html::similarity." http://search.cpan.org/~xern/HTML-Similarity-0.2.0/lib/HTML/Similarity.pm/.

[190] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms*, vol. 2, no. 1, 2004.

[191] D. Benoit and A. Trudel, "World's first web census," *International Journal of Web Information System*, vol. 3, 2007.

[192] "University of oregon route views project." http://www.routeviews.org/.

[193] D. E. Knuth, *The Art Of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[194] "Tcpdump/libpcap." http://www.tcpdump.org/.

[195] "Libevent." http://libevent.org/.

[196] "Shoutcast." http://www.shoutcast.com/.

[197] "Virusshare." http://virusshare.com/.

[198] "Tracking cyber crime: Inside the fakeav business." http://www.xylibox.com/2011/06/tracking-cyber-crime-inside-fakeav.html.

[199] "The missing link - some lights on urausy affiliate." http://malware.dontneedcoffee.com/2013/05/the-missing-link-some-lights-on-urausy.html.

[200] "Blackhole exploit kit v2 on the rise." http://research.zscaler.com/2012/10/blackhole-exploit-kit-v2-on-rise.html.

[201] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *ACM Conference on Computer and Communications Security*, 2007.

[202] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *Network and Distributed System Security Symposium*, (San Diego, CA), February 2008.

[203] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage, "Spamalytics: An Empirical Analysis of Spam Marketing Conversion," in *CCS*, October 2008.

[204] T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling, "Measuring and Detecting Fast-Flux Service Networks," in *Network and Distributed Systems Security Symposium*, 2008.

[205] G. Barish and K. Obraczke, "World wide web caching: Trends and techniques," *IEEE Communications magazine*, vol. 38, no. 5, pp. 178–184, 2000.

[206] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "SWAP: Mitigating XSS Attacks Using a Reverse Proxy," in *Workshop on Software Engineering for Secure Systems*, 2009.

[207] S. Labs, "RIG rig exploit kit," February 2015. https://www.trustwave.com/Resources/SpiderLabs-Blog/RIG-Exploit-Kit-%E2%80%93-Diving-Deeper-into-the-Infrastructure/.

[208] A. Nappa, Z. Xu, J. Caballero, and G. Gu, "Cyberprobe: Towards internet-scale active detection of malicious servers," in *Network and Distributed System Security Symposium*, 2014.

[209] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, October 2009.

[210] "Haproxy." http://www.haproxy.org/.

[211] "Pound." http://www.apsis.ch/pound/.

[212] "Apache Web server." http://httpd.apache.org.

[213] "nginx." http://nginx.org/.

[214] "Dns-bh malware domain blocklist." http://www.malwaredomains.com/.

[215] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1." RFC 2616 (Draft Standard), June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

[216] "http_version.nasl." http://plugins.openvas.org/nasl.php?oid=10107.

[217] "Nikto2—cirt.net." https://cirt.net/Nikto2.

[218] "Openvas - open vulnerability assessment system." http://www.openvas.org.

[219] R. Deraison, "The nessus attack scripting language reference guide." http://virtualblueness.net/nasl.html.

[220] P. Group, "phpinfo manual." http://php.net/manual/en/function.phpinfo.php.

[221] OWASP, "Owasp cookies database." https://www.owasp.org/index.php/Category:OWASP_Cookies_Database; [Online; accessed 26-Jan-2015].

[222] "Big-ip." https://f5.com/products/big-ip.

[223] "Kemp." http://kemptechnologies.com/.

[224] "Inserting an x-via http header to identify connections that have passed through the big-ip webaccelerator." https://support.f5.com/kb/en-us/solutions/public/8000/900/sol8912.html.

[225] "Understanding cache hit and miss headers with shielded services." https://docs.fastly.com/guides/shielding/understanding-cache-hit-and-miss-headers-with-shielded-services.

[226] "X-cache and x-cache-lookup." http://blog.lyte.id.au/2014/08/28/x-cache-and-x-cache-lookupheaders/.

[227] "Varnish." https://www.varnish-cache.org/docs/4.0/.

[228] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol description generation from network traces," in *USENIX Security Symposium*, (Boston, MA), August 2007.

[229] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," in *IEEE Symposium on Security and Privacy*, (San Francisco, CA), May 2012.

[230] D. Y. Wang, S. Savage, and G. M. Voelker, "Cloak and Dagger: Dynamics of Web Search Cloaking," in *ACM Conference on Computer and Communications Security*, 2011.