EFFECTIVE AND SCALABLE BOTNET DETECTION IN NETWORK TRAFFIC

A Thesis
Presented to
The Academic Faculty

by

Junjie Zhang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2012

EFFECTIVE AND SCALABLE BOTNET DETECTION IN NETWORK TRAFFIC

Approved by:

Professor Wenke Lee, Advisor
College of Computing
Georgia Institute of Technology

Professor Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Professor Nick Feamster
College of Computing
Georgia Institute of Technology

Professor Patrick Traynor
College of Computing
Georgia Institute of Technology

Professor John Copeland
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: June 04, 2012

To my family,

for their endless love and support.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Prof. Wenke Lee, for his invaluable guidance and support during my PhD study. Prof. Lee has been a wonderful advisor. I am always inspired by his principle of conducting high-quality and impactful research, his vision on new research problems, and his breadth of knowledge. I have benefited tremendously from his professional training. Prof. Lee sets an excellent example as a great and successful researcher and professor that I can follow through my future career.

I would like to thank other members of my dissertation committee, Prof. Mustaque Ahamad, Prof. Nick Feamster, Prof. Patrick Traynor, and Prof. John Copeland. Their insightful comments and invaluable suggestions have significantly improved the quality of my dissertation work.

A special thank goes to Dr. Yinglian Xie, Dr. Fang Yu, Dr. Jack W. Stokes, and Dr. Christian Seifert, for mentoring me during my summer internships at Microsoft Research. Their support and advice are indispensable for the completion of my dissertation work.

I would also like to express my deep gratitude to Prof. Roberto Perdisci, Prof. Guofei Gu, and Dr. Xiapu Luo, for all the valuable work done together that leads to the completion of this dissertation. I am especially grateful to Paul Royal, Dr. Christopher P. Lee, and Robert Edmonds, for providing me with experimental data. I want to thank Dr. Kapil Singh, Dr. Abhinav Srivastava, Martim Carbone, Manos Antonakakis, Long Lu, David Dagon, Dr. Monirul Sharif, Takehiro Takahashi, Brendan Dolan-Gavit, Ikpeme Erete, Daisuke Mashima, Yacin Nadji, Chengyu Song, Xinyu Xing, Tielei Wang, and everyone in the GTISC lab for their assistance in research as

well as friendship.

I would like to thank my parents, Hong Zhang and Zhenzhi Zhang, for their unconditional love and endless support. I am indebted to my wife, Rui Dai, for helping and supporting me all the time. Without their love, I would not have been able to complete this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Botnets represent one of the most serious threats against Internet security since they serve as platforms that are responsible for the vast majority of large-scale and coordinated cyber attacks, such as distributed denial of service, spamming, and information stolen. Detecting botnets is therefore of great importance and a number of network-based botnet detection systems have been proposed. However, as botnets perform attacks in an increasingly stealthy way and the volume of network traffic is rapidly growing, existing botnet detection systems are faced with significant challenges in terms of effectiveness and scalability.

The objective of this dissertation is to build novel network-based solutions that can boost both the effectiveness of existing botnet detection systems by detecting botnets whose attacks are very hard to be observed in network traffic, and their scalability by adaptively sampling network packets that are likely to be generated by botnets. To be specific, this dissertation describes three unique contributions.

First, we built a new system to detect drive-by download attacks, which represent one of the most significant and popular methods for botnet infection. The goal of our system is to boost the effectiveness of existing drive-by download detection systems by detecting a large number of drive-by download attacks that are missed by these existing detection efforts.

Second, we built a new system to detect botnets with peer-to-peer (P2P) command&control (C&C) structures (i.e., P2P botnets), where P2P C&Cs represent currently the most robust C&C structures against disruption efforts. Our system aims to boost the effectiveness of existing P2P botnet detection by detecting P2P botnets

in two challenging scenarios: i) botnets perform stealthy attacks that are extremely hard to be observed in the network traffic; ii) bot-infected hosts are also running legitimate P2P applications (e.g., Bittorrent and Skype).

Finally, we built a novel traffic analysis framework to boost the scalability of existing botnet detection systems. Our framework can effectively and efficiently identify a small percentage of hosts that are likely to be bots, and then forward network traffic associated with these hosts to existing detection systems for fine-grained analysis, thereby boosting the scalability of existing detection systems. Our traffic analysis framework includes a novel botnet-aware and adaptive packet sampling algorithm, and a scalable flow-correlation technique.

CHAPTER I

INTRODUCTION

1.1 Background

A botnet is defined as *a collection of bot-infected computers (*a.k.a *bots) that are remotely controlled by an attacker (*a.k.a *botmaster) via a command and control (C&C) channel in order to commit a variety of malicious activities* [36]. Botnets can serve as platforms capable of launching various large-scale and coordinated cyber attacks such as spamming [39], distributed denial of service (DDoS) attacks [4], phishing [28], click fraud [64, 62, 21], search engine abuse [44, 43], and information theft [17]. Dating back to 2007, Vint Cerf, who is recognized as one of fathers of Internet, estimated that up to a quarter of all computers are part of botnets [61]. The following five years unfortunately have experienced an aggravation of the botnet threat. On the one hand, botnets grow fast in terms of both population and diversity, and they have started to contaminate emerging infrastructures such as as smart phones [24] and industrial control systems [63]. On the other hand, botnets become increasingly stealthy and robust, introducing great challenges against detection and disruption efforts. As a consequence, botnets have been recognized as one of the most serious threats against Internet security and even national security.

It is therefore imperative to detect, mitigate, and prevent botnets. Building botnet detection systems is of fundamental importance since it serves as an indispensable step for further mitigation and prevention actions. Network-based botnet detection systems are particularly desired due to their visibility of network behaviors of all hosts in monitored networks and their tamper-resistant nature as a result of complete system-level isolation from bot-infected hosts. In order to detect botnets, we divide

the life cycle of a botnet into two critical phases according to its definition, namely *the infection phase* and *the control phase*.

1.1.1 The Infection Phase of Botnets

The *infection phase* is referred to as the phase in which a clean host is infected by a bot binary and then becomes a member of a botnet.

A variety of approaches could be used by attackers for botnet infection. Among them, the scanning-and-exploiting approach has been used for decades, from traditional worm propagation [75] in the 1980s to recent IRC-based bot propagation [59] and Conficker bot propagation [67]. In a scanning-and-exploiting attack, an attacker compromises a victim host by first initiating a network connection to a vulnerable service on that host and then injecting malicious content through the established connection in order to exploit that service. A successful exploitation can lead to the delivery a bot binary to the victim host, and such binary will subvert the host and turn it into a member of a botnet. The scanning-and-exploit approach has enabled astonishing success in worm propagation: in July of 2001, Code-Red worm propagated to more than 359,000 computers in less than 14 hours. Approximately ten years have passed, such infection approach is still used and still function as an important approach for botnet infection: a recent study of global scanning events [96] indicated that many botnets still adopt the scanning-and-exploiting approach for infection. However, the scanning-and-exploiting approach has an intrinsic drawback: an attacker or a bot needs to initiate and establish a network connection to the victim host so that exploit content could be delivered. As operating system vendors tend to disable unnecessary services in their operating systems and network operators are inclined to stop network connections initiated from uninvited hosts by using firewalls, the scanning-and-exploiting approach gradually loses its effectiveness. This motivates attackers to look for powerful alternatives for bot infection.

Spam-based infection serves as an optional solution. In a spam-based infection attempt, an attacker can send a spam email , which encloses bot binary, to a victim user and then attract the user to install the binary. While such approach seems promising given the popularity of email service, its effectiveness is significantly constrained because of the widely deployed spam detection systems [13, 71, 92], and especially its dependence of victim users' interaction (i.e., users have to install the bot binary).

With the advent of web 2.0, web browsers become the dominating network applications, and thus leveraging web browsers for bot infection attracts great endeavors from attackers. Web-based malware infection approaches can be generally categorized into two classes, namely social-engineering-based infection approaches and drive-by download attacks [11]. Similar to spam-based infection attacks discussed above, social-engineering-based attacks require victim users' interaction. A typical social-engineering-based attack is the rogue security software attack [57]. In such attacks, using security software (e.g., anti-virus tools) as camouflage, attackers host bot binaries in a website and attract innocent users to install them. While the dependence of users' interaction may hinder the effectiveness of social-engineering-based attacks, drive-by download attacks, on the contrary, take action without users' consent. In a drive-by download attack, an attacker embeds malicious content in the webpage visited by a client browser. When the vulnerable browser renders the webpage, the malicious content will attempt to exploit the vulnerability in the browser, which is usually introduced by poorly designed but unfortunately widely used plugins. A successful exploit causes the bot executable to be downloaded and executed, turning the victim host into a bot. The popularity of web browsers, the widely used vulnerable plugins on various browsers, and the independence of users' consent enable the unparalleled significance of drive-by download attacks. For example, the most notorious botnets such as *Zeus* [7] and *Torpig* [17] have used drive-by downloads for infection. Recent studies also revealed that drive-by download attacks have served as

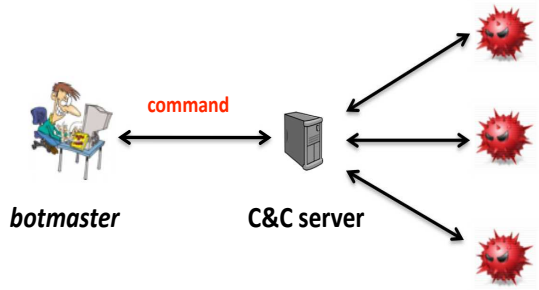


Figure 1: An example of a centralized botnet

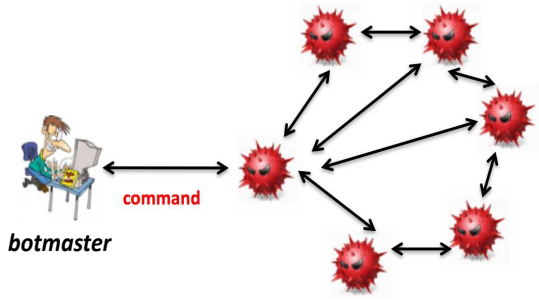


Figure 2: An example of a P2P botnet

the primary way for malware infection [8].

1.1.2 The Control Phase of Botnets

The *control phase* is referred to as the phase in which bot-infected hosts are controlled and coordinated by botmasters to launch cyber attacks.

In the control phase, botmasters rely on command&control (C&C) channels to control and coordinate their bots. Different structures could be used by botmasters to build C&C channels, such as centralized structures, namely centralized botnets, and peer-to-peer structures, namely P2P botnets. As illustrated in Figure 1, bots in a centralized botnet contact a single C&C server in order to fetch commands and report information back. Centralized C&Cs have been widely used mainly due to its simplicity. However, centralized C&Cs structures suffer from the single-point-of-failure problem, where the whole botnet will be disrupted if the central C&C server

is taken down.

In order to overcome such limitation, botmasters recently use P2P structures to organize their bots. As illustrated in Figure 2, all bots in a P2P botnet form a P2P network, and each bot can serve as a client and a server. The botmaster can inject a command into the P2P network and other bots will seek the command and execute it. Without central C&C servers, P2P botnets offer great resilience, since even if a significant portion of a P2P botnet is taken down the remaining bots may still be able to communicate with each other and with the botmaster. Modern botnets have started to use P2P C&Cs. Notable examples of P2P botnets are represented by `Nugache` [68], `Storm` [66], `Waledac` [37], and even `Confiker` [67], which has been shown to embed P2P capabilities.

1.1.3 Existing Botnet Detection Approaches

Detecting botnets in the infection phase is crucial, since it can either protect clean hosts from being compromised or prevent bot-infected hosts from engaging in malicious activities at their early stage. BotHunter [35] is dedicated to detect bot infection based on a pre-defined infection dialog model. Its current implementation mainly leverages scanning-and-exploiting attacks. To be specific, it correlates the "inbound scan", "inbound exploit", "binary download", "C&C communication", and "outbound scanning". Since drive-by download attacks have dominated scanning-and-exploiting attacks, BotHunter may expand its detection capability by extending the infection dialog model in order to cover drive-by download attacks. A natural way is to incorporate a new event that indicates a connection to a website used for the drive-by download attack. Such websites, which are used for drive-by downloads, could be identified by drive-by download detection systems. Drive-by download detection systems could not only enhance BotHunter, but also help network operators or search engines to block those websites used for drive-by download attacks to prevent

bot infection caused by these websites. Therefore, building drive-by download detection systems is very important. A number of drive-by download detection systems have been proposed [11, 41, 51, 56, 91]. These systems share the same characteristic: relying on exploit content or bot binaries presented to browsers for detection.

Although detecting botnets in their infection phase is vital, it is not sufficient. A major reason is that infection activities could be carried out beyond the visibility of deployed network-based detection systems. For example, a computer could be infected by a bot binary from a physically connected USB stick, thereby not arousing any malicious network traffic. As a consequence, building botnet detection systems focusing on botnets' control phase becomes necessary. Since C&Cs are essential for botnets, current detection systems detect C&C channels in network traffic. So far, a few botnet detection systems have been proposed. The representative ones include BotSniffer [33], BotMiner [36], T$\bar{A}$MD [78], and Rishi [40]. These systems experience either of the following two characteristics: i) performing deep packet inspection (DPI) to analyze packet content, and ii) observing attacks initiated by botnets. For example, Rishi [40], BotSniffer [33], and T$\bar{A}$MD [78] detect centralized botnet C&Cs in network traffic by identifying synchronized network communications that share similar packet payload. BotMiner [36] is capable of detecting both centralized and P2P botnets by discovering groups of hosts that share both similar communication patterns and similar attacks.

1.2 New Challenges

These proposed network-based detection systems, which can be used to detect botnets in either the infection phase [35, 11, 41, 46, 51, 56, 91] or the control phase [40, 35, 33, 36], have shown promising detection performance. However, as both botnets and the Internet are actively evolving, these detection systems are faced with two major challenges: the stealthiness of botnets' attacks and the huge volume of network traffic.

First, in reaction to the intensive growth of law enforcement on punishing cyber crimes, botmasters tend to instruct their botnets to perform attacks in an increasingly stealthy way for evading detection. Such stealthy attacks arouse little anomaly and thus become extremely hard to be observed in the network traffic. Take drive-by download attacks as an example, in order to evade detection, attackers could attempt to identify the existence of detection systems first and only deliver exploits and bot executables to hosts that are unlikely to be detectors. As demonstrated in a case study in [47], approximately 56% drive-by download attacks were missed by the existing state-of-the-art detection methods. Additionally, recent study [94] has confirmed that botnets have started to leverage popular email services to send spams, instead of directly sending spams from bot-infected computers. To be specific, a botmaster instructs his/her bots to sign in popular email service (e.g., Hotmail) using hijacked accounts and then send spams through the email service. Such malicious intentions, including signing in and then sending email to popular email service, are very hard to be observed in the monitored networks. Consequently, existing botnet detection systems may fail. For example, systems including [11, 41, 46, 51, 56, 91] could fail to detect a huge number of drive-by download attempts due to the absence of exploits and bot executables. Without observing attacks, BotMiner [36] also lacks evidence to perform correlation between communication patterns and attack patterns.

Second, as the population of networked computers and devices is huge and keeps exploding, the volume of network traffic is high and grows fast. Such huge volume of traffic demands great scalability for network-based detection systems, meaning that the detection systems need to process a large volume of network traffic efficiently. However, as most of existing botnet detection systems [40, 33, 78, 36] rely on deep packet inspection (DPI) to analyze packet content, which is computationally expensive, their scalability is significantly constrained. As a consequence, when these detection systems are deployed in high-speed or high-volume networks, they

may not be able to afford to perform detailed analysis for all network traffic related to bot-infected hosts and thus fail to detect bots.

To summarize, existing network-based botnet detection systems may fail to detect botnets, which represent one of the most serious threats against Internet security, as a result of two practical challenges. First, their effectiveness is significantly limited due to botnets' stealthy attacks, which are very hard to be observed in network traffic. Second, their scalability is greatly constrained due to their dependence of deep packet inspection compounded with the fast growing volume of network traffic. New network-based botnet detection systems are therefore needed to address these challenges.

1.3 Dissertation Overview

The objective of this dissertation is to build effective and scalable network-based botnet detection systems. The dissertation presents ***network-based solutions that can boost both the effectiveness of existing botnet detection systems by detecting botnets whose attacks are very hard to be observed in network traffic, and their scalability by adaptively sampling network packets that are likely to be generated by botnets***. To be specific, this dissertation describes three unique contributions.

First, we built a new system to detect botnets in their infection phase. To be specific, we designed a system [47] to detect drive-by download attacks, which represent one of the most significant and popular methods for botnet infection. *The goal of our system is to boost the effectiveness of existing drive-by download detection systems by detecting a large number of stealthy drive-by download attacks that are missed by these existing detection efforts.* In order to accomplish this goal, our system bootstraps from a set of drive-by download attacks detected by existing systems. It then aggregates individual drive-by download attacks that result in the same bot

executable into a malware distribution network (MDN). Our system further identifies MDNs that have central servers, where a central server covers the vast majority of drive-by download attacks in a MDN. Since central servers are stable and highly likely to be used for malicious purposes, we generate regular expression signatures based on URLs of central servers and then leverage the signatures for drive-by download detection. Experimental results based on an Internet-scale dataset have demonstrated that our system can boost the number of drive-by download attacks by 96% while only introducing an extremely low false positive rate ($7.4 * 10^{-5}\%$). We also found that our system can detect 93,488 drive-by download attacks in average 172 days earlier than several popular public domain/IP reputation systems.

Second, we built a new system to detect botnets in their control phase. Particularly, we designed a system to detect botnets with peer-to-peer (P2P) C&C structures [48], where P2P C&C structures represent currently the most robust C&C structures against disruption efforts. *The goal of our system is to boost the effectiveness of existing P2P botnet detection by detecting P2P botnets in two challenging scenarios*: i) botnets perform malicious activities in a stealthy way and thus their malicious activities are extremely hard to be observed; ii) bot-infected hosts are also running legitimate P2P applications (e.g., Bittorrent and Skype). In order to fulfill this goal, our system first identifies all hosts engaging in P2P communications in the networks (a.k.a, P2P clients), no matter they are legitimate P2P applications or P2P bots. Our system further extracts *fingerprint clusters* to profile P2P clients. Based on the obtained fingerprint clusters, our system finally identifies as a botnet a group of P2P clients that are persistently active on the underlying hosts and at the same time participate in coordinated search behaviors. Experimental results based on real-world network traces have demonstrated that our system can achieve high detection rate (100%) and low false positive rate (0.2%).

Finally, we built a novel traffic analysis framework to facilitate the deployment

of existing botnet detection systems in high-speed and high-volume networks. As we have discussed, the vast majority of botnet detection systems rely on deep packet inspection (DPI). Since the DPI-based analysis is computationally expensive, these systems may suffer from limited scalability on processing a large volume of traffic from high-speed or high-volume networks. *The goal of our work is to boost the scalability of existing detection systems.* In order to achieve this goal, we proposed a novel traffic analysis framework. Our framework can effectively and efficiently identify a small percentage of hosts that are likely to be bots, and then forward the network traffic associated with these hosts to existing detection systems for fine-grained analysis, thereby boosting the scalability of existing detection systems. Our traffic analysis framework includes a novel *botnet-aware* and *adaptive* packet sampling algorithm and a scalable flow-correlation technique. The packet sampling algorithm is bot-aware so that it can focus on capturing packets that are likely related to botnets; it is adaptive so that it can maintain the actual packet sampling rate close to a pre-defined sampling rate. The flow-correlation technique identifies network communications experiencing persistent similarity, which represents a typical behavior of botnet C&Cs. Experimental results based on real-world network traffic demonstrated that our system can enable existing detection systems to achieve good detection performance by only inspecting a small percentage of network traffic (e.g., 5%).

1.4 Organization

The rest of the dissertation is organized as follows:

1. In Chapter 2, we discuss the design of a new system to detect drive-by download attacks. We present our survey on existing drive-by download detection techniques and compare them to our system. We then elaborate on the design of processing components, including identifying malware distribution networks and generating regular expression signatures. The experimental evaluation in

terms of detection rate, false positive rate, and early detection is finally discussed.

2. In Chapter 3, we discuss the design of our P2P botnet detection system. We introduce the definition of *fingerprint cluster*, and present the clustering-based algorithm to derive fingerprint clusters from network traffic. We also present how to use fingerprint clusters to profile P2P applications. The features and the algorithm used for isolating P2P bots from legitimate P2P applications are also discussed. We subsequently present our evaluation results.

3. In Chapter 4, we describe our design of a traffic analysis framework aiming to boost the scalability of existing botnet detection systems. The design of the bot-aware packet sampling algorithm and the design of the scalable flow-correlation are presented. The evaluation based on real-world dataset is presented, followed by the comparison between our sampling algorithm and other sampling algorithms.

4. Chapter 5 summarizes our contributions and discusses potential future research directions.

CHAPTER II

DETECTING DRIVE-BY DOWNLOAD ATTACKS

2.1 Motivation

As discussed in Section 1.1, the life cycle of a botnet can be divided into two phases, namely "*infection phase*" and "*control phase*". Detecting botnets in the infection phase is very important and useful since it can protect clean hosts from being infected or prevent bot-infected hosts from performing malicious activities in their early state. Various approaches could be used for bot infection. Among them, drive-by download attacks become one of the most significant and popular approaches.

Drive-by download attacks are referred to as the download of malware through vulnerable web browsers without users' consent [11]. In drive-by download attacks, attackers embed malicious content in either the original webpage visited by the user, denoted as the landing page, or some content directly, or indirectly, referenced by the landing page, which is usually hosted on a compromised web server. When a web browser renders these webpages, the malicious content attempts to exploit the vulnerability in the browser. A successful exploit attempt often causes malware to be downloaded and executed, converting the underlying operating system to a malware-infected victim such as a bot. Since drive-by download attacks take advantage of web browsers, the most popular Internet applications, and do not require users' interaction, they have dominated the traditional scanning-and-exploit approaches [11], and they have served as the primary way for malware infection [8, 65].

Multiple steps are usually involved in a drive-by download attack, as illustrated in Figure 3. The malicious content embedded in the compromised webpage, which is initially rendered by the browser, usually does not directly exploit the browser.

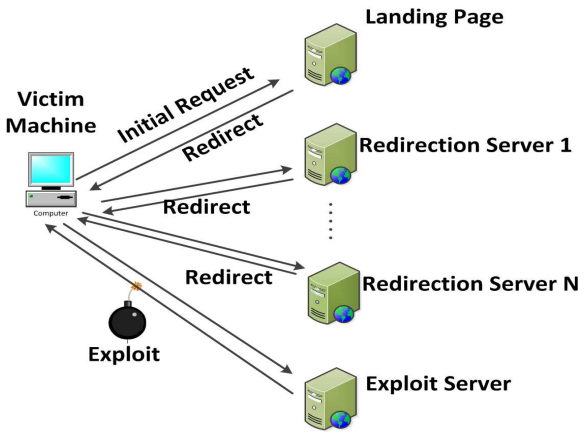


Figure 3: Multiple steps in a drive-by download attack

Instead it redirects the browser to other redirection servers which either provide external webpage content or further redirect to additional servers. After visiting one or more, possibly benign, redirection servers, the browser will eventually encounter a malicious redirection server which further redirects to the servers that attempt to exploit the browser and download malware. The malicious redirection server can be used to manage the attacks and decide the exploit server to use, which has the best matching set of exploits (e.g., IE exploits for IE browsers). A set of different drive-by downloads can be managed by the same attacker for a particular purpose (e.g., distributing the same malware binary for a botnet) and therefore form a *malware distribution network* (MDN). In this chapter, we define a MDN to be a collection of drive-by downloads that serve the same malicious objective such as distributing related malware executables.

Several methods [11, 41, 46, 51, 56, 91] have been proposed to detect drive-by download attacks and are described in detail in Section 2.2. Most of these methods [11, 41, 51, 56, 91] rely on the malicious *webpage content* returned by *servers used for exploits or malware distribution* to detect the attacks. For example, [11, 41, 56, 91] require the exploit content whereas [51] needs the downloaded binary. These systems may fail to detect a large number of drive-by download attacks (i.e. false negatives)

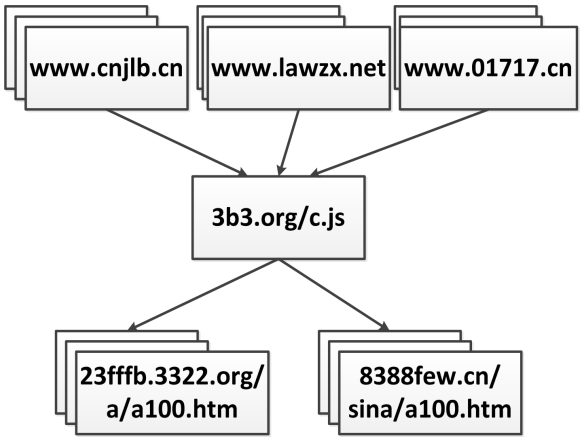


Figure 4: An example of a malware distribution network

if no exploit is detected for several reasons. First, the attackers may be aware of IP addresses of the detection systems. In this case, they can feed detection systems with benign content. Additionally, the detectors may not be configured correctly to match an attack. For example, the detector may present an unpatched version of Internet Explorer, but the malicious landing page may target a FireFox vulnerability. Finally, since the exploit and malware distribution servers may be hosted on compromised servers, their stability will be affected by the network churn. So when the detection system visits these servers, they may be temporally unavailable and thus no exploit attempt occurs in the victim's browser. Although incapable of detecting drive-by download attacks, *WebCop* [46] (see Section 2.2 for details) can handle this problem since it is based on the URLs of exploit/malware-distribution servers. However, the hostnames, IP addresses or the parameter values in the URLs can be frequently changed to make *WebCop* ineffective since it matches the entire URL of the malicious executable.

In this chapter, we will present a novel drive-by download detection system, namely *ARROW* [47]. The objective of our method is to boost the effectiveness of existing drive-by download detection systems by detecting a large number of drive-by downloads that are missed by them. In order to achieve this objective, we leverage

the URL information of the *central servers* in MDNs (See Section 2.3), where a central server is a common server shared by a large percentage of drive-by download samples in the same MDN. The example of an MDN with a central server is presented in Figure 4, where "3b3.org" serves as a central server. A central server usually provides certain critical information to make the drive-by download successful, and it is not necessarily the server used for exploit attempts or malware distribution. For example, it can be a redirection server used to optimize the MDN performance. A central server can even be a legitimate server where certain information is retrieved to calculate the location of the exploit servers dynamically, as presented in Section 2.4.3.3. To be specific, our method bootstraps from the drive-by download samples detected using existing methods, where we first aggregate drive-by download samples into MDNs based on the malware (i.e., hash value) information or the URL of the exploit server. For each MDN, we next discover the central servers if they exist. We further generate signatures in the form of regular expressions based on the URLs for the central servers. These signatures can then be distributed to a search engine or browsers to detect drive-by downloads. The lower half of Figure 5 illustrates our method. These signatures can boost the detection coverage in three ways. First, if a drive-by download attempt reaches the central server without hitting the servers for exploit attempts or malware distribution, our signatures can still detect the attack. Second for a drive-by download attempt, if there is only a URL request to the central server without malicious webpage content returned, our signatures can still detect it since the signatures are independent of the webpage content. Third, the signatures are in the form of regular expressions, which can capture the structural patterns of a central server's URL and therefore outperform exact string matching used by *WebCop*.

We have implemented our method in a system named *ARROW* and validated it using data generated from a large-scale production search engine. As presented in the experimental results in Section 4.4, our method can significantly increase the

detection coverage by 96%, with an extremely low false positive rate ($7.4 * 10^{-5}\%$). It can also detect tens of thousands of drive-by download attacks in average 172 days compared to several popular public domain/IP reputation systems. In summary, we made the following contributions:

1. We provide a method to identify malware distribution networks from millions of individual drive-by download attacks.

2. By *correlating* drive-by download samples, we propose a novel method to generate regular expression signatures of central servers of MDNs to detect drive-by downloads.

3. We build a system called *ARROW* to automatically generate regular expression signatures of central servers of MDNs and evaluate the effectiveness of these signatures.

2.2 Related Work

Since drive-by download attacks have served as the primary way for malware infection, including bot infection, great efforts have been spent to analyze and detect web-based malware attacks. Existing detection methods, summarized in Table 1, can be categorized into 2 classes, namely top-down and bottom-up. The table also illustrates how *ARROW* compares to prior work based on other features including whether the detection examines the content or the URL of the webpage, uses the results of the static or dynamic crawler, and correlates multiple instances of an attack.

The top-down approach for drive-by detection adopts a crawler-scanner based architecture. For drive-by downloads, the crawler collects URLs by traversing the dynamic web graphs in the forward direction, while in parallel, a scanner identifies

Table 1: Comparison of different detection methods

Approach	Page Content	URL	static crawler	dynamic crawler	Correlation of multiple Drive-by Downloads
HoneyMonkey [91]	✓			✓	
Crawler-Based Spyware Study [11]	✓		✓	✓	
Capture-HPC [25, 27]	✓			✓	
IFrame [65]	✓			✓	
PhoneyC [41]	✓			✓	
Malicious JavaScript Detection [56]	✓			✓	✓
Blade [51]	✓			✓	
WebCop [46]		✓	✓		
ARROW		✓		✓	✓

drive-by download attempts. The scanner could be a client honeypot using signature [26] and anomaly detection [25] methods. By rendering a URL, the scanner investigates the suspicious state change of the operating system or analyzes the webpage to detect drive-by downloads. The first seven systems presented in Table 1 belong to the top-down category. The upper-left part of Figure 5 presents the architecture of *HoneyMonkey* as an example for the top-down approach. Most of the drive-by download detection approaches [11, 27, 41, 51, 56, 65, 91] fall in the top-down category. For example, a high-interaction client honeypot scanner [25] has been used to dynamically execute the webpage content [11, 27, 91]. Provos et al. [65] also adopted high-interaction client honeypots to conduct large-scale measurements of drive-by downloads in the wild. Nazario [41] proposed a lightweight, low-interaction client honeypot called *PhoneyC* to detect drive-by downloads by analyzing the webpage content. Cova et al. [56] developed a machine learning-based detector to investigate the JavaScript embedded in webpage content for drive-by download detection. Recently, Lu et al. [51] designed a detector to identify drive-by downloads by correlating user action and the binary downloading events. These approaches have shown promising results. However, their effectiveness is significantly limited by the availability of a successful response with malicious content from a drive-by download attack. The lack of a malicious response will make these approaches ineffective and thus may introduce a large number of false negatives.

To deal with the limitations introduced by analyzing webpage content, Stokes et

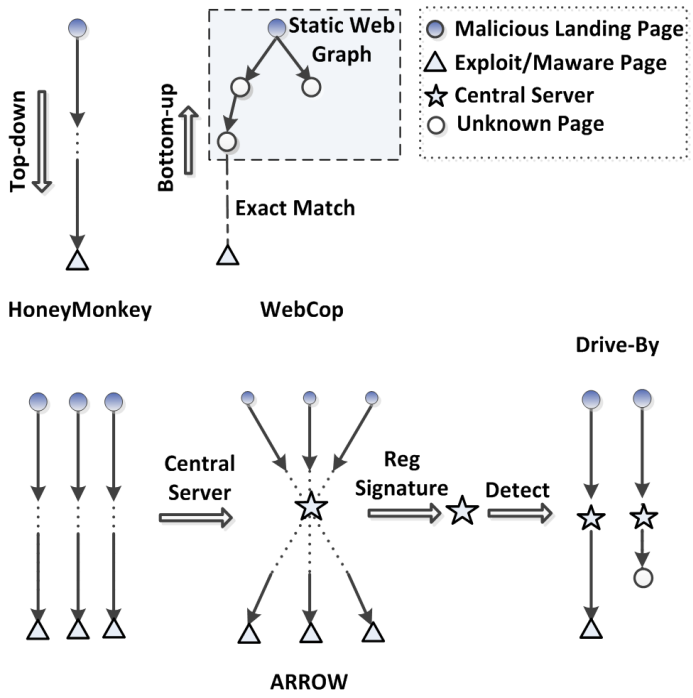


Figure 5: An illustration of existing drive-by download detection methods

al. [46] proposed a bottom-up based approach, called *WebCop*, to identify webpages that host malicious binaries. *WebCop* takes two inputs, i) a set of URLs for malware distribution sites that are contributed by a large number of anti-malware clients, and ii) a static web graph. *WebCop* traverses the hyperlinks in the web graph in a reverse direction to discover the malicious landing pages linking to the URLs for malware distribution sites. In Figure 5, the upper-middle part illustrates the architecture of *WebCop*. Nevertheless, *WebCop* has two limitations. First, *WebCop* uses an *exact* match to discover the malware distribution sites in the web graph, which may easily introduce false negatives especially when the parameter values are changed. Second, the web graph is based on static hyperlinks, which limit the detection of *WebCop*. For example, a malicious landing page may redirect the browser to the exploit server only if its dynamic content (e.g., malicious JavaScript code) is executed in the browser. Therefore, a static web graph has very limited visibility of the route from malicious landing pages to the malware distribution sites.

Provos et al. [65] proposed a method to discover malware distribution networks from drive-by download samples. This method requires the parsing and matching operation of the webpage headers (e.g., *Referer*) and content (e.g., JavaScript), which is heavyweight. Furthermore, the attackers could potentially obfuscate the webpage content to prevent their MDNs from being discovered. In contrast, *ARROW* identifies MDNs by aggregating drive-by download samples into different groups according to the malware hash values and URL information of exploit servers. Although this method provides less information of MDNs for measurement purpose (e.g., the number of redirection steps) compared to [65], it provides enough information for *ARROW* to detect central servers. In particular, this approach is more efficient and robust, which can identify more MDNs given a large number of drive-by download samples.

Automatic signature generation based on network information has been studied in previous work [42, 70, 74, 87, 92, 97] and has been used to detect various attacks. For example, most [42, 74, 87, 97] focus on *worm fingerprinting*. Perdisci et al. [70] generate signatures to detect *HTTP-based malware* (e.g., bots). *AutoRE* [92] outputs regular expression signatures for *spam detection*. Compared to these methods, *ARROW* mainly differentiates itself by detecting a different attack (a.k.a, drive-by download). Although *ARROW* uses a similar approach to build keyword-based signature trees proposed by Xie et al. [92] to detect spams, it is customized for drive-by download detection since drive-by downloads have different characteristics compared to those of spams (e.g., "distributed" and "bursty"), which are critical to guide *AutoRE* to build the signature tree.

2.3 System

The architectural overview of *ARROW* is presented in Figure 6. The input of the system is a set of HTTPTraces, which will be described in Section 2.3.1, and the output is a set of regular expression signatures identifying central servers of MDNs.

Figure 6: The system architecture

ARROW has three major processing components. The first component, Hostname-IP Mapping, discovers groups of hostnames that are closely related by sharing a large percentage of IP addresses. The second component, central server identification, aggregates individual drive-by download samples which form MDNs and then identifies the central servers. For an MDN with one or more central servers, the third component generates regular expression signatures based on the URLs and also conducts signature pruning. The following sections elaborate on each component.

2.3.1 HTTPTraces

HTTPTraces are initially collected from a cluster of high-interaction client honeypots, and an HTTPTrace example is presented in Table 2. Each honeypot visits the URL of the landing page and executes all of the dynamic content (e.g., JavaScript). By detecting state changes in the browser and the underlying operating system and file system, a honeypot can identify whether the landing page is suspicious. The suspicious landing page and other URLs consequently visited are recorded. If any exploit content is detected on a webpage, the crawler will also record its corresponding URL in "exploitURLs". Otherwise "exploitURLs" is set to be empty. Simply visiting a webpage with no user interaction should never cause an executable to be written to the file system. If the high-interaction client honeypot detects that an executable is written to disk, the file's hash value (e.g. SHA1) is stored in "bHash". The IP address corresponding to the hostname involved in each URL is also recorded. Either the non-empty "exploitURLs" or the non-empty "bHash" implies that a drive-by download attack has been *successfully launched and detected*, where "isDriveBySucc"

Table 2: An example of an HTTPTrace

HTTPTrace	
Landing Page	www.foo.com/index.php
URLs	www.bar.com/go.js www.redirect.com/rs.php www.exploit.com/hack.js www.malware.com/binary.exe
IPs	www.bar.com - 192.168.1.2 www.redirect.com - 192.168.1.3 www.exploit.com - 192.168.1.4 www.malware.com - 192.168.1.5
exploitURLs	www.exploit.com/hack.js www.malware.com/binary.exe
bHASH	4A19D50CBBBB702238....358
isDriveBySucc	True

is set as `TRUE`. Otherwise, "isDriveBySucc" is set as `FALSE`. It should be noted that "isDriveBySucc=`FALSE`" may indicate a false negative. For example, the exploit content is hosted in a compromised server and is temporally unavailable at the moment of detection. Although "isDriveBySucc" is set to be `FALSE` in this case, this exploit webpage is still considered to be harmful to other users.

2.3.2 Hostname-IP Mapping

We typically use a hostname or an IP address to represent a server. However, attackers can introduce great diversity of hostnames and IP addresses for an individual server. On one hand, IP addresses may exhibit great diversity due to fast-flux techniques [80, 85], where one hostname can be resolved to a large number of IP addresses. In this case, using the IP address to represent one server decreases the possibility of identifying central servers that share the same hostname but distribute to different IP addresses. On the other hand, attackers can register a number of hostnames (and thus domain names) and resolve them to one or a small pool of IP addresses. In this case, if we use a hostname to represent one server, we may fail to identify central servers with the same IP address but different hostnames.

In order to eliminate the diversity introduced by hostnames and IP addresses for

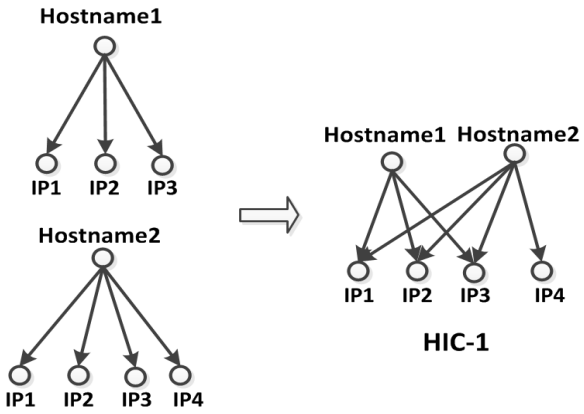


Figure 7: Hostname-IP mapping

representing a server, we design a data structure named **Hostname-IP Cluster** (HIC) to represent a group of hostnames that share a large percentage of IPs. A similar technique [69] was proposed to discover fast-flux networks. Each HIC has a set of hostnames and a set of IP addresses, denoted as $HIC = \{S_{Host}, S_{IP}\}$. We follow the steps below to discover HICs:

1 As the initialization phase, for each hostname (h_i) we have observed in the HTTPTraces, we identify all of the IP addresses resolved for h_i $(IP_1, IP_2, \ldots, IP_n)$. We initiate one HIC_i using $HIC_i.S_{Host} = \{h_i\}$ and $HIC_i.S_{IP} = \{IP_1, \ldots, IP_n\}$.

2 Suppose we have N Hostname-IP clusters, denoted as $HIC_1, \ldots, HIC_N$. For each pair of HIC_i and HIC_j, we investigate $r_{HIC} = \frac{|HIC_i.S_{IP} \cap HIC_j.S_{IP}|}{|HIC_i.S_{IP} \cup HIC_j.S_{IP}|}$, the overlap of the IP addresses of the two clusters. If $r_{HIC} > T_{HIC}$, where T_{HIC} is a pre-defined threshold and $0 \leq T_{HIC} < 1$, we first merge the second cluster into the first cluster using $HIC_i.S_{IP} = HIC_i.S_{IP} \cup HIC_j.S_{IP}$ and $HIC_i.S_{Host} = HIC_i.S_{Host} \cup HIC_j.S_{Host}$ and then discard HIC_j.

3 Repeat step 2 until no HICs are merged into other clusters.

T_{HIC} is a pre-defined threshold to determine whether two HICs should be merged or not. A small value for T_{HIC} indicates a relaxed condition. For example, $T_{HIC} = 0$

means that if two HICs share a single common IP, they are going to be merged together. This may introduce significant noise in the merged HICs. A large value of T_{HIC} enforces a strong condition for two HICs to be merged. For example, a T_{HIC} value close to 1 requires that two HICs share almost the same IPs. This may significantly decrease the possibility of merging more related HICs. In the current system, we set $T_{HIC} = 0.5$.

Figure 7 gives an example of identifying Hostname-IP clusters, where `Hostname1` and `Hostname2` share 3 out of 4 IP addresses ($r_{HIC} = 75\%$) and are grouped in one cluster. After identifying all the HICs, we use the index of each HIC to replace the hostname in each URL. For instance as described in Figure 7, the URLs of `hostname1/t.php?s=1` and `hostname2/t.php?s=2` will be represented as `HIC1/t.php?s=1` and `HIC1/t.php?s=2`. Therefore, instead of taking `hostname1` and `hostname2` as two different servers, we use the HIC to discover their relationship and represent them as a single server.

2.3.3 Identification of Central Servers

To identify the MDNs with one or more central servers, we need to first discover MDNs from a set of HTTPTraces. In *ARROW*, we use the hash value of the malware executable ("bHash") and URLs of exploit webpages ("exploitURLs") to aggregate HTTPTraces into MDNs. On one hand, we group all HTTPTraces with the same value of "bHash" into one MDN. It is possible that multiple organizations may install the same malware causing the two groups' MDNs to be merged. This should not be a major problem for the following reason. For malware that is automatically generated from a toolkit, a malicious executable is often customized for each attacker yielding different executable hash values: these similar attacks will be grouped into separate MDNs. On the other hand, for each URL in "exploitURLs", we identify its corresponding HIC and group all the HTTPTraces with same HIC index into one MDN.

For each MDN, we do not consider the landing pages as candidate URLs for discovering central servers, since they are usually compromised websites that are unlikely to serve as central servers. Also, we do not consider the URLs of "exploitURL" as candidate URLs; therefore, the identified central servers are likely used for redirection purpose.

After we aggregate all of the HTTPTraces into different MDNs, we eliminate (i.e. filter) the MDNs with a small number of HTTPTraces since small MDNs have a higher likelihood of incorrectly identifying central servers. For example, an MDN with two HTTPTraces may have a high probability of belonging to the same advertisement network, and thus the benign, advertisement server will be incorrectly identified as a central server in the MDN. Therefore $ARROW$ only identifies MDNs containing more than $T_{HTTPTrace}$ drive-by download samples where $T_{HTTPTrace}$ is currently set as $T_{HTTPTrace} = 20$.

Then for each MDN, we identify the central servers as the nodes (represented by an HIC index) that are contained in a majority of the HTTPTraces. Given an MDN with K HTTPTraces where each trace contains a collection of URLs, we have first replaced the hostname for each URL using its corresponding HIC. For each HIC, we determine the count, C, of the number of HTTPTraces employing this HIC. If $r_{cen} = \frac{C}{K}$ is greater than the pre-defined ratio T_{cen}, where $0 \leq T_{cen} \leq 1$, we take this HIC as a central server. We conservatively set T_{cen} with a large value (currently $T_{cen} = 0.9$) to guarantee the "central" property of the central server. For any two MDNs sharing one or more central servers, we merge them together into a new MDN. The shared central servers are taken as the central servers for the new MDNs, while the other central servers are discarded. This operation eliminates redundant central servers without compromising their coverage, and thus reduces the total number of signatures and consequently computationally expensive, regular expression matching operations. Also by merging smaller MDNs, we increase the

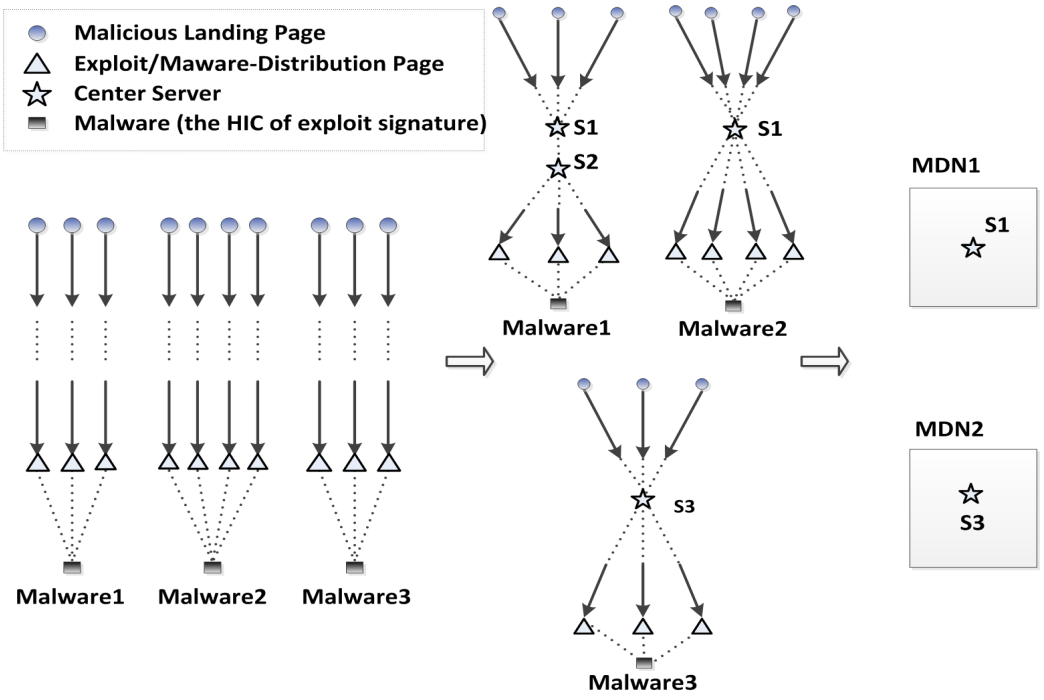


Figure 8: Discover MDNs and identify central servers

number of URLs corresponding to each central server, which helps to generate more generic signatures. Figure 8 illustrates an example of discovering MDNs and central servers. In Figure 8, S_1, S_2 and S_3 are initially identified as central servers since most of the HTTPTraces corresponding to "$Malware_{1/2/3}$" ($bHash_{1/2/3}$) contain $S_{1/2/3}$. The central server S_1, which is shared by two MDNs, is ultimately identified as the central server for the newly merged MDN. Although S_2 is discarded, S_1 still guarantees its coverage of drive-by download samples in this MDN.

2.3.4 Regular Expression Generation

In this section, we discuss how to generate regular expressions corresponding to the central servers in order to detect additional HTTPTraces exhibiting drive-by download attempts. There are two straight forward approaches to using the central server information to detect HTTPTraces of drive-by downloads. One option is to use the network level information of central servers (i.e. hostname and IP address). For example, if any URL in an HTTPTrace contains the hostname or IP address of any

central server, this HTTPTrace will be labeled as suspicious. However, this detection approach is too coarse and may introduce a large number of false positives, especially when the central server is benign such as the example described in Section 2.4.3.3. Another option is to use exact string matching of URLs in the MDNs corresponding to central servers. For example, if any URL in an HTTPTrace exactly matches any of the URLs of central servers, this HTTPTrace will be labeled as suspicious. However, this approach is too specific resulting in a huge number of false negatives. For example, a simple change in values for the parameters in the URL makes the exact match fail. These examples provide motivation to design generic signatures that can capture the invariant part of the URLs for central servers and also give an accurate description of the dynamic portion of these URLs. Thus *ARROW* generates regular expression signatures for each central server by investigating the structural patterns of its corresponding URLs.

To generate regular expressions, *ARROW* follows two steps:

1. For each central server in a MDN, generate tokens out of all the URLs that are contained in this MDN and corresponding to the central server, and then build the signature tree according to the coverage of each token.

2. Identify the branches with high coverage, and then generate signatures in the form of regular expressions.

These items are discussed in detail in the next two sections.

2.3.4.1 Token and Signature Tree Generation

Since URLs are well-structured strings, *ARROW* generates tokens based on the structure of each URL. *ARROW* collects tokens for the following 5 categories: the HIC index, the name for each directory, the name of the file, the suffix of the file and the name of the parameters. The information from the last four categories indicates the information of the toolkit that attackers use to organize the MDNs. For example, the

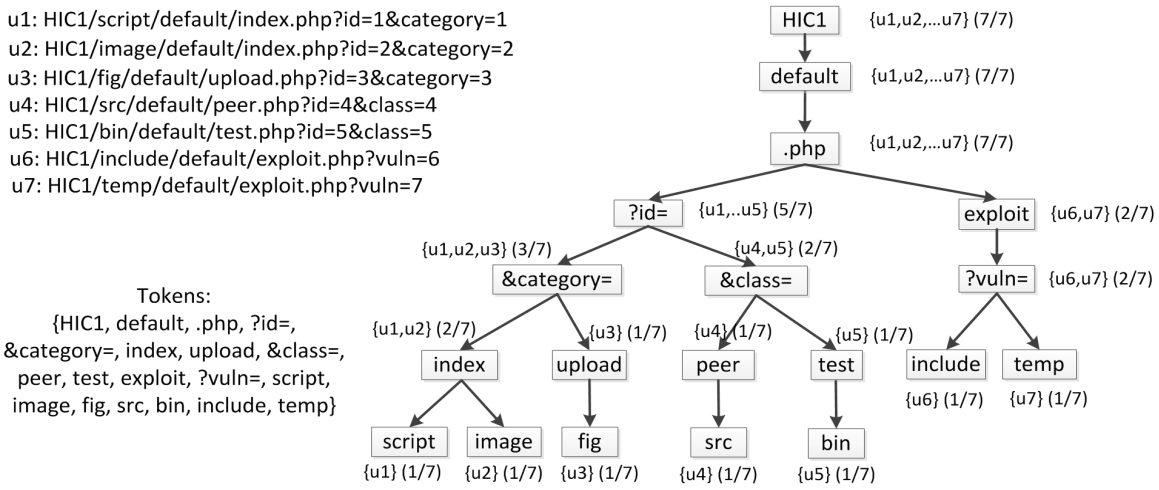


Figure 9: An example of a signature tree

name of the directory represents the directory or the sub-directory that organizes the scripts. The suffix of a script, which implies the script language, is usually identical in the same MDN for a script with the same functionality, even if the file names differ from each other. Taking the 7 URLs in Figure 9 as an example, we find the set of tokens as {`HIC1,default,index,.php,?id=,&category=,upload,&class=,peer,test,exploit,?vuln=,fig,script,image,src,bin,include,temp`}.

We build the signature tree based on the approach introduced in [92]. We denote the set of all URLs as U_{all}, and the set of URLs containing one token/node N_i as U_i^N. The set of URLs covered by a particular branch $B = \{N_{root}, N_1 \dots N_i\}$ is given by U_i^B, where $U_i^B = U_{root}^N \cap U_1^N \cdots \cap U_i^N$. To build the tree, two operations are defined: building the root node and building the child nodes. To build the root node, the token with the largest coverage of the URLs is taken as the root (N_{root}). To identify the child node(s) for one node N_i in branch $B = \{N_{root}, N_1 \dots N_i\}$, we follow the two steps below.

1. Let $U_{Rest}^B = U_i^B$ and $Set_{nodes} = \{N_{root}, N_1 \dots N_i\}$.

2. If $U_{Rest}^B = \emptyset$, exit. Otherwise, for each node $N_j \notin Set_{nodes}$, select the one with maximum $|U_{Rest}^B \cap U_j^N|$ as the child node. If $max(|U_{Rest}^B \cap U_j^N|) == 0$,

exit. Otherwise, suppose this node is N_k, then let $U^B_{Rest} = U^B_{Rest} - U^N_k$ and $Set_{nodes} = Set_{nodes} \cup N_k$. Repeat Step 2.

Returning to Figure 9, an example of the signature corresponding to the 7 URLs is presented. Since the HIC index, $HIC1$, is always contained in all of the URLs, we take it as the root of the tree.

2.3.4.2 Signature Generation

For each branch in the signature tree, we obtain more specific patterns for a subset of URLs as we approach the leaves. To learn the general pattern representing the URLs, we define a node N_i as a *critical-node* if $|U^N_i|/|U_{all}| \geq R$ and none of its child nodes satisfy this condition where the threshold $0 \leq R \leq 1$. We name the branch from the root to this critical-node as a *critical-branch*. In Algorithm 1 we describe a method to identify each critical-node. We run this algorithm multiple times by decreasing R until all the URLs are covered by the critical-nodes, which is described in Algorithm 2. In our system, we initialize $R = 0.3$ and $\alpha = 0.9$. For the signature tree described in Figure 9, we identify the critical-nodes by following the steps below.

1. $R = 0.3$. Identify critical-node "`&category=`".

2. $R = 0.3 * 0.9$. Identify critical-nodes "`&class=`" and "`?vuln=`".

After identifying the critical-nodes in the signature tree, we traverse the tree to find the branches from the root to each critical-node. The nodes in one of these branches composite one set of candidate tokens. For each token, we further investigate its average distance from the beginning of the URLs. For one set of candidate tokens, we sort the tokens according to the average distance and then obtain a sequence of candidate tokens. For the example above, $ARROW$ generates three sequences:

Algorithm 1: IdentifyCriticalNode($curNode$, R, N)

$curNode$: one Node in the tree.
R: the threshold ($0 < R < 1$).
N: $N = |U_{all}|$.
$numRest$: global variable initiated as $|U_{all}|$.

```
begin
    if curNode.isCriticalNode() then
        return;
    Boolean flag = true;
    foreach Node oneNode in curNode.getChildNodes() do
        if |U^B_oneNode| / N ≥ R then
            flag = false;
            break;
    if flag && |U^B_curNode| / N ≥ R then
        curNode.setCriticalNode();
        numRest = numRest − |U^B_curNode| ;
        return;
    else
        if curNode.isLeaf() then
            return;
        else
            foreach Node oneNode in curNode.getChildNodes() do
                IdentifyCriticalNode(oneNode, R, N);
end
```

Algorithm 2: ExploreTree(R)

R: the threshold ($0 < R < 1$).
N: $N = |U_{all}|$.
$numRest$: global variable.
α: decreasing ratio.

```
begin
    numRest = |U_all|;
    while numRest > 0 do
        IdentifyCirticalNode(rootNode, R, N);
        R = R * α;
end
```

1. seq_1 = { `HIC1,default,.php,?id=,&category=` }
2. seq_2 = { `HIC1,default,.php,?id=,&class=` }
3. seq_3 = { `HIC1,default,.php,exploit,?vuln=` }

Next, for each pair of consecutive tokens (and also for the last token and the end of URLs), we investigate the following properties of the strings that reside between them.

1. Are these strings identical?
2. The minimum and maximum length of the strings.
3. Are the characters in these strings lower or upper case?
4. Are the characters in these string letters or numbers?
5. Enumerate special characters (e.g., "." and "?") in these strings.

Finally, we summarize these properties in order to generate the regular expression. If these strings are identical, we directly present such string in the regular expression. Otherwise, we describe the properties in the regular expression format. For our running example, we obtain the three regular expressions:

1. $reg1 =$ `HIC1/[a-z]{3,5}/default/[a-z]{5,6}.php?id=[0-9]{1}&category=[0-9]{1}`
2. $reg2 =$ `HIC1/[a-z]{3,3}/default/[a-z]{4}.php?id=[0-9]{1}&class=[0-9]{1}`
3. $reg3 =$ `HIC1/[a-z]{4,7}/default/exploit.php?vuln=[0-9]{1}`

We further refer to the hostnames and IP addresses in $HIC1$. We generate the domain names for the hostnames and replace $HIC1$ using the domain names and IP addresses to get the regular expression signatures. For example, if $HIC1 = \{\{cnt1.foo1.com, cnt2.foo1.com, cnt1.foo2.com\}, \{192.168.1.2, 192.168.1.4\}\}$, we replace $HIC1$ using `foo1.com`, `foo2.com`, `192.168.1.2` and `192.168.1.4` for $reg1$, $reg2$ and $reg3$. For example, $reg3$ will be extended to be four signatures:

1. $reg3.1 =$ `foo1.com/[a-z]{4,7}/default/exploit.php?vuln=[0-9]{1}`

2. $reg3.2$ = `foo2.com/[a-z]{4,7}/default/exploit.php?vuln=[0-9]{1}`

3. $reg3.3$ = `192.168.1.2/[a-z]{4,7}/default/exploit.php?vuln=[0-9]{1}`

4. $reg3.4$ = `192.168.1.4/[a-z]{4,7}/default/exploit.php?vuln=[0-9]{1}`.

It is possible that some signatures are prone to induce false positives during detection. For example, a signature may be too general corresponding to a legitimate domain name. To decrease the possibility of false positives, we apply signature pruning. To be specific, we evaluate each signature using a large set of legitimate HTTPTraces, where each HTTPTrace is associated with a high-reputation landing page. We discard any signature successfully matching any URL in these legitimate HTTPTraces.

2.4 Evaluation

We have implemented a prototype system named *ARROW*, and evaluated it using a large volume of HTTPTraces. The HTTPTraces, described in Section 2.3 and Table 2, were collected by evaluating their landing pages using a production cluster of high-interaction client honeypots. The following sections describe the experimental setup and evaluation results.

2.4.1 Experimental Setup

Among all the HTTPTraces produced by the honeypot cluster, we randomly selected a set of 3.5 billion HTTPTraces (S_{total}) obtained from `February` and `March` of 2010. Out of S_{total}, $1,345,890$ HTTPTraces (denoted as $S_{malicious}$) are identified as drive-by download attacks. The remaining HTTPTraces are taken as the unlabeled dataset (denoted as $S_{unlabeled}$). In order to obtain benign HTTPTraces for signature pruning and evaluating false positives for popular landing pages, we use the following approach. We first collect a list of 87k URLs with high reputation scores that were recently confirmed by the analysts, where a high reputation score for a URL indicates an extremely low probability that its webpages are associated with malicious

Table 3: HTTPTraces for experiments

Trace	Number of samples
S_{total}	3,500,000,000
$S_{malicious}$	1,345,890
$S_{unlabeled}$	3,498,654,110
$S_{benign1}$	10,811,805
$S_{benign2}$	10,733,282

activities including drive-by download attacks, phishing, scamming, and spamming. We then randomly divide this list into two sublists and collect HTTPTraces whose landing pages contain one of these hostnames or URLs. The set of HTTPTraces corresponding to the first sublist (denoted as $S_{benign1}$) is used for signature pruning. The remaining HTTPTraces (denoted as $S_{benign2}$), which correspond to the second sublist, are used for false positive evaluation. Table 3 summarizes the number of HTTPTraces included in each data set described above, indicating a large-scale evaluation of the *ARROW* system.

The *ARROW* system applies regular expression signatures to match URLs in HTTPTraces. Regular expression matching is naturally computationally expensive. To speed up the matching process, we first aggregate the domain name and IP address associated with each signature into a set. For a URL, only in case its domain name or the corresponding IP address is contained (by a fast hash-based operation) in that set, signature matching is applied. The matching process is further implemented in a large-scale distributed computing infrastructure. Since the signatures will only match on rare occasion, the computational overhead to match ARROW signatures is negligible.

2.4.2 Experimental Design

We structured our experiments in four parts:

1. We compare the drive-by download attacks that are detected by *ARROW* signatures to those ($S_{malicious}$) that are detected by existing detectors. Essentially

we want to answer two questions: i) what is the coverage of ARROW signatures on drive-by download attacks that are detected by existing detectors? ii) how many *new* attacks can *ARROW* signatures detect?

2. Without knowing the ground truth for all benign HTTPTraces in our evaluation dataset, false positives generated by *ARROW* signatures become a major concern. Since it is extremely hard, if not impossible, to enumerate all benign samples among 3.5 billion HTTPTraces, we instead leverage several popular public domain/IP reputation systems to study false positives. The intuition is that if we find a domain or an IP associated with a signature is malicious, the HTTPTraces that match with this signature will be labeled as malicious.

3. We will show how many days in advance *ARROW* can detect those drive-by download attacks compared to those popular public domain/IP reputation systems.

4. We also demonstrate how *ARROW* can discover a sophisticated MDN, which leverages "twitter.com" as the central server to distribute malware, and how *ARROW* signatures detect more drive-by download attempts compared to deployed detection systems.

The following section will elaborate on the experimental results.

2.4.3 Experimental Results

We first ran *ARROW* on the $S_{malicious}$ HTTPTraces to discover Hostname-IP Clusters and identified $14,648$ $HICs$. Some hostnames show strong fast-flux patterns. For example, one HIC has only 6 hostnames but $1,041$ IP addresses, while another HIC has $34,882$ hostnames which resolve to a single IP address. The HIC structure can effectively discover and represent the relationship among such hostnames and IP addresses.

Table 4: Examples of signatures

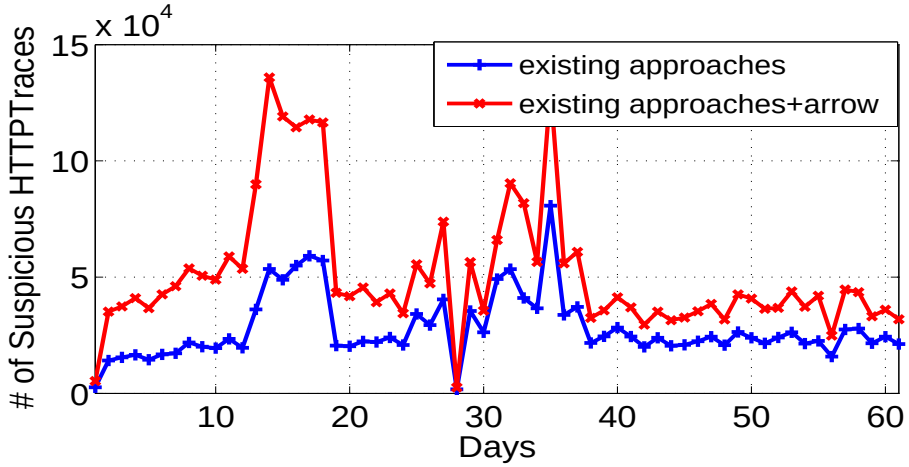
Signature
`twitter\.com\/trends\/daily\.json\?date\=2[0-9&-]{10,10}callback\=callback2`
`experimentaltraffic\.com\/cgi\-bin\/009[0-9a-zA-Z?=/.]{4,101}`
`saeghiebeesiogoh\.in\:3129\/js`
`qsfgyee\.com\:3129\/js`
`google\-analitics\.net\/ga\.js\?counter\=[0-9]{1,2}`
`servisesocks5\.com\/el\/viewforum\.php\/[0-9a-z]{32,32}\?spl\=mdac`
`chura\.pl\/rc\/pdf\.php\?spl\=pdf_ie2`
`trenz\.pl\/rc\/getexe\.php\?spl\=m[a-z_]{3,6}`

Table 5: Evaluation results

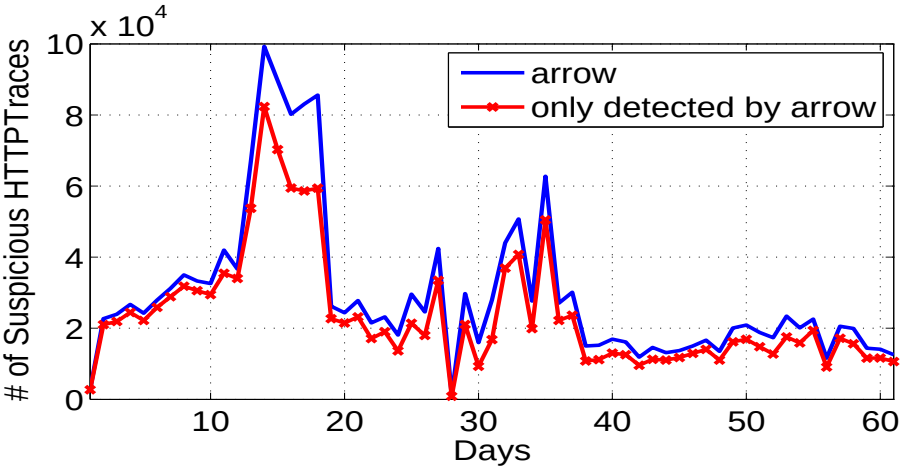
Metric	$\|H_d\|$	$\|H_a\|$	$\|H_a \cap H_d\|$	$\|H_a - H_d\|$	$\frac{\|H_a \cap H_d\|}{\|H_d\|}$	$\frac{\|H_a - H_d\|}{\|H_d\|}$
Value	1,345,890	1,612,166	320,310	1,291,856	23.8%	96.0%

After representing each server with the HIC index, $ARROW$ follows the approach described in Section 2.3.3 to identify MDNs and central servers. For the HTTPTraces in $S_{malicious}$ that are identified as drive-by download attacks by grouping them based on "bHash" or the HIC index of each URL in "exploitURLs", $ARROW$ identified $6,937$ MDNs in total and 97 MDNs (1.4%) using one or more central servers. Since each central server is represented by an Hostname-IP-Cluster (HIC), these 97 MDNs involve 104 central servers, which contain 918 hostnames and 190 IPs. By analyzing the URLs for the central servers of these 97 MDNs, $ARROW$ generated $2,592$ regular expression signatures. After pruning these signatures with $S_{benign1}$, $ARROW$ produced $2,588$ signatures including the examples presented in Table 4.

The obtained signatures could be used for detection. We apply all the signatures to the 3.5 billion HTTPTraces in S_{total}. We name H_d as the set of suspicious HTTPTraces already detected by the high-interaction client honeypots, where $H_d = S_{malicious}$. H_a denotes the set of suspicious HTTPTraces that are detected by the signatures generated by $ARROW$. Table 5 compares the result of the suspicious HTTPTraces detected by existing honeypots and signatures from $ARROW$. The column labeled $\frac{H_a \cap H_d}{H_d}$ illustrates that these central server signatures have a coverage of



(a) The deployed method v.s. ARROW



(b) ARROW

Figure 10: Detection results on a daily basis

Table 6: The first example of a central server

	Domain/IP	Confirmed by reputation systems?
Domains	mysterio.info	YES
IP	64.202.189.170	YES
	193.104.27.242	NO
	193.105.184.226	NO

Table 7: The second example of a central server

	Domain/IP	Confirmed by reputation systems?
Domains (totally 114)	lhooretb.info	YES
	pyyriwd.info	NO
	tikkiac.info	NO
	bissiqe.info	NO
	...	NO
	xassid.info	NO
	pyyriwb.info	NO
	assiqd.info	NO
	miisee.info	NO
IP	66.197.213.165	YES

23.8% of the HTTPTraces detected by existing approaches. The columns of "H_d" and "H_a" indicate that *ARROW* signatures identify more suspicious HTTPTraces. In particular, the *ARROW* signatures contribute a large number of *new* suspicious HTTPTraces (96.0%.) Figures 10(a) and 10(b) present the number of suspicious HTTPTraces detected by *ARROW* compared to the existing approach on a daily basis. Such boosted detection results demonstrate the significant advantage using *ARROW* as a parallel drive-by download detection system to existing honeypot-based detection techniques.

2.4.3.1 Evaluating False Positive Rate

In this section, we will focus on false positive evaluation. The ideal solution to evaluate false positives requires ground truth, which is referred to as all benign HTTPTraces. Given the fact that existing detection systems may introduce a huge number of false negatives, it is not a feasible approach to take those HTTPTraces, which are not

labeled as "malicious" by existing detection systems, as benign HTTPTraces. An alternative approach for acquiring ground truth is to verify each HTTPTrace by collaborating with website operators. For example, we can contact the operator who manages a certain landing page to investigate whether the HTTPTrace associated with this landing page was compromised during our detection period. Unfortunately, simply because of the scale of the dataset we studied (i.e., 3.5 billion HTTPTraces), this solution is practically impossible. In order to address this challenge, we instead leverage the maliciousness of domains or IPs involved in *ARROW* signatures. It is a practically feasible approach since several popular domain/IP reputation systems, which can suggest the maliciousness of a certain domain or IP, are publicly available, such as `support.clean-mx.de` and `www.malwaredomains.com`. Specifically, if the domain or the IP address in a signature is confirmed as "malicious", we will take HTTPTraces detected by this signature as malicious.

However, these reputation systems may suffer from limited visibility of whole malicious network infrastructure. Consequently, they may miss many malicious domains and IPs that are detected by *ARROW*. Table 6 and Table 7 present two examples to illustrate this problem. For instance, a center server contains one domain ("mysterio.info") and three IP addresses, among which only the domain and one IP are confirmed as "malicious" by the reputation systems. However, a manual investigation using search engine of the remaining two IPs revealed their great suspiciousness. For the second central server, 114 IP addresses are resolved to one single IP address, and only one domain and the IP address are verified as "malicious" by the reputation systems. However, the remaining domains appear to be generated by certain random domain generation algorithm [54], which is commonly used by attackers. In fact, many of these domains were actually confirmed to be malicious based on manual search. Therefore, apart from only focusing on the reputation of domains and IPs,

we consider the reputation of a central server, which is presented by an Hostname-IP-Cluster (see Section 2.3.2), a group of domains and IPs sharing the same network infrastructure. Intuitively, more domains or IPs in a central server are malicious, the higher probability their corresponding network infrastructure is used for malicious purpose. In our evaluation, we follow a very conservative way: we label a central server as "suspicious" if either *all* domains or *all* IPs in this central server are verified as "malicious" based on the domain/IP reputation systems. Given a "suspicious" central server, we label its remaining domains and IPs, which have not been verified as "malicious" yet, as "suspicious". Returning to the central server in Table 6, 100% of its domains , "mysterio.info", is labeled as "malicious", and hence we take this central server as "suspicious". As a result, "193.104.27.242" and "193.105.184.226" will be labelled as "suspicious".

Following aforementioned approaches, we have verified totally 506 (55%) out of 918 domains and 190 IPs as "malicious" using several popular domain/IP reputation systems. Further, we labeled 97 out of 104 central servers as "suspicious", which result in another 396 (43%) suspicious domains and IPs. Finally, 16 (2%) domains and IPs are neither "malicious" nor "suspicious", and we label them as "potentially benign". The first row in Table 8 presents the percentage of domains/IPs labelled as "malicious", "suspicious", and "potentially benign", respectively. The second row in the same table present the percentage of *newly detected* HTTPTraces that can match with those signatures, whose domains/IPs are labeled as "malicious", "suspicious", and "potentially benign", respectively. According to the table, we can find that the vast majority (92.4%) of newly detected drive-by attacks are associated with *malicious* domains/IPs and a significant percentage (7.4%) of them are associated with *suspicious* domains/IPs, leaving a tiny portion of 0.2% (2,600 HTTPTraces) as potential false positives.

We also match these signatures against all URLs in $S_{benign2}$. Out of total $10,733,282$

Table 8: The percentage of domains/IPs and HTTPTraces verified

	malicious	suspicious	potentially benign
Domains/IPs	55%	43%	2%
HTTPTraces	92.4%	7.4%	0.2%

Table 9: The percentage of domains/IPs and HTTPTraces verified for each period

$\leq$ Jan 2010	Feb & March 2010	$\geq$ April 2010
34.5%	56.7%	8.8%

legitimate HTTPTraces, only 2 HTTPTraces are matched with the signatures. This indicates that *ARROW* signatures introduce negligible low false positive rate for those landing pages with high reputation (e.g., landing pages that are popular and well-maintained).

To conclude, *ARROW* signatures yield 2,600 potential false positives in S_{total} (3.5 billion HTTPTraces) and 2 false positives in $S_{benign2}$ (11 million HTTPTraces). Such experimental results demonstrate an extremely low false positive rate of $7.4 * 10^{-5}\%$ (2602 out of around 3.5 billion HTTPTraces).

2.4.3.2 Early Detection

While the vast majority (92.4%) of newly detected are finally verified by public domain/IP reputation systems, *ARROW* signatures show significant advantage by achieving early detection. On the one hand, among 506 domains and IPs that are verified by the reputation systems, we successfully acquired exposure dates for 468 domains and IPs (corresponding to 89% drive-by download attacks). On the other hand, since our dataset were obtained in Feb and March 2010, we get *ARROW* signatures at the last day of March 2010. Then we can evaluate how many days in advance we can use *ARROW* signatures to detect drive-by download attacks compared to the knowledge acquired from popular public domain/IP reputation systems. As presented in Table 9, 34.5% drive-by download attacks could be detected using reputation systems before Feb 2010, and 56.7% can be detected during Feb and March 2010. However,

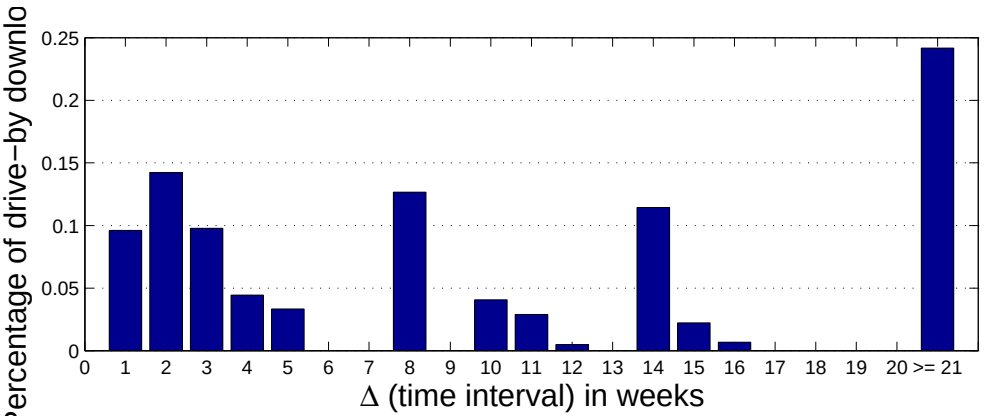


Figure 11: Δ (time interval) in weeks

8.8% drive-by download attacks (93,488 attacks) could not be detected based on reputation systems before the last day of March 2010, at which time *ARROW* signatures are generated for detection. For each of these 93,488 HTTPTraces, we calculate Δ as the time interval between April 1, 2010 (the detection time for *ARROW*) and the public exposure date for a domain or an IP in the used reputation systems. Figure 11 presents the distribution of Δ, which demonstrates that *ARROW* signatures can detect these drive-by download attacks much earlier compared to those reputation systems. The average Δ is 172 days.

2.4.3.3 Case Study for the Twitter Signature

Out of the signatures generated by *ARROW*, "twitter.com" appears in one signature in Table 4. This signature identified a large number of 135,875 suspicious HTTPTraces as described in column "H_a" in Table 10, contributing 8.4% ($\frac{135875}{1612166}$) of all detected suspicious HTTPTraces. Since "twitter.com" is a well known website used for social networking and microblogging, false positives are a concern. In this section, we conduct a detailed analysis of all HTTPTraces matched by the "twitter.com" signature to assess whether or not it is a false positive.

First, we briefly describe why twitter was involved in a large number of drive-by download attacks. Manual analysis reveals that a suspicious webpage retrieves the

Table 10: Evaluation results for the "twitter" signature

Metric	$\lvert H_a \rvert$	$\lvert H_d \rvert$	$\frac{\lvert H_a - H_d \rvert}{\lvert H_d \rvert}$	$\lvert H_k \rvert$	$ReflectRate(H_k, H_a)$
Value	135,875	60,159	125.9%	119,774	99.6%

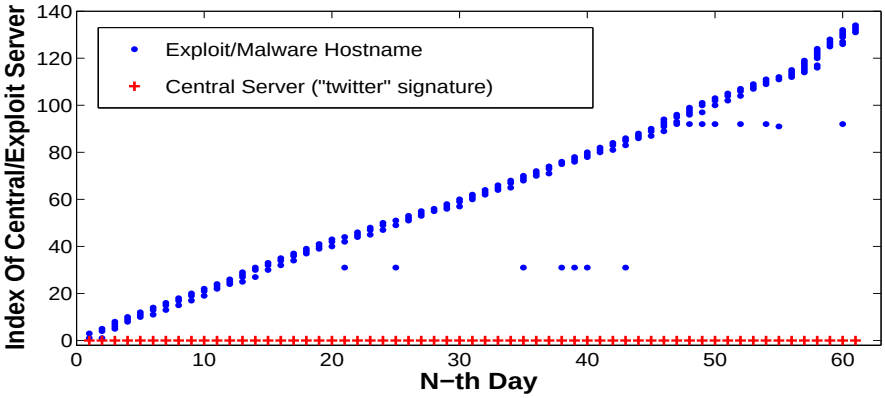


Figure 12: Active days for the central server and exploit servers (60 days)

week's top trending terms using Twitter's API, dynamically constructs a hostname from these changing terms, and then instructs the browser to retrieve the exploit from the server corresponding to that hostname. A detailed description of the script that performs these actions can be obtained from a website dedicated to raising awareness of online threats [6].

Figure 12 and its zoom-in view Figure 13 present the temporal patterns of the MDN identified by the "twitter.com" central server. The X-axis illustrates the days the hostname appears in our collected HTTPTraces over time, whereas the Y-axis represents the index into the set of dynamically generated hostnames based on Twitter's API. As the graph illustrates, the hostname is switched on a regular basis representing a strong *fast-flux* pattern, while the central server ("twitter.com") remains *stable*. This architecture introduces a great challenge for the detection techniques that identify the server responsible for hosting the actual exploit. In this MDN, this server changes every few days. However, the signature generated by *ARROW* captures the central server, which is the most stable point over time. Therefore, signatures generated by *ARROW* can detect a large number of the suspicious HTTPTraces, even if

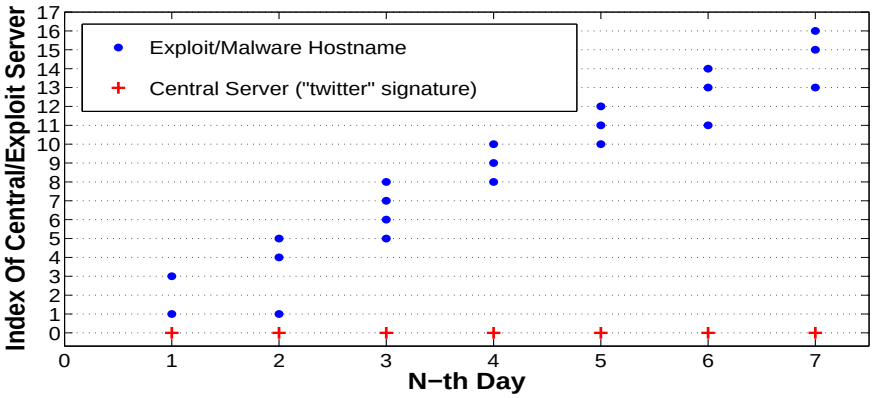


Figure 13: Active days for the central server and exploit servers (zoom-in view for first 7 days)

the corresponding server hosting the exploit changes or is temporarily unavailable.

To assess the false positive rate of the signature, we need to further categorize the matched HTTPTraces. The column H_a is described previously. In addition, 60,159 HTTPTraces, denoted as H_d in Table 10, in the H_a traces have been identified as drive-by downloads by the existing approach. These HTTPTraces in H_a can be classified as follows into three categories:

1. The client honeypot retrieves the exploit from the aforementioned server and is compromised. The client honeypot can successfully detect the drive-by downloads in this scenario.

2. The client honeypot makes a request to the exploit server but fails to retrieve the exploit.

3. The client honeypot visits "twitter.com", but no further connection attempts are made to the exploit server.

Manual analysis reveals that an HTTP request to the server hosting the exploit contains specific keywords as part of the URL's path. These keywords are listed in Table 11 and would be an indication of malicious intent (categories 1 and 2 above).

In total, 119,774 or 88.15% of HTTPTraces match these keywords as described in column "H_k" in Table 10 leaving a large portion of 11.85% as potential false positives. Since no subsequent requests are made after "twitter.com" has been contacted for category 3, it is challenging to obtain evidence on whether these HTTPTraces also have malicious intent. However, if we assume that a compromised server hosts a large number of similar malicious webpages, we can further assess the malicious intent of these HTTPTraces. We do so with a function called *ReflectRate*, which expresses the rate of suspicious HTTPTraces based on that assumption.

Table 11: Keywords for detection

keywords	"*/tre/*", "*/nte/*", "*/ld/*", "*.exe*"

The *ReflectRate* is calculated as follows. We use ht to represent an HTTPTrace and H to denote a set of HTTPTraces ($H = \{ht_1, ht_2, \ldots, ht_n\}$). We further use the functions $LP()$ to return the landing page for an HTTPTrace and $Host()$ to obtain the hostname of a URL. Given H, we introduce a function $HostSet()$ to indicate the set of hostnames in the landing pages, $HostSet(H) = \cup_{i=1}^{n}\{Host(LP(ht_i))\}$. We then define a function $Reflect(H_1, H_2)$ to return a set of HTTPTraces $H_r = Reflect(H_1, H_2)$, where $H_r = \{ht_i | ht_i \in H_2, Host(LP(ht_i)) \in HostSet(H_1)\}$. H_r represents a subset of HTTPTraces in H_2, whose landing pages are hosted on servers (e.g., $HostSet(H_1)$) that have been confirmed to serve malicious webpage content. Based on these functions, we define $ReflectRate(H_1, H_2) = \frac{|H_r|}{|H_2|}$. For example, suppose $H_1 = \{ht_1, ht_2\}$ is a set of suspicious HTTPTraces detected by existing methods, where $LP(ht_1) =$ `www.foo.com/index.php` and $LP(ht_2) =$ `www.bar.com/index.php`. We also find $HostSet(H_1) =$ { `www.foo.com`, `www.bar.com`}. Assume we are also presented with a set of unlabeled HTTPTraces $H_2 = \{ht_a, ht_b, ht_c, ht_d\}$, where $LP(ht_a) =$ `www.foo.com/contact.php`, $LP(ht_b) =$ `www.foo.com/test.php`, $LP(ht_c) =$ `www.bar.com/temp.php` and $LP(ht_d) =$ `www.test.com/index.php`. We can classify three out of four HTTPTraces in H_2 as suspicious, which is $Reflect(H_1, H_2) = \{ht_a, ht_b, ht_c\}$, since

$Host(LP(ht_{a/b/c})) \in HostSet(H_1)$. The resulting $ReflectRate(H_1, H_2)$ would be 75%.

Applied to our case study of "twitter.com", we obtain a *ReflectRate* of 99.6% leaving 0.4% or 544 of the identified traces as potential false positives as shown in the last column of Table 10.

2.5 Discussion

Our evaluation has shown that the *ARROW* system successfully boosts detection of suspicious pages with an overall low false positive rate of $7.4 * 10^{-5}$%. The ability to run *ARROW* in production may be compromised if the signatures cause popular websites to be detected as malicious (i.e. false positives). Strategies to reduce the likelihood of this happening, such as white listing or clustering based on page characteristics, are left for future work. Signature pruning is based on a large data set of benign HTTPTraces. Collecting a comprehensive dataset of definite benign traces, however, is very hard in practice. One could collect URLs corresponding to the most popular queries in the search engine logs or URLs that are most popular in the ISP or enterprise networks with the assumption being that popular webpages are less likely to host drive-by downloads. Nevertheless, as attackers usually compromise legitimate sites to hijack its traffic as an entry point into their MDN, this assumption may not always hold true. It may result in some signatures being pruned that shouldn't have been and some HTTPTraces being incorrectly marked as false positives. However, given that malicious webpages are overall quite rare among popular webpages (e.g., 0.6% of the top million URLs in Google's search results led to malicious activity according to [65]), the vast majority of HTTPTraces collected in such a way will indeed be benign and can be used for pruning and evaluation.

Evasion may be another concern with our system. Similar to other drive-by download detection systems, by knowing our detection algorithm, attackers can always

carefully redesign the attack strategy to evade our detection. For example, the attackers can generate a different binary for each time of an attack, which will in consequence cause different binary hash values and thus prevent *ARROW* from aggregating them into the same MDN. To deal with this problem, more information, such as the similarity of different binaries or behavior characteristics [55, 70, 86], can be adopted to aggregate polymorphic malware into MDNs. The attackers can also decentralize the MDNs to eliminate the central servers or hide the MDN structure from the client (e.g. through server side includes instead of redirect chains.) As the MDNs identified by *ARROW* do not provide comprehensive coverage on all HTTPTraces (with current coverage 23.8%) identified by traditional methods, it is an indication that some of these evasion techniques are used today. These are accepted shortcomings of our approach, and we will look towards refining our existing algorithm and exploring new approaches to detect MDNs as part of our future work.

2.6 Summary

Drive-by download attacks have become the primary approach used for malware infection. Therefore, detecting drive-by download attacks is an extremely important task since it can help us detect bots in the infection phase, thereby preventing bots from performing malicious activities in their early stage. We have shown that by aggregating data from a large number of drive-by download attempts, we are able to discover both simple malware distribution networks as well as more complex MDNs including one or more central servers. We have conservatively estimated that 1.4% (97 out of 6,937) of MDNs employ central servers. While this percentage is currently small, the overall coverage of these complex MDNs is 23.8% (320,310 out of 1,345,890 malicious traces) which is reasonably large considering the small number of actual MDNs with central servers. Going forward, we expect the number of attackers employing sophisticated methods to manage their operations to grow.

The major hurdle for deploying *ARROW* to detect MDNs utilizing central servers and block all landing pages which redirect to these servers is developing ways to accurately identify false positives. This problem is generic to any method attempting to solve this problem and is not a reflection of the proposed system. We have shown that the regular expression signatures have a very low false positive rate of $7.4*10^{-5}\%$. However, it is not practical to employ an army of analysts to investigate all signatures generated by the system, particularly given the highly dynamic ecosystem being used by attackers today. Developing better, automated methods of assessing the purpose of these central servers in the absence of a successful attack needs to be a focus of future research. Without access to this ideal system in the near term, it may be prudent to restrict internet content from users (i.e. not display the webpages from a search query or serve ads which redirect to the central server) in order to err on the side of caution. While this may potentially penalize legitimate content providers initially as the system is deployed, having a method for an individual organization to understand the underlying cause of any true false positives and a method to quickly rectify any errors will help balance the competing objectives of providing end users, the potential bot-infection victims, with the widest range of content while keeping them safe from harm.

CHAPTER III

DETECTING PEER-TO-PEER BOTNETS

3.1 Motivation

In Chapter 2, we focus on detecting botnets in the infection phase. While detecting botnets in the infection phase is of great importance, it is not sufficient. The major reason is that certain infection activities may fly under the radar of deployed network-based detection systems. For example, a computer could be infected by a bot binary from a physically connected USB stick, thereby not arousing any malicious network traffic. As a consequence, detecting botnets in the control phase becomes indispensable, which is the focus of this chapter.

In the control phase, botmasters rely on C&C channels to control and coordinate bots. Botmasters could build the C&Cs with different structures, such as *centralized* structures and *peer-to-peer* (P2P) structures. In a centralized structure, all bots in a botnet contact one (or a few) C&C server(s) *owned* by the botmaster. Centralized C&C channels based on the `IRC` or `HTTP` protocol have been used by many botnets due to their simplicity and availability of open-source, reusable C&C server code. However, centralized C&C servers represent a *single point of failure*. Therefore, attackers have recently started to build botnets with a more resilient C&C architecture, using a peer-to-peer (P2P) structure [76, 66, 67] or hybrid P2P/centralized C&C structures [37]. Bots belonging to a P2P botnet (i.e., a botnet that uses P2P-based C&C communications) form an overlay network in which any of the nodes (i.e., any of the bots) can be used by the botmaster to distribute commands to the other peers or collect information from them. While more complex, and perhaps more costly to manage compared to centralized botnets, P2P botnets offer higher resiliency, since

even if a significant portion of a P2P botnet is taken down (e.g., by law enforcement or network operators) the remaining bots may still be able to communicate with each other and with the botmaster. Notable examples of P2P botnets are represented by `Nugache` [68], `Storm` [66], `Waledac` [37], and even `Confiker`, which has been shown to embed P2P capabilities [67]. `Storm` and `Waledac` are of particular interest because they use P2P C&C structures as the primary way to organize their bots, and have demonstrated resilience to take-down attempts.

To date, a few approaches for detecting P2P botnet have been proposed [36, 72, 79]. BotMiner [36] finds groups of hosts within a monitored network that share similar communication patterns with outside machines and at the same time perform similar malicious activities, such as scanning, spamming, launching remote exploits, etc. If such groups of hosts exist, they are considered to be part of a botnet and an alarm is raised. The intuition is that bots belonging to the same botnet will share similar C&C communication patterns, and will respond to the botmaster's commands with similar malicious activities. Unfortunately, modern botnets are using more and more stealthy ways to perform malicious activities. For example, some botnets may send spam through large popular webmail services such as `Gmail` or `Hotmail` [94]. Such activities are very hard to detect through network flow analysis, due to encryption and overlap with legitimate webmail usage patterns, thus making BotMiner ineffective. BotGrep [72] analyzes network flows collected over multiple large networks (e.g., ISP networks), and attempts to detect P2P botnets by analyzing the communication graph formed by overlay networks. Starting from a global view of Internet traffic, BotGrep first identifies groups of hosts that form a P2P network. To further differentiate P2P botnets from the legitimate P2P networks (e.g., P2P file sharing networks), BotGrep requires additional information to bootstrap its detection algorithm. For example, BotGrep may use a list of nodes in a communication (sub-)graph that are related to *honeypot* hosts, or may leverage the detection results from

intrusion detection systems. However, acquiring both a sufficiently global view of Internet communications and enough *a priori* information to bootstrap the detection algorithm may be very challenging and makes the detection results (which in [94] were mainly based on simulations) heavily dependent on other systems, thus limiting the real-world applicability of BotGrep. Recently, Yen et al. [79] have proposed an algorithm that aims to distinguish between hosts that run legitimate P2P file sharing applications and P2P bots. However, the proposed algorithm [79] does not take into account the fact that some popular legitimate P2P applications may not exhibit network patterns typical of P2P file sharing applications. For example, `Skype`, a very popular P2P-based instant messenger, does not usually behave in a way similar to file sharing applications. For example, large file transfers through Skype are usually rare, compared to its use as an instant messenger or voice-over-IP (VoIP) client. Therefore, Skype's P2P traffic may cause a significant number of false positives. Moreover, the algorithm in [79] is not able to detect bot-compromised hosts that exhibit mixed legitimate and botnet-related P2P traffic (e.g., due to users running a file sharing P2P application on machines compromised with P2P bots).

In this chapter, we present a novel botnet detection system that is able to identify *stealthy* P2P botnets, whose attacks are very hard to be observed in the network traffic. Our approach identifies P2P bots within a monitored network by detecting the C&C communication patterns that characterize P2P botnets, regardless of how they perform malicious activities in response to the botmaster's commands. To accomplish this task, we first identify all hosts within a monitored network that appear to be engaging in P2P communications. Then, we derive *statistical fingerprints* of the P2P communications generated by these hosts, and leverage the obtained fingerprints to distinguish between hosts that are part of legitimate P2P networks (e.g., file-sharing networks) and P2P bots. Unlike previous work, our system is able to identify stealthy P2P bots within a monitored network even when the P2P botnet traffic is overlapped

with traffic generated by legitimate P2P applications (e.g., Skype) running on the same compromised host.

To summarize, our work makes the following contributions:

1. A new *flow-clustering*-based analysis approach to identify hosts that are most likely running P2P applications, and estimate the *active time* of the detected P2P nodes.

2. An efficient algorithm for P2P traffic *fingerprinting*, which we use to build a statistical profile of different P2P applications.

3. A P2P botnet detection system that can effectively and accurately detect P2P bots, even when they perform malicious activities in a stealthy, non-observable way. In addition, our system is able to identify bot-compromised machines, even when the P2P botnet traffic is overlapped with traffic generated by legitimate P2P applications (e.g., Skype) running on the same compromised machine.

4. A scalable system design based on computationally efficient detection algorithm and load-balance workload distribution.

5. An implementation of our detection system and an extensive experimental evaluations. Our experimental results show that i) the system can detect P2P bots with a detection rate of 100% and 0.2% false positive rate, ii) the system achieves great scalability, processing more than 80 million flows within 1 hour.

3.2 Related Work

As P2P botnets become robust infrastructures for various malicious activities, they have attracted a lot of efforts from researchers [76, 66, 81, 67, 20]; the most notable and studied P2P botnets are `Nugache` [76], `Storm` [66, 81], `Waledac` [37], and `Confiker` [67]. A few approaches have been proposed that can be used for P2P botnet

detection [36, 72, 79], which have been discussed in Section 3.1. `BotHunter` [35] was proposed to detect a bot, centralized or P2P, in its *infection phase* if the infection behaviors are consistent with the infection model used by `BotHunter`. However, bots now use a wide variety of approaches for infection (e.g., drive-by downloads), which may not be consistent with `BotHunter`'s infection model.

Our work focuses on the detection of P2P botnets using network information. Compared with the existing approaches, the design goals of our approach are different in that: 1) our approach does not assume that malicious activities are observable, unlike [36]; 2) our approach does not require any botnet-specific information to make the detection, unlike [72]; and 3) our approach aims to detect the compromised hosts that run both P2P bot and other legitimate P2P applications at the same time, unlike [79]. To achieve these design goals, our system includes multiple components. The first one is a *flow-clustering*-based analysis approach to identify hosts that are mostly likely running P2P applications. In contrast to existing approaches of identifying hosts running P2P applications [83, 73, 82, 15, 60], our approach differs from them in the following ways: 1) unlike [73], our approach does not need any content signature because encryption will immediately make content signature useless; 2) our approach does not rely on any transport layer heuristics (e.g., fixed source port) used by [82, 83], which can be easily violated by P2P applications; 3) we do not need training data set to build a machine learning based model as used in [15], because it is very challenging to get traffic of P2P botnets before they are detected; 4) in contrast to [60], our approach can detect and profile various P2P applications rather than identifying a specific P2P application (e.g., `Bittorrent`); and 5) our analysis approach can estimate the active time of a P2P application, which is critical for botnet detection.

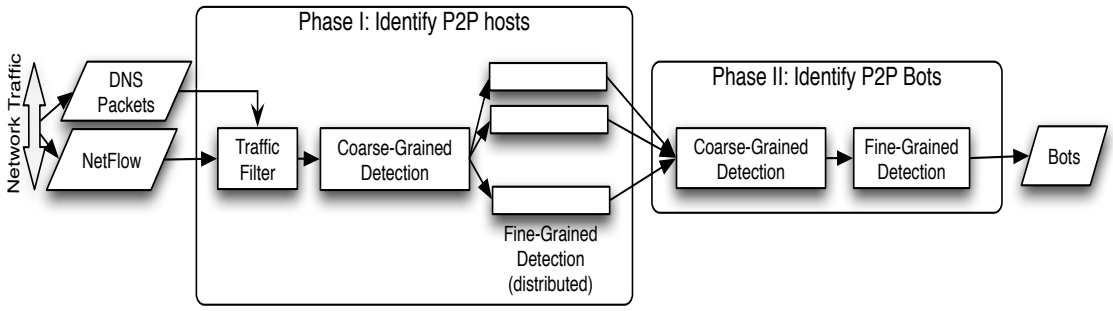


Figure 14: System overview

3.3 System

3.3.1 Problem Formulation

Our goal is to monitor the network traffic at the edge of a network (e.g., an enterprise network), and identify whether any of the machines within the network perimeter has become part of a P2P botnet. In particular, we consider the scenario in which bots perform malicious activities in a stealthy way, for example spam-bots that send spam through stolen or malicious web-mail accounts (e.g., `Gmail` or `Hotmail` accounts) [94], whose malicious activities are very hard to detect from traffic analysis. In general, we assume that the bots' malicious activities may not be easily observable, and therefore we only focus on their C&C communication patterns. We assume that at least two or more machines within the monitored network are part of the same P2P botnet, and leverage the similarity in communication patterns across multiple bots for detection purposes.

3.3.2 System Overview

P2P-based botnets rely on a P2P protocol to establish a C&C channel and communicate with the botmaster. As such, we intuitively assume that P2P bots exhibit some network traffic patterns that are common to other P2P client applications (either legitimate or malicious). Therefore, we divide our systems into two phases.

Phase I: At the first phase, we aim at detecting all hosts within the monitored network that appear to be engaging in P2P communications, as shown in Figure 14. We analyze raw traffic collected at the edge of the monitored network (e.g., an enterprise network), and apply a pre-filtering step (discussed in Section 3.3.3) to reduce the data volume and only consider network flows that are potentially related to P2P communications. Then, we analyze the remaining traffic and extract a number of statistical features (described in Section 3.3.7.1), which we use to isolate flows related to P2P communications from unrelated flows, and identify *candidate* P2P clients.

Phase II: In the second phase, our botnet detection system (detailed in Section 3.3.6) analyzes the traffic generated by the candidate P2P clients and classifies them into either *legitimate* P2P clients or P2P *bots*. The architecture of our botnet detection system is based on a number of observations.

1. First, bots are malicious programs used to perform profitable malicious activities. They represent valuable assets for the botmaster, who will intuitively try to maximize their *utilization*. As a consequence, bot programs usually make themselves persistent on the compromised system and run as long as the system is powered on. This is particularly true for P2P bots, because in order to have a functional overlay network (the botnet), a sufficient number of peers needs to be always online. In other words, the active time of a bot should be comparable with the active time of the underlying compromised system. If this was not the case, the botnet overlay network would risk to *degenerate* into a number of disconnected subnetworks, due to the short life time of each single node. On the other hand, the active time of legitimate P2P applications is determined by users. For example, some users tend to use their file-sharing P2P clients only to download a limited number of files, before shutting down the P2P application [30]. In this case, the active time of the legitimate P2P

application may be much shorter compared to the active time of the underlying system. Based on this observation, our botnet detection system first estimates the active time of a P2P client and eliminates those hosts that are running P2P applications with short active time, compared to the underlying system. It is worth noting that some users may run certain legitimate P2P applications for as long as their machine is on. For example, `Skype` is a popular P2P application for instant messaging and voice-over-IP (VoIP) that is often setup to start after system boot, and that keeps running until the system is turned off. Therefore, such `Skype` clients (or other "persistent" P2P clients) will not be filtered out at this stage.

2. In order to discriminate between legitimate *persistent* P2P clients and P2P bots, we make use of the following observations: 1) bots that belong to the same botnet use the same P2P protocol and network, and 2) the set of peers contacted by two different bots have a much larger overlap, compared to peers contacted by two P2P clients connected to the same legitimate P2P network. While the first observation is obvious, the second observation deserves explanation. Assume that two hosts in the monitored network, say h_A and h_B, are running the same legitimate P2P file-sharing application (e.g., Emule). The users of these two P2P clients will most likely have uncorrelated usage patterns. Namely, it is reasonable to assume that in the general case the two users will search for and download different content (e.g., different media files or documents) from the P2P network. This translates into a *divergence* between the set of IP addresses contacted by hosts h_A and h_B (remember that at this stage we are only considering the P2P traffic generated by the hosts). The reason is that the two P2P clients will tend to exchange P2P control messages (e.g., ping/pong and search requests) with different sets of peers which "own" the content requested by their users, or peers that are *along the path* towards the

content. On the contrary, if h_A and h_B are compromised with P2P bots, one of the characteristics of the bots is that they need to periodically *search for commands* published by the botmaster [81]. This typically translates into a *convergence* between the set of IPs contacted by h_A and h_B.

To summarize, in order to detect P2P bots we follow the high-level steps reported below:

1. Identify the set $\mathbf{H}$ of all hosts engaged in P2P communications.
2. Identify the subset $\mathbf{P} \subseteq \mathbf{H}$ of P2P clients whose active time is similar to the active time of the underlying systems.
3. Identify the subset $\mathbf{B} \subseteq \mathbf{P}$ which exhibit similar P2P communication patterns, and have a significant overlap of the set of contacted peers. We classify the hosts in set $\mathbf{B}$ as P2P bots.

To illustrate the statistical features and motivate the related thresholds used by our system, we used five popular P2P applications (see Table 12) for 24 hours to collect their traffic traces. For the Bittorrent application, we generated two separate 24-hour traces (`T-Bittorrent` and `T-Bittorrent-2`). In this section we report a number of measurements on the obtained traffic traces to better motivate some of our design choices. Table 14 reports the feature values measured on the collected traffic traces. The notation used for our statistical features is summarized in Table 13.

We now describe the components of our detection system in more details.

3.3.3 Traffic Volume Reduction

As a first step, we want to filter out network traffic (and their sources) that is unlikely to be related to P2P communications. This is accomplished in part by passively analyzing DNS traffic, and identifying network flows whose destination IP addresses were previously *resolved* in DNS responses. The reason is that P2P clients usually

Table 12: P2P applications

P2P Apps	Version	Protocol
Bittorrent	6.4	Bittorrent
Emule	0.49c	Kademlia
Limewire	5.4.8	Gnutella&Bittorrent
Skype	4.2	Skype
Ares	2.1.5	Gnutella&Bittorrent

Table 13: Notations and descriptions

notation	Description
T_{p2p}	the active time of P2P application
N_f	the number of failed connections per hour
No-DNS Peers	the percentage of flows associated with no domain names
N_{clust}	the number of clusters left by enforcing Θ_{bgp} and Θ_{p2p}
N_{bgp}	the largest number of unique bgp prefixes in one cluster
$\hat{T_{p2p}}$	the estimated active time for P2P application

Table 14: Measurement of features

Trace	T_{p2p}	N_f	*No-DNS Peers*	N_{clust}	N_{bgp}	$\hat{T_{p2p}}$
T-Bittorrent	24hr	1602	96.85%	17	12857	24hr
T-Emule	24hr	318	99.99%	8	1133	24hr
T-Limewire	24hr	1278	99.97%	36	5661	24hr
T-Skype	24hr	81	99.93%	12	12806	24hr
T-Ares	24hr	489	99.99%	16	1596	24hr

contact their peers directly, by looking up IPs from a routing table for the overlay network, rather than resolving a domain name (a possible exception may be when a peer bootstraps into a P2P network by looking up domain names that resolve to stable *super-nodes*). This observation is supported by Table 14 (*No-DNS Peers*), which illustrates the percentage of flows whose destination IP addresses were not resolved from a domain name. It confirms that the vast majority of flows generated by P2P applications do not have destination IPs resolved from domain names. The remaining small fraction of flows are either related to bootstrapping (e.g., in the case of `bittorrent.com` and `skype.com`) or for downloading advertisement content from popular websites. Since most non-P2P applications (e.g., browsers, email clients, etc.) often connect to a destination address resulting from domain name resolution, this simple filter can eliminate a very large percentage of non-P2P traffic (see Section 3.4) while retaining the vast majority of P2P communications.

3.3.4 Identifying P2P Clients

After traffic volume reduction we consider the remaining traffic, and for each host h within the monitored network we identify three flow sets (we call "outgoing" those flows that have been initiated by h):

1. $S_{tcp}(h)$: flows related to successful outgoing TCP connections.

2. $S_{udp}(h)$: flows related to successful outgoing UDP (virtual) connections.

3. $S_o(h)$: flows related to failed outgoing TCP/UDP connections.

We consider as successful those TCP connections with a completed `SYN, SYN/ACK, ACK` handshake, and those UDP (virtual) connections for which there was at least one "request" packet and a consequent response packet.

P2P applications act as both clients and servers. A node in a P2P network can initiate (TCP or UDP virtual) connections to its peers and accept connections initiated by other peers. In client-mode, P2P nodes periodically probe their peers with `ping/pong` messages to maintain a view of the overlay network (usually for routing purposes), or search for content. A consequence of this behavior is the fact that P2P nodes will often generate a large number of failed outgoing flows. The reason is that P2P networks are usually characterized by a significant node churn [30], due to previous nodes that leave the network and new nodes that join it (the churn is intuitively correlated with users that turn on or off their P2P applications or machines). Therefore, a node that sends a ping message to a known peer will often discover that the peer is not up anymore (no pong is received, thus causing a failed connection).

At this point, we retain all hosts that generated at least a successful outgoing TCP or UDP connection, and that generated more than a predefined number Θ_o of outgoing failed TCP/UDP connections. Namely, we retain a host h if $|S_{tcp}(h)| + |S_{udp}(h)| > 0$ AND $S_o(h) > \Theta_o$, and discard all other hosts (it is worth noting that here we are only considering those flows that passed the DNS-based traffic volume reduction filter described in Section 3.3.3). Table 14, reports the number N_f of failed outgoing connections per hour for different P2P applications. We can see that P2P applications typically generate a large number (from several tens up to thousands) of failed connection attempts with other peers. Therefore, we conservatively set $\Theta_o = 10$.

What we just described is a "*Coarse-Grained Detection of P2P Clients*" that allows us to focus on *candidate* P2P nodes. We further apply a "*Fine-Grained Detection of P2P Clients*" to prune away hosts that are not actual P2P nodes. For example, we want to eliminate hosts that made it into the list of *candidate* P2P nodes by chance (e.g., because of scanning behavior). To this end, we first consider the fact that each node of a P2P network frequently exchanges a number of control messages (e.g., ping/pong messages) with other peers. Also, we notice that the characteristics of

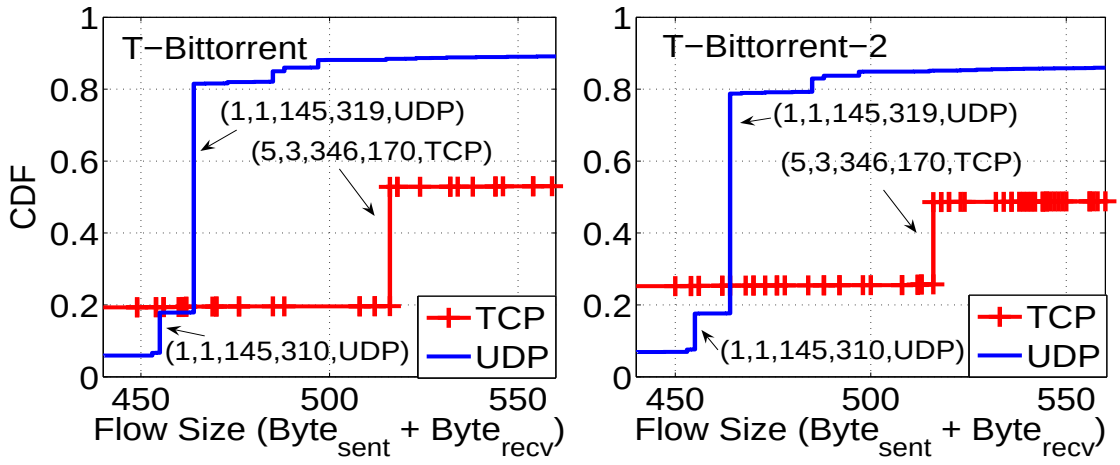


Figure 15: CDF of flow sizes

these messages, such as the size and frequency of the exchanged packets, are *similar* for nodes in the same P2P network, and *vary* depending on the P2P protocol and network in use. In addition, we notice that a node will often exchange control messages with a relatively large number of peers distributed in many different networks, where each network can be represented by its `BGP` prefix. Figure 15 describes the distribution of flow sizes for two `Bittorrent` traces, where a large number of flows share similar sizes.

To identify flows corresponding to P2P control messages, we first apply a flow clustering process intended to group together similar flows for each candidate P2P node h. Given sets of flows $S_{tcp}(h)$ and $S_{udp}(h)$, we characterize each flow using a vector of statistical features $v(h) = [Pkt_s, Pkt_r, Byte_s, Byte_r]$, in which Pkt_s and Pkt_r represent the number of packets sent and received, and $Byte_s$ and $Byte_r$ represent the number of bytes sent and received, respectively. We then apply a clustering algorithm (described in Section 3.3.7.1) to partition the set of vectors (i.e., of flows) $V_{tcp}(h) = \{v(h)_i\}_{i=1..|S_{tcp}(h)|}$ and $V_{udp}(h) = \{v(h)_i\}_{i=1..|S_{udp}(h)|}$ into a number of clusters. Each of the obtained clusters of flows, $C_j(h)$, represents a group of flows with similar size. For each $C_j(h)$ (notice that each vector can be mapped back to the flow it describes), we consider the set of destination IP addresses related to the flows

in the clusters, and for each of these IPs we consider its BGP prefix (using BGP prefix announcements). Finally, we count the number of distinct BGP prefixes related to destination IPs in a cluster $bgp_j = BGP(C_j(h))$, and discard those clusters of flows for which $bgp_j < \Theta_{bgp}$. We call *fingerprint clusters* the remaining cluster of flows. Therefore, each host h can now be described by a set of fingerprint clusters $FC(h) = \{FC_1, .., FC_k\}$. We label h as P2P node if $FC(h) \neq \emptyset$, namely if h generated at least one fingerprint cluster.

We applied clustering-based flow analysis to the sample traces of 5 P2P clients. N_{bgp} in Table 14 presents the maximum number of distinct BGP prefixes of destination IPs in a fingerprint cluster. We therefore conservatively set the threshold $\Theta_{bgp} = 50$, which is much smaller than the measured N_{bgp}.

Figure 16 illustrates an example of the flow clustering process for a P2P node. Flows corresponding to ping/pong and peer-discovery share similar sizes respectively, and therefore they are grouped into two clusters (FC_1 and FC_2). Since the number of destination BGP prefixes involved in each cluster is larger than Θ_{bgp}, we take FC_1 and FC_2 as its fingerprint clusters. A *fingerprint cluster summary*, ($\overline{Pkt_s}, \overline{Pkt_r}, \overline{Byte_s}, \overline{Byte_r}$, proto), presents the protocol and the average number of sent/received packets/bytes for all the flows in this fingerprint cluster. Examples of fingerprint cluster summaries for two Bittorrent traces, including T-Bittorrent and T-Bittorrent-2, and one Skype trace, are illustrated in Table 15. "(1 1 145 319, UDP)" and "(1 1 109 100, UDP)" are shared by both Bittorrent sample traces. The payload of flows corresponding to these two fingerprint clusters are illustrated in Table 16. It reveals that the fingerprint cluster of "(1 1 145 319, UDP)" represents the flows for node discovery, and that of "(1 1 109 100, UDP)" contains the flows for ping/pong.

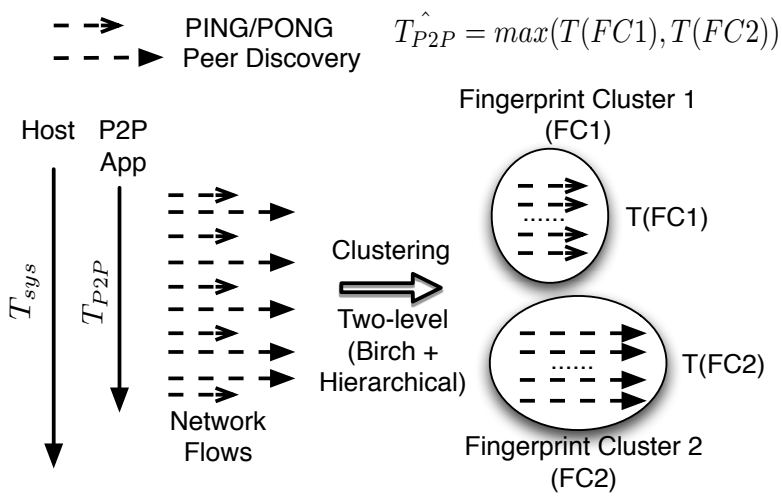


Figure 16: Example of identify P2P hosts based on flow-clustering analysis

3.3.5 Identifying Persistent P2P Clients

As we mentioned at the beginning of Section 3.3, P2P bots make themselves persistent into the compromised system, and run for as long as the system is powered on. Based on this observation, the fist component in the "Phase II" of our system ("*Coarse-Grained Detection of P2P Bots*") aims at identifying P2P clients that are active for a time T_{P2P} close to the active time T_{sys} of the underlying system they are running on. While this behavior *is not unique* of P2P bots and may be representative of other P2P applications (e.g., Skype clients that run for as long as a machine is on), identifying persistent P2P clients *takes us one step closer* to identifying P2P bots.

To estimate T_{sys} we proceed as follows. For each host $h \in \mathbf{H}$ that we identified as P2P clients according to Section 3.3.7.1, we consider the timestamp $t_{start}(h)$ of the first network flow we observed from h and the timestamp $t_{end}(h)$ related to the last flow we have seen from h. Afterwards, we divide the time $t_{end}(h) - t_{start}(h)$ into w epochs (e.g., of one hour each), denoted as $T = [t_1, ..t_i, .., t_w]$. We further compute a vector $A(h, T) = [a_1, ..a_i, .., a_w]$ where a_i is equal to 1 if h generated any network traffic between t_{i-1} and t_i. We then estimate the active time of h as $T_{sys} = \sum_{i=1}^{w} a_i$.

The challenge is how to accurately estimate the active time of a P2P application.

Since a P2P application periodically exchanges network control (e.g., ping/pong) messages with other peers as long as the P2P application is active, we can leverage the active time of a fingerprint cluster, which represents flows of control messages, in order to estimate the active time of the corresponding P2P application. For each host h (again, we consider only the hosts in $\mathbf{H}$, which we previously identified as P2P clients) we consider the set of its fingerprint clusters $FC(h) = \{FC_1, ..FC_j.., FC_k\}$ (see Section 2.3), and for each fingerprint cluster FC_j we compute a vector $A(FC_j, T) = [a_1^j, ..a_i^j, .., a_w^j]$ where an element a_i^j is equal to 1 if the fingerprint cluster FC_j contains a flow between t_{i-1} and t_i, otherwise $a_i^j = 0$. We compute the active time of a fingerprint cluster FC_j as $T(FC_j) = \sum_{i=1}^{w} a_i^j$. Finally, we estimate the active time (T_{P2P}) of a P2P application as $\hat{T_{P2P}} = \max(T(FC_1), ..T(FC_j), ..T(FC_k))$.

If the ratio $r(h) = \frac{\hat{T_{P2P}}}{T_{sys}} > \Theta_{P2P}$, we say that h is running a persistent $P2P$ application, and add it to a set $\mathbf{P}$ of *candidate* P2P bots. Host h will then be input to our botnet detection algorithm (see Section 3.3.6), where h will be represented by a set of persistent fingerprint clusters for h, denoted as $FC_p(h) = \{FC_i^1, .., FC_k^j\}$ where $T(FC_i)/T_{sys} > \Theta_{P2P}$ for any $FC_i \in FC_p(h)$.

As illustrated in Table 14, the estimated active time $\hat{T_{P2P}}$ is the same as the actual active time (T_{P2P}) of the P2P application, which demonstrates that $\hat{T_{P2P}}$ can accurately approximate T_{P2P}. As we can see from Table 14, when we leave a P2P application running for as long as the machine is on (24 hours for this particular experiment) we obtain a ratio $r(h) = 1$. Therefore, we decided to conservatively set $\Theta_{P2P} = 0.5$. N_{clust} in Table 14 illustrates the size of $FC_p(h)$, the number of fingerprint clusters (FCs) whose $BGP(FC) > \Theta_{bgp}$ and $T(FC) > \Theta_{p2p}$.

3.3.6 P2P Botnet Detection Algorithm

Once we have identified the set $\mathbf{P}$ of candidate P2P bots, we apply our botnet detection algorithm, which has been implemented in the second component in the "Phase

Table 15: Examples of fingerprint cluster summaries

Trace	Fingerprints
T-Bittorrent	1 1 145 319, UDP 1 1 109 100, UDP 1 1 146 340, UDP 5 3 346 170, TCP 1 1 145 310, UDP
T-Bittorrent-2	1 1 145.01 317.66, UDP 1 1 109 100, UDP 1 1 146 342, UDP 5 3 346 170, TCP 2 2 466 461, UDP

Trace	Fingerprints
Skype	1 1 74.58 60, UDP 1 1 78 60, UDP 1 1 75 60, UDP 1 1 76 60, UDP 1 1 79 60, UDP

Table 16: Payload of flows in a fingerprint cluster of a Bittorrent application

Fingerprints	flows	outgoing content	incoming content	description
1 1 145 319, UDP	1	d1:ad2:...find_node1:...:y1:qe	d1:rd2:...nodes208:...:y1:re	peer discovery
	2	d1:ad2:...find_node1:...:y1:qe	d1:rd2:...nodes208:...:y1:re	
	...	d1:ad2:...find_node1:...:y1:qe	d1:rd2:...nodes208:...:y1:re	
1 1 109 100, UDP	1	d1:ad2:...:ping1:...:y1:qe	d1:rd2:...:y1:re	ping/pong
	2	d1:ad2:...:ping1:...:y1:qe	d1:rd2:...:y1:re	
	...	d1:ad2:...:ping1:...:y1:qe	d1:rd2:...:y1:re	

II" of our system ("*Fine-Grained Detection of P2P Bots*"). At this stage, our objective is to differentiate between legitimate persistent P2P clients and P2P bots. As we mentioned at the beginning of Section 2.3, our detection approach is based on the following observations: i) bots that belong to the same botnet use the same P2P protocol and network, and ii) the set of peers contacted by two different bots have a large overlap, compared to peers contacted by two P2P clients connected to the same legitimate P2P network. Accordingly, we look for P2P clients that are running the same protocol and connect to the same P2P network, and whose sets of contacted destination IPs overlap significantly. We do so by introducing a measure of similarity between the fingerprint clusters, and then grouping P2P clients according to similarities between their respective *fingerprint clusters*.

We proceed as follows. For each host $h \in \mathbf{P}$, we consider the set of persistent fingerprint clusters $FC_p(h) = \{FC_1, .., FC_k\}$ (see Section 3.3.7.1). For each $FC_i \in FC_p(h)$, we compute the average number of bytes sent, $\overline{Byte}_{s,i}$, and received, $\overline{Byte}_{r,i}$, in all flows in FC_i (remember that each fingerprint cluster FC_i is a cluster of flows).

Also, for each cluster FC_i we extract the set of peers Π_i, i.e., the set of all destination IPs for the flows in FC_i. Therefore, each fingerprint cluster FC_i can be summarized by the tuple ($\overline{Byte}_{s,i}$, $\overline{Byte}_{r,i}$, Π_i). This allows us to define a notion of distance between fingerprint clusters. In practice, we define two separate distance functions as follows

i) $d_{bytes}(FC_i, FC_j) = \sqrt{(\overline{Byte}_{s,i} - \overline{Byte}_{s,j})^2 + (\overline{Byte}_{r,i} - \overline{Byte}_{r,j})^2}$

ii) $d_{IPs}(FC_i, FC_j) = 1 - \frac{|\Pi_i \cap \Pi_j|}{|\Pi_i \cup \Pi_j|}$

and then we define the distance between two hosts h_a and h_b as

$$dist(h_a, h_b) = \min_{i,j}\Big(\lambda * \frac{d_{bytes}(FC_i^{(a)}, FC_j^{(b)}) - min_B}{max_B - min_B} + (1-\lambda) * d_{IPs}(FC_i^{(a)}, FC_j^{(b)})\Big)$$

where

- $FC_k^{(x)}$ is the k-th fingerprint cluster of host h_x
- $min_B = \min_{i,j} d_{bytes}(FC_i^{(a)}, FC_j^{(b)})$
- $max_B = \max_{i,j} d_{bytes}(FC_i^{(a)}, FC_j^{(b)})$
- λ is a predefined constants, which we set to $\lambda = 0.5$.

After computing the distance between each pair of hosts (i.e., each pair of candidate P2P bots in set $\mathbf{P}$), we apply hierarchical clustering, and group together hosts according to the distance defined above. In practice, the hierarchical clustering algorithm will produce a dendrogram (a tree-like data structure) as shown in Figure 19. The dendrogram expresses the "relationship" between hosts. The closer two hosts are, the lower level they are connected at in the dendrogram. Two P2P bots in the same botnet should have small distance and thus are connected at lower level (forming a *dense* cluster). Even if these P2P bots' traffic is overlapped with traffic of legitimate P2P applications, the distance between two bot-compromised hosts is decided by the

minimum distance of their respective *fingerprint clusters*. Since the distances of fingerprint clusters from botnet P2P protocols have smaller distance compared to those from legitimate P2P protocols (due to bots' large overlap of peer IPs), the minimum distance will stem from fingerprint clusters of P2P bots instead of legitimate P2P applications. Therefore, two bot-compromised hosts running legitimate P2P applications will still exhibit small distance. We then classify hosts in *dense* clusters as P2P bots, and discard all other clusters and the related hosts, which we classify as legitimate P2P clients. In practice, we cut the dendrogram at Θ_{bot} ($\Theta_{bot} \in [0, 1]$) of the maximum dendrogram height ($\Theta_{bot} * height_{max}$).

To set Θ_{bot}, we assume that: a) we do not have a labeled data set of botnet traffic; b) the distance between two legitimate P2P applications is much larger than that between two bots belonging to the same botnet (as motivated above). Therefore, we conservatively set $\Theta_{bot} = 0.95$.

3.3.7 Scalability Optimization

System scalability becomes a serious concern as the traffic volume increases. Therefore, we need to evaluate and improve the scalability of proposed system to enable its deployment in high-speed and high-volume networks. In this section, we first identify the performance bottleneck of the proposed system and then present our design to improve system performance. In the following discussion, we will use n to represent the number of flows generated by a host and N to represent the number of hosts in the monitored network. Out of 5 system components, "traffic filter" can be implemented with $O(n)$ operation, where flows only need to be scanned once to decide whether their IPs are matched with any DNS responses. The "*Coarse-Grained Detection of P2P Clients*" and "*Coarse-Grained Detection of P2P Bots*" share the similar performance, where both of them scan the flows once to obtain the number of failed connections, and the active time for P2P applications and hosts. The remaining two components,

"*Fine-Grained Detection of P2P Clients*" and "*Fine-Grained P2P Detection of P2P Bots*", involve clustering operation (a.k.a., pairwise comparison). The "*Fine-Grained P2P Bot Detection*" compares the similarity of two *persistent P2P clients*. Given the small percentage of potential P2P clients in the ISP network (e.g., 3%-13% as reported in [32]) and even smaller portion of *persistent P2P clients*, this component is unlikely to introduce performance bottleneck. For example, given a typical /16 ISP network with 65,536 hosts, if we assume that 8% hosts use P2P clients and further assume that half of them are persistent, then this component will only needs to process 2,221 hosts. The "*Fine-Grained P2P Client Detection*" component, however, may introduce significant performance overhead. It calculates the pairwise distance of *all* successful outgoing flows from each host, and conduct such calculation for *all* hosts that are identified in the previous component. Since the number of queries initiated by a host could be huge and a large number of hosts still remain after the "*Coarse-Grained P2P Client Detection*" component (e.g., around 30% in our evaluation), this component will likely have the computational complexity of $O(N * n^2)$, which could be prohibitively high. Therefore, the performance bottleneck is the component of "*Fine-Grained P2P Client Detection*".

We follow two directions to eliminate the performance bottleneck. First, we improve the performance of the clustering operation, which groups the similar flows into *a single host*. Second, we distribute the flow-clustering workload for *all* the hosts into a set of computation nodes (e.g., the Map-Reduce infrastructure) to parallelize the computation.

3.3.7.1 Clustering Performance Improvement

Instead of directly using the clustering algorithm with $O(n^2)$ computational complexity, we design a two-level clustering scheme to improve the performance of clustering

operation. First, we use BIRCH [84], a streaming clustering algorithm with time complexity $O(n)$, to efficiently identify at most Cnt_{birch} sub-clusters from the sets of TCP and UDP flows respectively. Cnt_{birch} is a pre-defined parameter and the distance of two flows is defined as the Euclidean distance of $[Pkt_s, Pkt_r, Byte_s, Byte_r]$. Second, for each sub-cluster, we aggregate flows in it and represent it using a vector, where this vector describes the average value of each feature $[\overline{Pkt_s}, \overline{Pkt_r}, \overline{Byte_s}, \overline{Byte_r}]$ of flows in this sub-cluster. We further apply *hierarchical clustering* with *DaviesBouldin* validation index [58] on top of the vectors (sub-clusters), and find clusters of vectors, where each cluster represents a set of similar vectors (sub-clusters). Since we obtain at most Cnt_{birch} sub-clusters from the first-level clustering, the computational complexity for the hierarchical clustering is bounded by $O(Cnt^2_{birch})$. For all the sub-clusters belonging to a cluster, we finally group the flows in these sub-clusters to the same cluster of flows. For this two-level clustering scheme, the computational complexity to process the flows of one P2P node is mainly bounded by $O(n+Cnt^2_{birch})$. Currently we configure $Cnt_{birch} = 4000$ (the evaluation of system performance over Cnt_{birch} is presented in Section 3.4.3.5).

3.3.7.2 Load-Balance Workload Partition

Since the flow analysis based on two-level clustering is independent for each host, we can distribute the analysis workload to a set of computation nodes, where each node only needs to perform clustering-based flow analysis for a subset of hosts. The problem can be formulized as follows: given N hosts denoted as $H = \{h_1, h_2, ...h_N\}$ and M computation nodes denoted as $C = \{c_1, c_2, ...c_M\}$, how we can partition H into M exclusive subsets $HS_1, HS_2..HS_M$ so that the computation time, which is the maximum execution time for each node $T = max(exc(c_i, HS_i))$, is minimized. $exc(c_i, HS_i)$ returns the execution time for HS_i on c_i. Theoretically, given the assumption that each computation node has the same capacity, T is minimized when

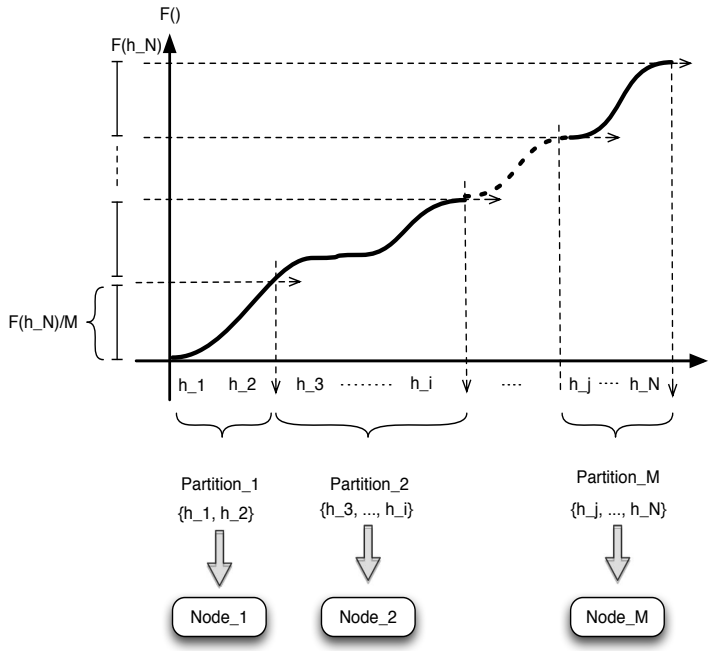


Figure 17: Partition hosts to M distributed nodes

$exc(c_i, HS_i)$ is equal to each other. Therefore, our goal is to evenly partition the workload of N hosts to M computation nodes.

To achieve an even distribution of the workload, we need a function that estimates the workload (e.g., execution time) for each host, denoted as $f(h_i)$. However, it does not exist an explicit expression describing the relationship of execution time and the volume of flows for a host. Therefore, we adopt an empirical approach to derive this function. As each host h_i initiates two sets of successful outgoing TCP and UDP flows ($S_{tcp}(h_i)$ and $S_{udp}(h_i)$), we first introduce a function $g(h_i)$ that returns the number of flows from h_i, where $g(h_i) = |S_{tcp}(h_i)| + |S_{udp}(h_i)|$. Second, we randomly sample a small number of K hosts (e.g., $K = 10$) from H, denoted as $U = \{h_i^1, h_j^2, \ldots, h_n^K\}$, and get a set of flow-set sizes denoted as $V = \{v_1, v_2, \ldots, v_K\} = \{g(h_i^1), g(h_j^2), \ldots, g(h_n^K)\}$. We further apply the two-level flow-clustering analysis on each host in U, and obtain their execution time, denoted as $T = \{t_1, t_2, \ldots, t_K\}$. Given $V = \{v_1, v_2, \ldots, v_K\}$ and $T = \{t_1, t_2, \ldots, t_K\}$, we can finally use regression method to learn a function

$\bar{t_i} = \phi(v_i)$ that estimates the execution time from the volume of flows. Since the two-level flow-clustering algorithm has less than $O(n^2)$ time complexity, we use the linear regression model. By incorporating function $g()$, we can get the expression that estimates the execution time $\bar{t_i}$ given an host h_i, where $\bar{t_i} = f(h_i) = \phi(g(h_i))$.

After getting $f(h_i)$, we introduce a cumulative function $F(h_i) = \sum_{w=1}^{w=i} f(h_w)$. We assign h_i to the dth ($d = 1, 2...M$) computation node if $(d-1) * \frac{F(h_N)}{M} < F(h_i) \leq d * \frac{F(h_N)}{M}$. Figure 17 illustrates this procedure.

By following these two approaches, we successfully eliminate the performance bottleneck by reducing the computational complexity from $O(N * n^2)$ to $O(\frac{N}{M} * (n + Cnt_{birch}^2))$.

3.4 Evaluation

In this section we present an evaluation of our detection system.

3.4.1 Experimental Setup

We evaluated the performance of our detection system using real-world network traffic, including traffic collected from our academic network, traffic generated by popular P2P applications, and *live* P2P botnet traffic.

The traffic we collected from our academic network came from a span port mirroring all traffic crossing the gateway router (around 200-300Mbps) for the college networks. We used `Argus` [1] to efficiently collect network flow information of the traffic between internal and external networks for one entire day. Along with various flow statistics we also recorded the first 200 bytes of each *flow payload*, which we used to identify known legitimate P2P clients within our network. To reduce the volume and noise in our network traces, we excluded all traffic related to email servers, DNS servers, and `planetlab` nodes from our botnet detection analysis. The DNS traffic was collected simultaneously with the network flow information, using `dnscap`, to

keep track of all the domain-to-IP mappings needed to perform traffic volume reduction. Overall, we observed 953 active hosts, as reported in Table 17. We refer to the traffic collected from our academic network as NET_{CoC}.

In order to establish some ground truth in terms of what hosts are running P2P applications, we used a signature-based approach that matches the signatures from [95] onto the first 200 bytes of each network flow. We further manually investigated each of these hosts to eliminate false positives (we found some spurious signature matches deriving from traffic towards SMTP servers that we were not able to pre-filter, and a few web requests towards our departmental website). After manual validation, we identified a total of 3 hosts that were running `Bittorrent` , which in the following we denoted as "BT1@C", "BT2@C" and "BT3@C". Furthermore, there exists no signature that can match P2P traffic generated by `Skype`, since `Skype` communications are encrypted. However, using the statistical traffic fingerprints, we were able to identify 5 likely `Skype` clients within our network (we discuss this in more detail in Section 3.4.3.1), denoted as "Skype1@C", "Skype2@C", .., "Skype5@C". We refer to the network traces corresponding to these 8 P2P clients as $NET_{P2P@CoC}$. One possible reason why we found only a few (fewer than expected) P2P hosts is that our college network is well-managed and the usage of file sharing applications is highly discouraged. In addition, the vast majority of the hosts we have monitored are desktops managed by the college, where regular users have no permission to install software including `Skype`.

In order to increase the number and diversity of P2P nodes in our network, we ran 5 popular P2P applications, whose name and version are listed in Table 12. We ran each of the 5 P2P applications in two different (virtual) hosts for several hours (e.g., 24 or 5 hours) *simultaneously*. Each host was represented by a WindowsXP (virtual) machine with a public IP address selected within a /24 network. Given a P2P application among the 5 we considered, we manually interacted with one instance

Table 17: Statistics of network traffic in our academic network

Trace	duration	# of TCP / UDP flows	# of clients
t-c	24h	61,745,989 / 20,226,837	953
Trace	duration	# of domains	# of IPs
t-dns	24h	328,965	268,753

Table 18: Traces of popular P2P applications

Trace	Dur	# of flows	# of Dst IPs	Avg Flow Size
Bittorrent-1/2	24 hr	250960/297785	17337/17657	68310/350205
Limewire-1/2	24 hr	229215/638103	11602/64994	1003/2038
Emule-1/2	24 hr	58941/110821	6649/14554	124267/22681
Skype-1/2	24 hr	88927/49541	10699/6264	514/1988
Ares-1/2	5 hr	17566/21756	1918/3118	69373/24755

(on one host) to simulate typical human-driven application usage behavior, and we fed the second instance of the application (on the second host) with automatically generated user-interface input. This artificial user input was simulated using an `AutoIt` [2] script that randomly selects contents to be downloaded or uploaded using the P2P application at random time intervals. Therefore, overall we obtained 10 additional network traces related to traffic generated by P2P applications (Table 18 shows some statistics related to these network traces). We refer to these network traces as NET_{P2P}.

In addition, we were able to obtain network traces for two popular P2P botnets, `Storm` and `Waledac`. Both traces were collected by purposely running `Storm` and `Waledac` malware samples in a controlled environment, and recording their network behavior. The `Storm` traces included 13 different bot-compromised hosts, while the `Waledac` included 3 different bot-compromised hosts, as shown in Table 19. It is worth noting that both traces were collected at a time when the two botnets were fully active, before any successful takedown attempt was carried out by law enforcement or network operators. We refer to these network traces as NET_{bots}.

Table 19: Traces of botnets

Trace	duration	size	# of bots
Waledac	24hr	1.1G	3
Storm	24hr	4.8G	13

3.4.2 Experimental Design

We structured our experiments in five parts:

1. Evaluate the effectiveness of identifying and profiling P2P applications using *statistical* fingerprint clusters. (see Section 3.4.3.1)

2. Evaluate the detection performance by pretending that a number of machines in our network have been compromised with either `Storm` or `Waledac` (Section 3.4.3.2).

3. Determine whether our system is able to detect P2P bots running on compromised machines that are also running legitimate P2P clients at the same time (Section 3.4.3.3).

4. Estimate the detection performance in special cases, where only two bots or no bot (e.g., a "clean" network) appear in the monitored networks (Section 3.4.3.4).

5. Analyze the system scalability (Section 3.4.3.5).

6. Analyze the effect of system parameter Θ_{bot} (Section 3.4.3.6).

We prepared four data sets for evaluation, D_1, D_2, D_1' and D_2'. We obtained D_1 as follows: For each host (denoted as h_{p2p}) of both 16 P2P bots (in NET_{bots}) and 10 P2P applications (in NET_{P2P}), we randomly selected one host (denoted as h_{CoC}) from trace NET_{CoC}, and we overlaid h_{p2p}'s traffic to the h_{CoC}'s traffic. We aligned the start time of the h_{CoC}'s traffic according to the start time of its corresponding h_{p2p}'s traffic. If the duration of h_{CoC}'s traffic was t_h and that of h_{p2p} was t_p, where $t_h > t_p$, we only kept the first t_p of h_{CoC}'s traffic. In effect, we simulated the scenario

Table 20: Bot traces overlaid with P2P application traces

Bot	P2P App	Before Overlaying (Bot)			After Overlaying (Bot+P2PApp)		
		# of flows	# of DstIPs	avg flow size	# of flows	# of DstIPs	avg flow size
Waledac1	Emule1	341784	850	12829	452645	15338	55688
Waledac2	BT2@C	319119	760	11372	361135	1359	348708
Storm1	Limewire1	200237	6390	1342	429458	16635	1714
Storm2	BT3@C	275451	7319	1337	310667	8307	3381
Storm3	Bittorrent2	133955	5584	1344	432464	23261	172945
Storm4	Skype4@C	171471	7277	1280	199101	7520	1266
Storm5	Skype1	164917	6686	1328	214548	13137	1307
Storm6	Ares1	220459	6618	1307	238063	8543	6244

where the P2P bots/applications are running persistently in the underlying hosts. D_1 represents the scenario that a host is compromised by a P2P bot and some legitimate P2P applications are active in the same monitored network.

For D_2, we randomly selected half (8) of the P2P bots from NET_{bots}. Then for each of the 5 P2P applications we ran, we randomly selected one out of its two traces from NET_{P2P} and overlaid its traffic to the traffic of a randomly selected host from NET_{CoC}. We further randomly chose 3 P2P hosts from $NET_{P2P@CoC}$ identified in the first experiment (Section 3.4.3.1). We finally overlaid each of 8 P2P bot traces to each of the selected 8 P2P traces (5 from NET_{P2P} and 3 from $NET_{P2P@CoC}$), as illustrated in the first two columns in Table 20. D_2 represents the scenario that a host, which is compromised by a P2P bot, has an active legitimate P2P application running at the same time.

We use D_1' to represent a "clean" network, where no host is compromised by P2P bots. We got D_1' by simply removing all the hosts overlaid with bots' traces from NET_{bots}. In order to get D_2', we randomly selected hosts compromised by two bots for each botnet from D_2 and discarded the rest of the hosts overlaid by the traces from NET_{bots}. So D_2' represents the scenario in which only two bots from each botnet exist in the monitored network.

3.4.3 Experimental Results

Table 21 summarizes the experimental results in Section 3.4.3.2, 3.4.3.3 and 3.4.3.4, where we set the parameters as $\Theta_{bot} = 0.95$ and $Cnt_{birch} = 4000$. The effect of varying

Table 21: Experimental results

	TP	FP	Data	Description
1	100%	0.2%	D_1	bots overlaid with host
2	100%	0.2%	D_2	bots overlaid with P2P host
3	100%	0.2%	D_2'	only two bots
4	–	0.2%	D_1'	a "clean" network

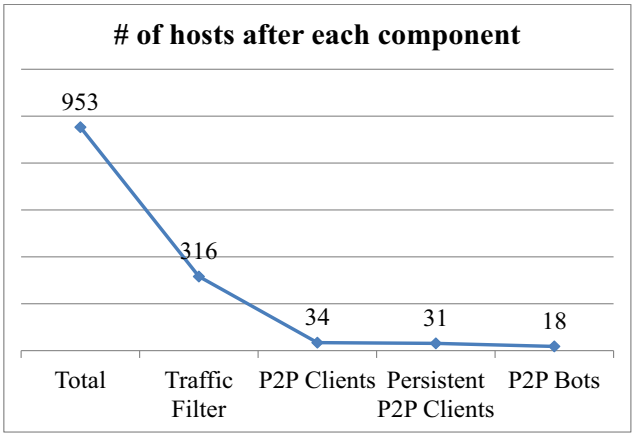


Figure 18: Number of hosts identified by each processing component

Θ_{bot} and Cnt_{birch} is discussed in Section 3.4.3.5.

3.4.3.1 Identifying and Profiling P2P Applications

We applied our detection system on data set D_1. The number of hosts kept after each step is presented in Figure 18. Traffic reduction using DNS traffic significantly reduced the number of hosts and the number of flows we needed to process, thereby greatly reducing the workload for the coming steps. For example, as illustrated in Figure 18, other components only need to process approximately one-third of the hosts (316 out of 953) after traffic reduction.

Our system identified 34 hosts as P2P clients in total. These 34 hosts are composed of i) *all* 16 P2P bots, ii) *all* 10 hosts with 5 popular P2P applications we have tested, and iii) 8 other hosts in the college networks. For those 8 hosts, 3 are `Bittorrent`-related hosts (a.k.a, BT1@C, BT2@C and BT3@C), which have been verified by the content-based signatures. The remaining 5 identified hosts do not

Table 22: Fingerprint cluster summaries for 3 `Bittorrent` clients

Trace	Fingerprints
BT1@C	1 1 109 100, UDP 1 1 109 91, UDP 1 1 104 178, UDP 1 1 319 145, UDP 1 1 145 319, UDP
BT2@C	1 1 145 319, UDP 1 1 75 75, UDP 1 1 65 65, UDP
BT3@C	7 6 1118 1767, TCP

Table 23: Fingerprint cluster summaries for 5 potential `Skype` clients

Trace	Fingerprints
Skype1@C	1 1 73 60, UDP 1 1 76 60, UDP 1 1 75 60, UDP 1 1 72 60, UDP 1 1 74 60, UDP
Skype2@C	1 1 75 60, UDP 1 1 74 60, UDP 1 1 76 60, UDP
Skype3@C	1 1 72 60, UDP 1 1 74 60, UDP 1 1 79 60, UDP 1 1 76 60, UDP
Skype4@C	1 1 73 60, UDP
Skype5@C	1 1 74 60, UDP 1 1 75 60, UDP

match any content-based signature. We present their fingerprint cluster summaries ($\overline{Pkt_s}, \overline{Pkt_r}, \overline{Byte_s}, \overline{Byte_r}$, proto) in Table 22 and Table 23.

The fingerprint cluster summaries for 3 `Bittorrent` clients are presented in Table 22. For BT1@C and BT2@C, "(1 1 145 319 UDP)" is consistent with one fingerprint cluster of a sample `Bittorrent` trace described in Table 15. The fingerprint of BT3@C is different from other two, which may represent another version implementation of the `Bittorrent` protocol.

The fingerprint cluster summaries for the remaining 5 unknown P2P hosts are presented in Table 23. By referring to Table 15, their fingerprint cluster summaries are very close to those of the `Skype` trace. For example, "(1 1 75 60, UDP)" is shared by most of these clients and the sample `Skype` traffic. This indicates that these 5 hosts are mostly likely `Skype` clients.

Some fingerprint cluster summaries for `Storm` and `Waledac` are presented in Table 24. P2P bots in the same botnet exhibit great similarity on fingerprint clusters,

Table 24: Fingerprint cluster summaries for P2P bots

Trace	Fingerprints	Trace	Fingerprints
Storm1	2 2 94 554, UDP 2 2 94 1014, UDP 2 2 94 278, UDP ...	Storm2	2 2 94 554, UDP 2 2 94 1014, UDP 2 2 94 278, UDP ...
Waledac1	4 3 224 170, TCP 3 3 186 162, TCP 5 4 286 224, TCP ...	Waledac2	4 3 224 170, TCP 3 3 186 162, TCP 5 4 285 224, TCP ...

while their fingerprint clusters are different compared to those of another P2P botnet and legitimate P2P applications (e.g., `Bittorrent` and `Skype`). We applied our system on D_2 to investigate whether our system can effectively profile different P2P applications if a bot-compromised host is also running a legitimate P2P application. Table 25 presents several fingerprint cluster summaries for two bots overlaid with legitimate P2P applications, `Waledac2+BT2@C` and `Storm4+Skype4@C`. For the example of `Waledac2+BT2@C`, we found that its fingerprint clusters come from two applications, where "(1 1 145 139, UDP)" and "(1 1 75 75, UDP)" are from `Bittorrent` protocol (referring to the second row in Table 22), and "(4 3 224 170, TCP)" together with "(3 3 185 162, TCP)" are from `Waledac` (referring to Table 24).

These experimental results demonstrate that our system can effectively identify hosts engaging in P2P communications. In addition, the generated fingerprint clusters can effectively profile P2P applications.

3.4.3.2 Detecting P2P Bots

We applied our system on D_1 to detect P2P bots. As we discuss in Section 3.4.3.1, the system identified 34 P2P hosts. By estimating the active time of the P2P application for each of the 34 hosts, our system identified 31 hosts exhibiting *persistent* P2P communications.

For these 31 hosts, our system constructs a hierarchical tree (Figure 19(a)) by

Table 25: Fingerprint cluster summaries for the `Storm` botnet and the `Waledac` botnet

Trace	Fingerprints
Waledac2+BT2@C	1 1 145 319, UDP (Bittorrent) 4 3 224 170, TCP (Waledac) 3 3 185 162, TCP (Waledac) 1 1 75 75, UDP (Bittorrent) ...
Storm4+Skype4@C	2 2 94 554, UDP (Storm) 2 2 94 1014, UDP (Storm) 1 1 73 60, UDP (Skype) ...

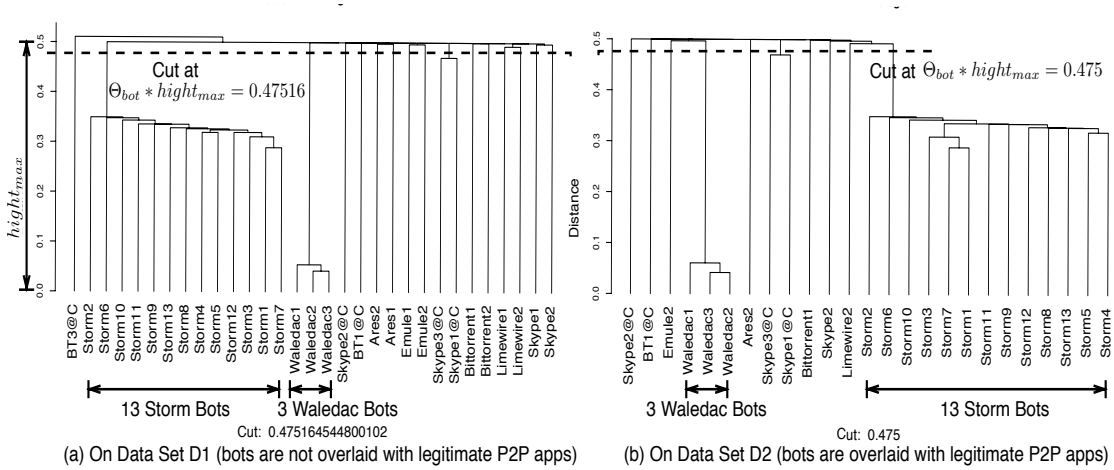


Figure 19: Hierarchical tree on persistent P2P hosts

evaluating the distance ($dist(h_a, h_b)$ defined in Section 3.3.6) between P2P hosts. P2P bots share same P2P protocol and have large overlap of the peer IP addresses in fingerprint clusters, thereby resulting in small distances and dense clusters in consequence. As shown in the Figure 19(a), both `Storm` and `Waledac` bots have small distances to each other and form dense clusters respectively. We cut the tree at $\Theta_{bot} * height_{max} = 0.475$ ($\Theta_{bot} = 0.95$) to identify dense clusters. As a consequence, three clusters are identified and therefore a total of 18 hosts were labeled as suspicious. All 16 P2P bots were detected, resulting in a high detection rate of 100% and a low false positive rate of 0.2% (2/953). The false positives appear to be two `Skype` clients. The reason for these two false positives is the conservatively configured value of Θ_{bot}, which is close to 1.

3.4.3.3 Detecting P2P Bots Overlaid with P2P Applications

We applied our detection system on data set D_2 to evaluate the detection accuracy when a bot-compromised host happens to run a legitimate P2P application. Table 20 presents some statistics of the bot traces *before* and *after* overlaying legitimate P2P application traces. Some of bot-compromised hosts' traffic profiles are significantly distorted after traffic overlaying. For example, after overlaying `BT2@C` (a real P2P client identified in the college network) traffic to the `Waledac2` traffic, the average flow size is increased from 11372 to 348708 and the number of destination IP addresses, which are involved in the successful outgoing connections, is also increased from 760 to 1359. It is because the `Bittorrent` application could be actively used for downloading/uploading files, thereby dominating the traffic profile of the host. In this case, if we use the traffic profile of the *entire* host (e.g., the average flow size and number of destination IP addresses) to detect the bot, the bot behavior will be concealed by the `Bittorrent` traffic. As a consequence, the detection approaches such as [79], which use the traffic profile of the *entire* host for detection, will lose

effectiveness.

However, since our system leverages fine-grained information of *fingerprint clusters*, which describe the profiles of *P2P applications* instead of *entire host*, it can still detect bots even if their underlying hosts are running legitimate P2P applications. The hierarchical tree for detection is presented in Figure 19(b), where `Waledac` bots and `Storm` bots still form dense clusters. Compared to the hierarchical tree in Figure 19(a), the tree structure in Figure 19(b) stays stable, which is not affected by the overlaid legitimate P2P applications. It is because the distance of two bot-compromised hosts is based on the minimum distance of fingerprint clusters from two P2P bots, the new fingerprint clusters introduced by the P2P application would not affect the minimum distance.

In D_2, our system identified 26 P2P clients, where 25 out of them exhibit persistent P2P behaviors. With $\Theta_{bot} = 0.95$, we cut the tree at 0.475, and identify three groups of hosts (18 in total). Among these 18 suspicious hosts, all 16 P2P bots are successfully identified with a low false positive rate (0.2%). The detection result is not affected by the overlaid traffic from legitimate P2P applications. This demonstrates that our system can effectively detect bots even if bot-compromised hosts run legitimate P2P applications.

3.4.3.4 Detection Performance in Special Cases

It is possible that in the monitored network, only two hosts are compromised by bots from the same botnet. We applied our system on data set D_2', and achieved the detection rate of 100% and false positive rate of 0.2%.

It is also possible that the monitored network is "clean", where no host is compromised by P2P bot. In this case, the false positive is a concern. We applied our system on D_1', which simulates a "clean" network environment, where we get a low false positive rate of 0.2%.

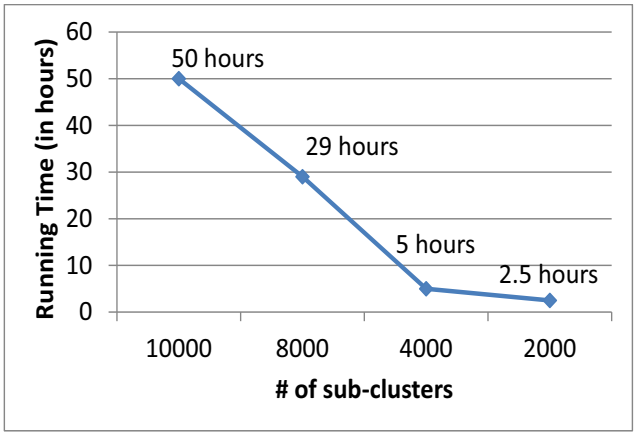


Figure 20: System performance with different values of Cnt_{Birch}

3.4.3.5 Analyzing The System Scalability

We optimize the system scalability by designing a two-level clustering-based flow analysis scheme and distributing workload to achieve parallel computation. Two parameters are involved respectively, Cnt_{birch} and M. Cnt_{birch} represents the expected number of sub-clusters from BIRCH. As Cnt_{birch} decreases, the second-level hierarchical clustering operation will process less sub-clusters and thus achieve better performance. Figure 20 presents the system running time on a single computation node as we decrease the value of Cnt_{birch}. By reducing Cnt_{birch} from $10,000$ to $2,000$, the system performance is improved by 95% (from 50 hours to 2.5 hours). M represents the number of computation nodes we can use to distribute the workload. As M increases, each node processes less workload. Figure 21 illustrates the system running time as we increase M from 1 to 10 ($Cnt_{birch} = 4000$). We randomly pick 10 hosts to run them in a host to obtain regression model, whose time consumption is ignorable. For each value of M, we repeat the experiments 5 times and report the average running time. As demonstrated in the experimental results, distributing system workload can significantly improve the system performance. For example, the running time is reduced from 5 hours to 0.69 hours by partitioning the workload to 10 hosts. We also compare our load-balance partition strategy to the random-partition

method. As indicated in Figure 21, our design performs better. All of these experiments achieve the same detection accuracy with 100% detection rate and 0.2% false positive rate.

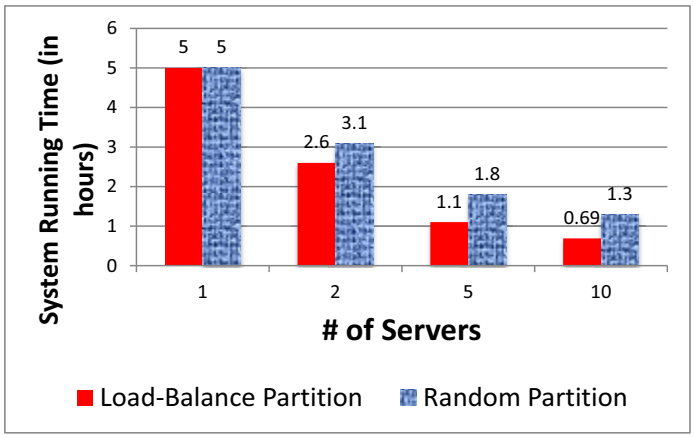


Figure 21: System performance with workload distribution

3.4.3.6 Analyzing The Effect of System Parameters

While the measurement in Section 2.3 motivates the parameter values for Θ_o and Θ_{p2p}, we study system parameters Θ_{bpg} and Cnt_{birch} in this section.

Θ_{bpg} is used to identify fingerprint clusters, where each fingerprint cluster is likely to contain network flows corresponding to the same type of P2P control messages for a particular P2P application. Therefore, the ideal solution to obtain the Θ_{bpg} value has three steps. First, given the network flows generated by a P2P application, we perform clustering-based flow analysis in order to obtain flow clusters. Second, we perform content analysis in order to label as fingerprint clusters those flow clusters whose flows contain control messages. Finally, we can enumerate the number of distinct BGP prefixes in each fingerprint cluster and select the minimum value for Θ_{bpg}. While not impossible, such solution suffer from a great practical challenge since it is practically extremely hard to get all P2P applications, especially P2P bots. Therefore, in our current system, we estimate the value of Θ_{bpg} based on empirical studies. To be specific, we first run several popular P2P applications (e.g., five applications

in Table 14), and then perform clustering-based flow analysis. Further, for each application, we get the maximum number of BGP prefixes in a cluster, as indicated in each entry in the N_{bgp} column in Table 14. Obviously, the Θ_{bpg} value should not be larger than the minimum value, denoted as B_{min} ($B_{min} = 1133$ in this specific case), in the N_{bgp} column. Otherwise, our system will miss the chance to detect certain P2P applications. Therefore, the value of Θ_{bpg} should fall into the range of $[2, B_{min}]$. Practically, we tend to obtain a small value in this range, by considering the possibility that bots may be more stealthy compared to a popular P2P application, thereby contacting a smaller number of peers/networks. However, the side-effect of a small Θ_{bpg} is that it may identify some small clusters, whose flows are not used for P2P control messages, as fingerprint clusters. Fortunately, such problem could be mitigated by our system since we investigate the temporal persistence of each fingerprint cluster and only use *persistent fingerprint clusters* to evaluate the distance between two P2P clients. In our current system, we conservatively set Θ_{bpg} to be a small value (e.g., 50) in this range. However, we evaluated different Θ_{bpg} values covering a large range inside $[2, B_{min}]$. Table 26 presents the evaluation results, including the number of fingerprint clusters, persistent fingerprint clusters, number of detected bots, and false positives, where $Cnt_{birch} = 4000$ and $\Theta_{bot} = 0.95$. As illustrated in the table, as Θ_{bpg} increases, the number of fingerprint clusters and persistent fingerprint clusters drops. Particularly, if Θ_{bpg} is extremely small (e.g., 2), it may identify many small clusters as fingerprint clusters and consequently introduce a large number of false positives. For example, the system generates 13 false positives if we set $\Theta_{bpg} = 2$. Similarly, if Θ_{bpg} is too large, it may discard those fingerprint clusters for P2P bots, resulting in a huge number of false negatives. For example, all bots are missed when $\Theta_{bpg} \geq 1000$. However, our system has demonstrated it effectiveness over a large range of Θ_{bpg} values. For example, our system achieves great detection performance when Θ_{bpg} falls into a large range of $[30, 200]$. Such experiment implies that $[3.5\%B_{min}, 40\%B_{min}]$ would be

Table 26: The evaluation of the Θ_{BGP} parameter, $Cnt_{birch} = 4000$ and $\Theta_{bot} = 0.95$

Θ_{BGP}	# of fingerprint clusters	# of persistent fingerprint clusters	# of detected bots	# of false positives
2	24,185	2,840	16	13
10	3,884	2,445	16	5
20	2,311	1,973	16	5
30	1,709	1,522	16	3
40	1,399	1,268	16	2
50	1,200	1,104	16	2
60	1,069	997	16	2
70	942	879	16	2
80	820	768	16	0
90	724	682	16	0
100	656	621	16	0
150	409	393	16	0
200	305	297	16	0
500	92	91	16	0
800	27	26	9	0
1000	15	14	0	0
1600	12	11	0	0

a feasible configuration.

Cnt_{birch} may introduce a trade-off between system efficiency and effectiveness. For example, by decreasing Cnt_{birch}, the system has less vectors to process in *Hierarchical clustering* and thus increase the system efficiency. However, a small Cnt_{birch} may force dissimilar flows to be aggregated into the same sub-cluster and therefore into the same fingerprint cluster, resulting in inaccurate fingerprint clusters. To evaluate Cnt_{birch}, we conducted the following experiments. We applied our system D_2 with different Cnt_{birch} values, including 2000, 4000, 8000 and 10000. For each Cnt_{birch} value, we further adopted different Θ_{bot} (i.e., 0.1, 0.3..0.95) values to evaluate the detection rate and false positive rate. The results of detection rate (DR) and false positive (FP) rate are described in Table 27. The experimental results indicate that the detection performance is stable over a large range of Cnt_{birch} (e.g., ≥ 4000) and $\Theta_{bot} \in [0.7, 0.95]$ is a good candidate value. This experiment also suggests that 0.8 or 0.9 may be a better value for Θ_{bot}. This implies that when a labeled data set of P2P botnet traffic

Table 27: Detection rates and false positive rates for different values of Θ_{bot} and Cnt_{birch}

		Θ_{bot}						
Cnt_{birch}	-	0.1	0.3	0.5	0.7	0.8	0.9	0.95
2000	DR	0	0	2/16	3/16	16/16	16/16	16/16
	FP	0	0	0	0	0	0	2/953
4000	DR	2/16	3/16	3/16	16/16	16/16	16/16	16/16
	FP	0	0	0	0	0	0	2/953
8000	DR	2/16	3/16	3/16	16/16	16/16	16/16	16/16
	FP	0	0	0	0	0	0	2/953
10000	DR	2/16	3/16	3/16	16/16	16/16	16/16	16/16
	FP	0	0	0	0	0	0	2/953

is available we can tune this threshold (Θ_{bot}) to find a better trade-off between false positives and false negatives.

In summary, our system can effectively detect all the P2P bots with a very low false positive rate, even if the bot-compromised hosts are running legitimate P2P applications. Our system is stable over a large range of values for system parameters and shows great scalability.

3.5 Discussion

For practical deployment, the system can be configured to automatically run daily. In this case, `Argus` and `dnscap` collect flow and DNS data in real-time and our detection system analyzes the data in batches at the end of each day. The memory consumption is mainly constrained by the maximum number of flows per host.

If botmasters get to know about our detection algorithm, they could attempt to modify their bots' network behavior to evade detection. This situation is analogous to evasion attacks against other intrusion detection systems. Since our detection algorithm is based on differentiating P2P protocols used by P2P bots from legitimate P2P applications, botmasters may instruct the bots to join existing legitimate P2P networks, and use legitimate P2P networks to propagate commands. The initial version of `Storm` adopted this strategy. However, such approach exposes the botnet to *sybil*

attacks, where researchers can infiltrate the P2P network and enumerate/detect the bots [81]. Therefore, current P2P botnet, including `Storm` and `Waledac`, isolate their own P2P network from existing legitimate P2P networks. Botmasters may leverage our traffic volume reduction component to evade detection. For example, the botmaster may set up a malicious DNS server, and instruct each bot to query this server before contacting any peer, asking the malicious DNS server to return a response containing the peer's IP address. In this case, our traffic reduction component would eliminate the corresponding flows from the analysis. To avoid this evasion attempt, we could filter traffic based only on DNS responses for popular domains, i.e., domains queried by a non-negligible fraction of hosts in the monitored networks. Bots could also intentionally try to reduce the number of contacted peer IPs (or BGP prefixes) or the active time of the bot, in order to bypass the P2P client identification or the component that detects persistent P2P applications. However, such techniques could have a serious negative impact on the resiliency of the C&C infrastructure and limit the usability of the entire botnet. Another evasion approach could exploit the Θ_{p2p} threshold. For example, the P2P bots could exchange traffic for a short period of time, then go idle for several hours, and repeat this pattern. However, this evasion technique is equivalent to increasing the churn rate for the P2P nodes, which may eventually bring to a complete disruption of the overlay network [30]. Bots could also randomize their P2P communication patterns to prevent our system from getting an accurate profile of P2P protocols. For example, bots could inject noise into network flows related to P2P control messages. In this case, we could use other features (e.g., the distribution of flow sizes) to profile the P2P protocols. A P2P botnet could also attempt to reduce the overlap between peers contacted by the bots. For example, the botnet could partition the peers into different sets and ask each bot to contact disjoint sets of peers. Such technique may require a lot of efforts for the design and operation of the P2P botnets. We leave the analysis of such complex botnets to future

work. We should always strive to develop more robust defense techniques. Combining different complementary detection techniques to make the evasion harder is one of the possible directions that we intend to explore in our future work.

3.6 Summary

Detecting botnets in their control-phase is of great importance. In this chapter, we present a novel botnet detection system that is able to identify botnets with P2P C&Cs, which represent currently the most robust C&C structures against disruption efforts. Especially, our system aims to detect all P2P botnets, even in the case in which their attack behaviors are extremely hard to be observed in the network traffic. To accomplish this task, we first identify all hosts within a monitored network that appear to be engaging in P2P communications. Then, we derive *statistical fingerprint*s of the P2P communications generated by these hosts, and leverage the obtained fingerprints to distinguish between hosts that are part of legitimate P2P networks (e.g., file-sharing networks) and P2P bots. We also optimize the scalability of our system to achieve great efficiency to process a large volume of data. We have implemented a prototype version of our system, and performed an extensive experimental evaluation. Our experimental results confirm that the proposed system can detect stealthy P2P bots with a high detection rate and a low false positive rate, and also achieves great scalability.

CHAPTER IV

BOOSTING THE SCALABILITY OF BOTNET DETECTION SYSTEMS

4.1 Motivation

Due to the severity of botnet threats, botnet detection has attracted intensive research efforts. As a result, a number of detection systems have been proposed, and most of them [45, 40, 33, 9, 23, 78, 89, 36] focus on detecting botnets in the control phase. These systems have demonstrated promising detection results. However, these systems may suffer from limited scalability when they process a huge volume of network traffic, which is typical for high-speed and high-volume networks. Their limited scalability mainly stems from their dependence of *deep packet inspection* (DPI) techniques, based on which they perform fine-grained analysis on the payload of network packets. For example, BotHunter [35] uses a payload-based anomaly detector and a signature-based detection engine. BotSniffer [33] and Rishi [40] need to parse the content of IRC communications. T$\bar{A}$MD [78] inspects packet payloads to compute content similarity scores. BotMiner [36] requires DPI to perform *activity-plane* (A-Plane) monitoring, such as binary downloading and remote exploit detection. Although BotMiner's *communication-plane* (C-Plane) analysis does not require DPI, it suffers from scalability issues that prevents its deployment in high-speed networks (Section 4.4.3.2). While these systems have shown promising results, because DPI is computationally expensive, they cannot be directly deployed in high-speed or high-volume networks without special (usually very expensive) hardware support. Furthermore, even when special hardware support is available, most of the proposed techniques may still not be able to keep up with the traffic, due to the relatively high computational cost of

their traffic analysis algorithms. Load-balancing (i.e., distributing traffic and computation to multiple processing units) may represent a possible solution. However, a deployment of these systems in load-balancing requires special design and significant changes to the existing detection algorithms.

In this chapter, we will present our contribution on boosting the scalability of botnet detection systems. To be specific, we propose a new packet sampling and scalable spatial-temporal flow correlation approach that aims to *efficiently* and *effectively* identify a small number of suspicious hosts that are likely bots. Their traffic can be forwarded to fine-grained botnet detectors for further analysis. This allows us to significantly reduce the amount of traffic on which fine-grained analysis such as DPI is applied. Thus, we boost the scalability of botnet detection for high-speed and high-volume networks.

Network flow analysis typically requires far fewer resources than DPI. However, collecting *precise* network flow information in high-speed networks is challenging, because we may not be able to afford to process every packet in the network. In order to solve this problem, packet sampling techniques are commonly employed to reduce the number of packets to be processed. For example, *uniform sampling* and its variant *periodic sampling* are among the most popular packet sampling techniques, and they allow a network operator to reconstruct approximate network flow information. However, their limitation is that they are able to reconstruct relatively precise information about large flows (i.e., flows that carry a high number of packets), such as media streaming flows, but may poorly approximate or miss outright information about small and medium flows. In order to address this issue, some new sampling algorithms have been recently proposed. For example, FlexSample [14] is a programmable framework where a network operator can set conditions to increase the sampling rates packets from specific traffic subpopulations (e.g., packets in small and

medium flows). Unfortunately, because different botnet implementations may introduce strong diversity in the properties (e.g., flow size) of their C&C communication flows, it is challenging to set conditions that allow FlexSample to sample packets targeted for a wide range of botnet C&Cs. For example, flows of HTTP-based C&Cs are usually small (i.e., short lived) while those related to IRC-based C&Cs are intrinsically larger. In order to address this problem, we introduce a new *adaptive* sampling technique. Our sampling technique is *botnet-aware* since it is driven by intrinsic characteristics of botnets such as *group similarity*, where the group similarity reflects the fact that bots belonging to the same botnet share similar C&C communication patterns. We also propose a new scalable spatial-temporal correlation approach to identify hosts that share *persistently similar* communications. That is, we aim to identify hosts in a network that persistently share similar communication patterns for a relatively long (not necessarily continuous) period of time. Our spatial-temporal flow correlation analysis is motivated by the following observation. Because of their (illegal) economy-driven nature, botnets are used by the botmasters for as long as possible to maximize profits (e.g., several months, or until the botnet is dismantled by law enforcement), so their C&C communications will be active for a relatively long period of time.

Our work makes the following contributions:

1. We propose a network traffic analysis approach for botnet detection in high-speed and high-volume networks. The objective of our analysis is to efficiently and effectively narrow down suspicious hosts that are likely to be bots. The network traffic generated by these suspicious hosts can then be forwarded to fine-grained botnet detectors for further analysis.

2. We introduce an *adaptive* sampling technique based on *group similarity*, an intrinsic characteristic of botnets, to sample packets that are likely related to C&C communications with high probability.

3. We propose a new scalable spatial-temporal correlation analysis to identify hosts in a network that share *persistently similar* communication patterns, which is one of the main characteristics of botnets.

4. We implemented a proof-of-concept version of our system, and evaluated it using *real-world* legitimate and botnet-related network traces. Our experimental results show that the proposed approach is scalable and can effectively detect bots with a small number of false positives, which can be further reduced by fine-grained botnet detection systems.

4.2 Related Work

Researchers have proposed many approaches to detect botnets. Some of the approaches [23, 89, 9, 40, 45] are designed for detecting botnets with IRC-based C&Cs, relying on analysis of packet content for detection. Some other detection approaches are driven by specific attack information (i.e., spam). Ramachandran et al. [12] used DNSBL to identify bots for spamming, while Zhao et al. used Hotmail logs in BotGraph [94]. Hu et al. [90] proposed RB-Seeker to detect redirection botnets based on spam and network flow information. Compared to these approaches, our system mainly uses packet header and network flow information, indicating a wider deployment. Some detection algorithms uses correlation approaches. BotHunter [35] associates IDS events to a pre-defined bot infection dialog model for detection. BotSniffer [33] leverages the homogeneity of messages and activities to identify botnet C&Cs. Yen et al. [78] proposed T$\bar{A}$MD to detect bots by aggregating traffic which shares the same external destination, similar payloads and OS platforms. BotMiner [36] is a protocol- and structure-independent botnet detection system using clustering techniques. These systems depend on DPI-based components, which limit their usage in high-speed networks. In our system, we design botnet-aware packet sampling algorithm and scalable spatial-temporal flow correlation approach for efficient and

effective botnet detection, which aims at the deployment in high-speed networks.

Various sampling algorithms have been proposed to reduce the amount of data the network devices have to process in high speed networks and infer the traffic statistics based on the sampled packets. Most of them focus on sampling large flows and improving their estimation accuracy [93]. Recently researchers proposed approaches to focus on sampling packets in small flows. Kumar et al. [10] and Hu et al. [22] proposed algorithms to sample packets in small flows. However, their overall sampling rate depends on the Zipfian nature [88] of Internet and thus they cannot achieve a predefined target sampling rate. Ramachandran et al. [14] designed FlexSample, which can sample packets based on pre-defined conditions. FlexSample can be configured to capture packets in small/medium flows while keeping a target sampling rate. However, characteristics of network flows for botnet C&Cs exhibit great diversity among different botnets and thus it is very challenging to propose good conditions to describe all the flows of botnet C&Cs. Therefore, these existing sampling algorithms maybe ineffective to sample packets for botnet C&Cs. In contrast to the above sampling algorithms, our algorithm is driven by the intrinsic characteristics of botnet C&Cs, and thus our sampling algorithm captures more botnet packets related flows given a certain sampling rate.

4.3 System

4.3.1 System Overview

As shown in Figure 22, our botnet detection framework has three components: Flow-Capture, Flow-Correlation, and Fine-Grained Detector.

The Flow-Capture module aims to monitor the traffic at the edge of high-speed networks to gather network flow information based on the sampled packets. The Flow-Capture module is further divided in two components: Packet-Sampling and Flow-Assembler. Packet-Sampling is a botnet-aware sampling algorithm. Given an

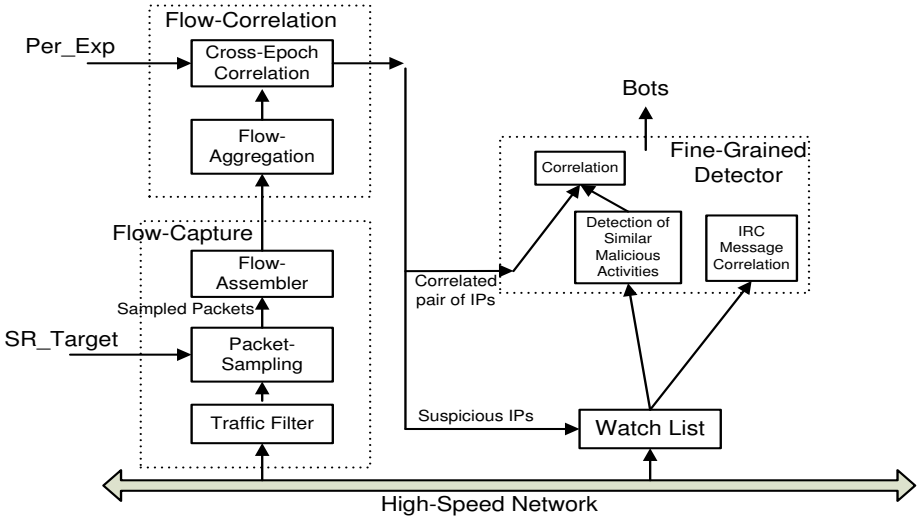


Figure 22: Architectural overview

overall target sampling probability (SR_{Target}), it samples packets likely related to botnet C&C communications and delivers them to Flow-Assembler, along with their corresponding *instant* sampling probabilities (Section 4.3.2). The Flow-Assembler assembles sampled packets into **raw flow**s (defined in Section 4.3.2.2).

The Flow-Correlation module groups flows output by Flow-Assembler into C-flows (defined in Section 4.3.3.1). A C-flow is an abstraction introduced in BotMiner [36] to represent the C&C communication patterns of potential bots. Each C-flow represents a view of the communication patterns from a monitored host to a remote service over a certain epoch (e.g, 12 hours). Flow-Correlation applies a scalable clustering algorithm over the C-flows to identify hosts that exhibit similar communication patterns towards machines outside the monitored network. This step is similar to the C-Plane analysis performed by BotMiner [36], but there are two fundamental differences. First, we use a significantly more efficient flow clustering process (see Section 4.3.3.2), compared to BotMiner, which can handle large traffic volumes typical of high-speed networks. Second, unlike BotMiner, our Flow-Correlation module performs *cross-epoch* correlation to identify hosts that show persistently similar communication pattens, a telltale sign of botnets. Any pair of hosts that exhibit persistently similar communication

patterns will then be labeled as suspicious hosts (potential bots) and delivered to the Fine-Grained Detector for further in-depth analysis. The Fine-Grained Detector can then focus on monitoring the packets related to only the suspicious IPs provided by our Flow-Correlation module, thus reducing the overall cost of the botnet detection process.

The design and implementation of the Flow-Capture and Flow-Correlation modules and the detection framework are the main contributions of this work. Existing DPI-based botnet detectors can be plugged within our framework with little or no modification to constitute the Fine-Grained Detector module. We developed a Fine-Grained Detector derived from BotMiner [36] and BotSniffer [33], and we plugged it into our botnet detection framework. In particular, we used two components: i) an implementation of the malicious activities detector derived from BotMiner's A-Plane monitor, which can identify groups of similar malicious activities based on the attack features (e.g., the scanned port, the exploits or binary content), and ii) BotSniffer's IRC-based botnet detection module. Similar to the Cross-Plane correlation in BotMiner, the correlation component correlates communication patterns and activity patterns to detect bots. Any pair of IPs that share persistently similar communication patterns (generated by Flow-Correlation) and similar malicious activities (generated by the malicious activities detector) are labeled as bots by the correlation component. And any host identified by the BotSniffer's IRC-based botnet detection module will be labeled as bot.

4.3.2 Flow Capture

The Flow-Capture performs packet sampling and reassembles raw flows using a novel *botnet-aware* adaptive sampling algorithm, which we call B-Sampling. Our B-Sampling algorithm leverages the *intrinsic* characteristic of bots, namely *group similarity*, to guide the sampling procedure. Given a pre-defined target sampling rate, B-Sampling

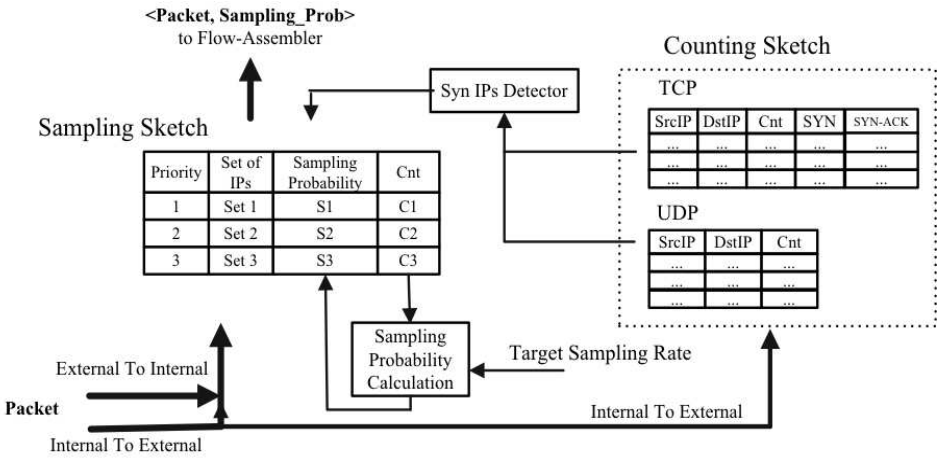


Figure 23: The architecture of the packet sampling component

adaptively tunes the instant sampling probabilities for different categories of IPs. For example, priority will be given to packets related to IPs that share similar communication patterns, while keeping the overall sampling rate close to the overall target sampling rate SR_{Target}. The target sampling rate is usually suggested by the process capacity of the monitor device and the traffic speed of the monitored network. For example, the monitor device with capacity of Cap_{device}bps and the network with the speed of $Cap_{network}$bps indicate $SR_{Target} = \frac{Cap_{device}}{Cap_{network}}$.

4.3.2.1 Packet Sampling

As described in Figure 23, Packet-Sampling has four components: Counting-Sketch, Sampling-Sketch, Synchronized IPs Detector (SID), and Priority-based Sampling Probability Calculation (PSPC). Counting-Sketch tracks the number of packets sent from a SrcIP to a DstIP. After each time interval of T, Packet-Sampling transfers the Counting-Sketch to the SID, and then resets the Counting-Sketch to 0 for next interval. The end of each time interval also triggers the SID and PSPC to identify IP addresses with synchronized behaviors and recalculate the instant sampling probability for each category of IPs. Sampling-Sketch gets the instant sampling probability

for a packet and decides whether this packet is going to be sampled.

Counting/Sampling Sketch The Counting-Sketch is a table indexed by $Hash(SrcIP||DstIP)$ for TCP and UDP packets, where each entry in the table is defined as a *track-flow*. Each entry contains a pair of IPs ($SrcIP$ and $DstIP$) and a counter *cnt*, which represents the number of packets for this pair of IPs. For TCP packets, the entry keeps $SYN/SYNACK$ flag. On arrival of a packet, the SrcIP and DstIP will be recorded and the counter in the corresponding entry will be increased by 1. The Counting-Sketch only handles the packets from internal networks to external networks. Such design can simplify the system implementation by just monitoring the separated physical line for outgoing traffic. Moreover, it reduces the time and memory consumption to access the table. Counting-Sketch is reset to be 0 after the time interval T (currently 15 minutes).

Each entry in Sampling-Sketch records a category/set of IPs, a counter of packets related to these IPs, a sampling probability and a priority. On arrival of a packet, Sampling-Sketch checks the category of this packet based on its SrcIP and DstIP. It then finds the instant sampling probability (p_i) for the corresponding category and samples this packet with probability p_i. The sampled packets, together with their sampling probabilities, are sent to Flow-Assembler.

Synchronized IPs Detector The SID identifies two kinds of hosts with synchronized behaviors: i) *syn-servers*: the hosts in external networks whose clients have similar network behaviors; ii) *syn-clients*: the hosts in internal networks that share similar network behaviors to multiple destination hosts.

The detection of syn-servers is motivated by the network behavior of C&C servers for centralized-based botnets, where their clients (bots) are synchronized and thus share similar network behaviors. For the legitimate servers, especially the popular ones, their clients' behaviors usually diverse from each other due to various usage

patterns of different users. The detection of syn-clients is motivated by the network behaviors of P2P-based C&Cs. P2P-based bots usually actively query their peers to maintain the overlay P2P network for botnet C&Cs. Such behaviors will cause many similar connections to multiple peer bots.

To detect syn-servers and syn-clients, we introduce "homo-server" and "similar-client".

1. Homo-server: We aggregate entries in Counting-Sketch based on each DstIP. For each DstIP that has at least two SrcIPs, we calculate the variance of the track-flow sizes. We sort the variances and get the medium value v_{medium}. For one DstIP, if its variance $v_i < v_{medium}$, we mark it as a homo-server. Otherwise, we take the server as non-homo-server if it has at least two SrcIPs.

2. Similar-clients: We keep an array of bins (denoted as B in Algorithm 3) and a pre-defined size R (currently $R = 10$). Each bin b_i is represented by its center that is the average size of track-flows in this bin. For a flow with size L, if $|L - b_i.center| \leq R$, we insert this track-flow into b_i and then update the $b_i.center$. Otherwise, we build a new bin and insert this flow into it. In each bin, if we find a pair of SrcIPs and each of them has more than C (currently $C = 10$) flows (e.g., connecting to C different DstIPs), we take this pair of SrcIPs as similar-clients.

On identifying the syn-clients, we currently discard the TCP and UDP track-flows with size smaller than 10 to avoid potential false positives generated by popular network services like DNS or by the scanning-like behaviors. On identifying the homo-servers, we ignore the TCP track-flows with size of 1 or with only $SYNACK$ flag. A TCP track-flow with size of 1 indicates an unsuccessful connection. The

Algorithm 3: Identify Synchronized Hosts

Input: Counting Sketch, Set_d, t_{cur}
Output: Set_d: Records for syn-clients/servers.
begin
  foreach *Record* $R \in Set_d$ **do**
    if $t_{cur} - R.timestamp \geq T_{rec}$ **then**
      Remove R from Set_d;
  foreach *DstIP* dh_i *in the Counting Sketch* **do**
    if dh_i *is homo-server* **then**
      $Arr.get(dh_i).score+ = step_{up}$;
      if $Arr.get(dh_i).score \geq TH_{syn-server}$ **then**
        $set_d.add(dh_i, t_{cur})$; $Arr.get(dh_i).score = TH_{syn-server}$;
    if dh_i *is non-homo-server* **then**
      $Arr.get(dh_i).score- = step_{down}$;
      if $Arr.get(dh_i).score \leq TH_{down}$ **then**
        $Arr.get(dh_i).score = TH_{down}$;
  foreach *SrcIP* sh_i *in the Counting Sketch* **do**
    if sh_i *is similar-client* **then**
      $Arr.get(sh_i).score+ = step_{up}$;
      if $Arr.get(sh_i).score \geq TH_{syn-client}$ **then**
        $set_d.add(sh_i, t_{cur})$; $Arr.get(sh_i).score = TH_{syn-client}$;
    else
      $Arr.get(sh_i).score- = step_{down}$;
      if $Arr.get(sh_i).score \leq TH_{down}$ **then**
        $Arr.get(sh_i).score = TH_{down}$;
  return Set_d;
end

flag of *SYNACK* indicates a TCP connection initiated from external networks, which is unlikely a connection for botnet C&Cs. Bots usually initiate connections to external C&C servers for two reasons. First, the widely deployed firewall/NAT devices block the connections initiated from external networks. For example, researchers have shown that more than 40% `storm` bots are behind a firewall or NAT [20]. Second, the dynamic IPs make it very hard for C&C servers to initiate connections to bots with dynamical IPs accurately.

For each time interval T, we identify the homo-servers and similar-clients. We

accumulate evidence over multiple intervals to decide whether a host is syn-server or syn-client. We keep each syn-server and syn-client in the Sampling Sketch for T_{rec} (currently $T_{rec} = E/2$, where E is one epoch of 12 hours) from its last update. The algorithm is described in Algorithm 3. $TH_{syn-server/client}$ is the threshold of the score to identify syn-server/client. TH_{down} is the lower bound of the score. $step_{up/down}$ is the step to increase/decrease the score. We set $TH_{syn-server/client} = 4$, $TH_{down} = -10$, $step_{up} = 1$, and $step_{down} = 0.2$. *Record* represents one data structure for IP and time stamp. Arr is an array of scores indexed by the hosts and t_{cur} is the time stamp derived from current packet. If one record in Arr is not updated from its last update for T_{Arr} (currently $T_{Arr} = E/2$), we can eliminate it from Arr.

Algorithm 4: Priority-based Sampling Algorithm

Input: P_t, $f_1, f_2, \ldots, f_n$

Output: $p_1, p_2, \ldots, p_n$

begin

  $budget = 1$;

  foreach $i = 1 \ldots n$ **do**

    if $f_i == 0$ *or* $budget \leq 0$ **then**

      $p_i = 0$;

      continue;

    else

      $p_i = budget * \frac{P_t}{f_i}$;

      $p_i = p_i > 1?1 : p_i$;

      $budget- = p_i * \frac{f_i}{P_t}$;

  return $\{p_1, p_2, \ldots, p_n\}$;

end

Sampling Probability Calculation We dynamically calculate the sampling probability for each category of IPs to fulfill two targets: i) to get as many packets as possible that are related to syn-clients or syn-servers; ii) to keep the actual sampling

rate close to the target sampling rate.

To keep the actual sampling rate close to the pre-defined sampling rate, a scheme for allocating instant sampling probabilities for different categories has been proposed by Ramachandran et al. [14]. However, this scheme requires pre-configured budgets for different categories. Inappropriate allocated budgets may affect the packet sampling process. For example, the inadequate budget for synchronized IPs will cause the lost of packets related to botnet, while the over-allocated budget for synchronized IPs would be a waste of the resources. To fully utilize the resources to capture packets, we design a sampling algorithm named Priority-based Sampling Probability Calculation algorithm. The principle for this algorithm is as follows: under a pre-defined sampling rate, we use the available resources (budget) to capture as many packets in the first priority category as possible. The remaining available resource will be used to capture as many packets as possible in the next level priority category. Such process will continue until there is no further category or no available resource. Algorithm 4 shows this approach. P_t is the pre-defined target sampling rate. $\{f_1, f_2, \ldots, f_n\}$ is the fraction of packets in each category where $priority_1 > priority_2 \cdots > priority_n$. $\{p_1, p_2, \ldots, p_n\}$ is a set of instant sampling rates for different priorities and *budget* is for the available budget.

The following equation illustrates how the budget allocation helps the sampling component to keep a target sampling rate. Suppose there are a total of K packets and the target sampling rate is P_t. Given n categories and suppose each category has f_i fraction of the total packets and we give budget b_i to this category, we can calculate the sampling probability for category i as $p_i = P_t \frac{b_i}{f_i}$. In this case, the number of sampled packets Q and overall sampling rate would be $Q = \sum_{i=1}^{n} K f_i p_i = K \sum_{i=1}^{n} f_i (P_t \frac{b_i}{f_i}) = K P_t \sum_{i=1}^{n} b_i$. According to this equation, as long as $\sum_{i=1}^{n} b_i = 1$, the overall sampling rate $\frac{Q}{K}$ would be P_t, the target sampling rate. Since f_i cannot be obtained precisely in advance, we dynamically estimate f_i using WMA (weighted moving average) based

on the observed value for it in the previous and current intervals, which is $f_i = w_1 f_i^{prev} + w_2 f_i^{curr}$ where $w_1 = 0.2$ and $w_2 = 0.8$ in our current design. The system can dynamically assign $priority_1$ or $priority_2$ to syn-servers or syn-clients. The fewer the packets related to one of these two categories, the higher priority it has. The intuition behind such design is to use enough resource to build the accurate flows for the category that requires least resource. In practice, operators can also fix the priority or introduce more categories/priorities based on known knowledge (e.g., a category for the packets that are sent to confirmed bot peers). The packets related to the rest of IPs are labeled as the lowest priority ($priority_3$).

4.3.2.2 Flow Assembler

The Flow-Assembler assembles sampled packets to generate *raw flow*s, where each raw flow is identified by 5-tuple key (SrcIP, SrcPort, DstIP, DstPort, Proto). For TCP flow, the first two handshake packets (SYN and $SYNACK$) can be used to identify the flow direction. However, since packet sampling may result in the loss of TCP handshake packets, we use following approaches to identify TCP flow direction. First, if one of these two handshake packets is sampled, we can easily identify the flow direction. Second, for a TCP flow without TCP handshake packets sampled, we take this flow as it is initiated from internal networks (e.g., its SrcIP is from internal network). These approaches guarantee that every TCP flow from internal network will be attributed to the correct direction. Flow-Assembler outputs a flow if the flow is finished (e.g., the TCP FIN/RST flag is observed) or it expires (e.g., no packet comes for this flow for 10 minutes). For one raw flow, we record information including $time_{Start}$, $time_{End}$, $size_{Actual}$ (# of packets observed), $byte_{Actual}$ (# of bytes observed) and $size_{Est}$. $size_{Est}$ is the estimated flow size based on the sampled packets and their corresponding instant sampling probabilities. Suppose there are n packets for one raw flow and each packet has b_i bytes and sampling probability of p_i, we

compute the metrics for this raw flow as follows: $size_{Est} = \sum_{i=1}^{n} \frac{1}{p_i}$, $size_{Actual} = n$, $byte_{Actual} = \sum_{i=1}^{n} b_i$.

4.3.3 Flow Correlation

The goal of Flow-Correlation is to identify hosts with persistently similar communication patterns. By evaluating the capacity of the fine-grained detectors and the monitored network, operators can estimate the percentage of hosts Per_{Exp} (as described in Figure 22) that fine-grained detectors can afford to monitor. For example, if we assume that the traffic is evenly distributed over the hosts in the monitored network, the capacity of a fine-grained detector ($Cap_{detector}$bps) and the network speed ($Cap_{network}$bps) indicate a $Per_{Exp} = \frac{Cap_{detector}}{Cap_{network}}$. The Flow-Correlation component identifies groups of hosts (up to Per_{Exp}) that share most similar communication patterns and show persistence.

4.3.3.1 Flow Aggregation

We use **C-flow** to represent the communication pattern from a host to a remote host and port. We define a C-flow as a set of raw flows sharing same tuple of (SrcIP, DstIP, DstPort, Proto) in a certain epoch E (currently E = 12hours), denoted as $c = \{f_1, \ldots, f_n\}$. To get C-flows, we filter out the raw flows that satisfy either of two conditions: i) The raw flow is initiated from external network to internal network, where the reason is discussed in Section 4.3.2.1. ii) The raw flow has traffic in only one direction, which indicates an unsuccessful connection. We represent a C-flow ($c = \{f_1, \ldots, f_n\}$) using the following 10 features.

1. The *mean*s and *variance*s of fph (the number of flows per hour), ppf (the number of packets per flow), bpp (the number of bytes per packet), pps (the number of packets per second), which have similar definition in BotMiner [36]. We use $size_{Est}$ to compute ppf and pps, while $\frac{byte_{Actual}}{size_{Actual}}$ is used for bpp.

2. fph_{max}: the maximum number of flows per hour.

3. $time_m$: the median time interval of two consecutive flows.

4.3.3.2 Cross-Epoch Correlation

Given Per_{Exp}, cross-epoch correlation identifies pairs of IPs where each pair shares persistently similar communication patterns for at least M epochs out of totally N epochs ($M \leq N$).

We get a set (G) of C-flows over multiple epochs, and each C-flow has an epoch-tag. After clustering C-flows, we get a set of clusters $\{g_1, g_2, \ldots, g_n\}$ where each cluster g_i represents a set of similar communication patterns ($G = g_1 \cup g_2 \cup \cdots \cup g_n$). For C-flows in one cluster g_i, we further aggregate them into different groups (denoted as $\{c_i^1, c_i^2 \ldots c_i^N\}$ and $g_i = c_i^1 \cup c_i^2 \cdots \cup c_i^N$) according to their epoch-tags. For example, c_i^j represents the C-flows that are similar in jth epoch (spatial-similarity). For each cluster g_i, if a pair of SrcIPs share at least M common groups, it indicates that they share persistently similar communication patterns over at least M epochs. Therefore, we label this pair of SrcIPs as suspicious. We denote the percentage of all the detected suspicious IPs over all the SrcIPs as Per. Figure 24 presents an example of cross-epoch correlation. $A/B/C/D$ is the C-flow associated with the host $h_A/h_B/h_C/h_D$, and the remote host and port of $A/B/C/D$ are not necessarily to be the same over multiple epochs (e.g., A represents $< h_A, h_{remote}, port_{remote} >$). Some similar C-flows associated with $h_A/h_B/h_C/h_D$ are clustered together in a cluster g_i. By investigating the epoch-tag related to each C-flow, we aggregate these C-flows to three groups ($c^1/c^2/c^3$), as described in the left part of Figure 24. The right part of Figure 24 presents that h_A and h_B share 3 common groups, which indicates that they share similar communication patterns for 3 epochs. If we set $M \leq 3$, h_A and h_B are labeled as suspicious.

To get clusters of C-flows that represent similar communication patterns, we use

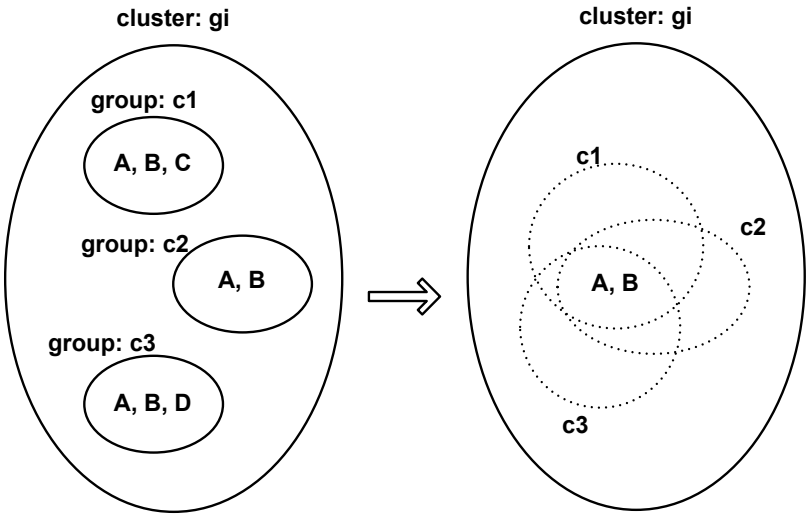


Figure 24: An example of cross-epoch-correlation

clustering algorithm. BotMiner uses two-level clustering scheme (X-Means and Hierarchical) that cannot scale well for large number of C-flows as shown in Figure 26. To process C-flows in an efficient manner, we use a scalable clustering algorithm Birch [84]. Given a certain value of "diameter", Birch can first efficiently discover clusters of C-flows within such distance. Second, cross-epoch correlation can detect suspicious IPs based on the clustering results. We repeat these two steps by increasing the value of "diameter". This process terminates when the percentage of suspicious IPs Per for the next step reaches at the expected percentage Per_{Exp} or the number of rounds reaches at a pre-defined $MaxRound$ (currently 50).

4.4 Evaluation

We implemented a prototype system and evaluated it using traces of real-world network traffic and different botnets.

4.4.1 Experimental Setup

We mounted our monitors on a span port mirroring a backbone router at the college network (200Mbps-300Mbps at daytime) to collect data. The traffic covers various applications and we believe such kind of traffic provides good traces to evaluate our

Table 28: Background traces

Trace	# of Pkts	Dur	Info
Mar25	205,079,914	12h	header
Mar26	280,853,924	24h	header
Mar27	318,796,703	24h	header
Mar28	444,260,179	24h	header
Mar31	102,487,409	1.5h	full

Table 29: Botnet traces

Trace	Dur	Bots
Bot-IRC-A	4days	3
Bot-IRC-B	4days	4
Bot-HTTP-A	4days	3
Bot-HTTP-B	4days	4
Bot-HTTP-C	4days	4
Bot-P2P-Storm	4days	2
Bot-P2P-Waledac	4days	3

system. The dataset contains TCP and UDP headers for continuous 3.5 days and full packets for 1.5 hours in Table 28. We eliminated a `B/16` subnet for dynamic IPs allocated for wireless connections, which are frequently changed and can not accurately represent the same hosts for multiple epochs. We observed a total of 1460 different IP addresses in 3.5 days. We also collected 1.5 hour traces with full payload.

We collected the traces of 7 different botnets including IRC-, HTTP- and P2P-based botnets, as described in Table 29. `Bot-IRC-A` and `Bot-HTTP-A` were collected by running bot instances ("TR/Agent.1199508.A" and "Swizzor.gen.c") in multiple hosts in the honeypot. `Bot-IRC-B` and `Bot-HTTP-B/C` were generated using *Rubot* [52], a botnet emulation framework. In `Bot-HTTP-B`, bots periodically contacted the C&C server every 10 minutes. And in `Bot-HTTP-C`, the bots contacted the C&C server in a more stealthy way by adding a random time interval between 0 to 10 minutes on each time of visiting. Both of them conducted scanning attack on receiving the "scan" command. Bots in `Bot-IRC-A` send packets much more frequently to C&C server in the IRC session, resulting in much larger C&C flows compared to `Bot-IRC-B`. We collected traces of two P2P-based botnets, *Storm* [76] and *Waledac* [37], by running

binaries in the controlled environment.

After aligning the timestamp of each packet in botnet traces according to the time of the first packet in background traces, we mixed 3.5 consecutive days of botnet traces into the college traces by overlaying them to randomly picked client IPs in college network. We took one epoch E as 12hr so there are 7 epochs in total. The filter covers major local DNS, email servers in the college, the IP ranges of the popular service networks (e.g., MICROSOFT, GOOGLE, YAHOO, SUN, etc.), popular content distribution networks (e.g., AKAMAI) , whose IP ranges are unlikely to be used for Botnet C&Cs, and IPs of top $10,000$ *alexa* domains (corresponding to 12230 IPs).

4.4.2 Experimental Design

We structured our experiments in three parts.

First, we will study the effectiveness of our sampling algorithm. To be specific, we will investigate whether the actual sampling rate achieved by our sampling algorithm is close to the pre-defined target sampling rate. We will also present packet sampling rates for those packets that are generated by botnets.

Second, we will study the effectiveness of the cross-epoch correlation method. Particularly, we will show the scalability of the correlation method, and evaluate its effectiveness together with the packet sampling algorithm.

Finally, we will investigate the percentage of packets that fine-grained botnet detectors need to perform DPI-based analysis by using our traffic analysis framework. This helps us estimate how much our system can boost the scalability of existing botnet detection systems.

4.4.3 Experimental Results

4.4.3.1 Evaluation of Sampling Algorithm

We evaluated B-Sampling algorithm using the mixed traces with different target sampling rates (0.01, 0.025, 0.05, 0.075 and 0.1). We further perform comparison with following popular sampling algorithms:

1. FlexSample [14]: FlexSample is a state-of-the-art sampling algorithm that can be configured with different "conditions" for different purposes. FlexSample used a specific condition illustrated in Table 30 (Figure 10 in FlexSample [14]) to capture botnet packets by allocating the majority of budgets to packets related to "servers with high indegree of small flows". However, since the number of infected machines could be small in real-world, the "high fan-in" feature may not hold and thus will probably miss the botnet packets. As illustrated in Table 31, this condition causes very low sampling rates on botnet packets in our traces. Therefore, we modify the condition and only use the condition related to flow size for FlexSample. We configured FlexSample using a condition presented in Table 30 with ($size \leq 20, budget = 0.95$), which means that FlexSample uses 95% resource to capture the packets in flows with sizes smaller than 20.

2. Sketch Guided Sampling [10]: Sketch Guided Sampling (SGS) algorithm leverages a sketch for sampling, where each entry in the sketch keeps the number of packets observed for this flow. The sampling rate for a packet, which belongs to a flow with size s, is $f(s) = \frac{1}{1+\epsilon^2 s^{(2\alpha-1)}}$. According to this formula, SGS enables higher sampling rates for packets belonging to small flows (e.g., s is small), which is similar to FlexSample. However, in contrast to B-Sampling, FlexSample, Random Sampling, and Periodic Sampling, SGS cannot maintain a target sampling rate. Its actual sampling rate heavily depends on the distribution of the flow size. For example, if the flow size follows Zipfian distribution, where

Table 30: Condition for FlexSample

```
vars = 1
conditions = 1
var_1 := srcip.srcport.dstip.dstport.prot
var_1 in (0, 20]: 0.95
```

Table 31: Packet sampling rates using condition in Figure 10 in FlexSample [14]

SR_T	$SR_{I-A/B}$	$SR_{H-A/B/C}$	SR_{Storm}	$SR_{Waledac}$
0.025	0.003/0.01	0.013/0.011/0.01	0.006	0.008
0.05	0.006/0.018	0.023/0.019/0.017	0.012	0.015

large flows carry the vast majority of packets, the actual sampling rate will be low. In our experiment, we use the same parameter values in [10], where $\epsilon = 0.1$ and $\alpha = 1$.

3. Random Sampling: Random sampling algorithm is a widely deployed sampling algorithm. It gives each packet with probability r (r is the predefined sampling rate), and therefore it can well approximate the pre-defined target sampling rate. Periodic sampling algorithm is a variant of random sampling algorithm. Instead of sampling each packet with sampling rate of r, it samples the nth packet, where $n = \frac{1}{r}$.

The experimental results for B-Sampling, FlexSample, SGS, and random sampling algorithms are presented in Table 32, Table 33, Table 34, and Table 35, respectively. The analysis of experimental results is discussed as follows:

1. In each table, the first column (SR_T) reports pre-defined target sampling rates and the second column (SR_{Actual}) presents the actual sampling rates achieved by each sampling algorithm. According to the results, B-Sampling, FlexSample, random sampling, and periodic sampling algorithm can keep the actual sampling rates close to the target sampling rates. The actual sampling rates from B-Sampling algorithm are a little higher compared to the target sampling rates. It is mainly due to the fact that we set the time interval, which we used to tune the instant sampling rates for different categorizes, is large (e.g., 15 minutes).

Table 32: Packet sampling rates for B-Sampling

SR_T	SR_{Actual}	$SR_{IRC-A/B}$	$SR_{HTTP-A/B/C}$	SR_{Storm}	$SR_{Waledac}$
0.01	0.012	0.65/0.68	0.55/0.69/0.68	0.02	0.02
0.025	0.027	0.93/0.92	0.72/0.93/0.93	0.16	0.18
0.05	0.052	0.96/0.96	0.74/0.96/0.96	0.48	0.48
0.075	0.076	0.97/0.97	0.75/0.97/0.97	0.72	0.7
0.1	0.1	0.98/0.98	0.76/0.98/0.98	0.83	0.81

Table 33: Packet sampling rates for FlexSample

SR_T	SR_{Actual}	$SR_{IRC-A/B}$	$SR_{HTTP-A/B/C}$	SR_{Storm}	$SR_{Waledac}$
0.01	0.01	0.001/0.07	0.06/0.07/0.06	0.05	0.07
0.025	0.025	0.002/0.16	0.16/0.17/0.16	0.11	0.16
0.05	0.05	0.004/0.32	0.32/0.35/0.33	0.23	0.33
0.075	0.075	0.006/0.48	0.50/0.50/0.48	0.33	0.48
0.1	0.1	0.008/0.6	0.6/0.64/0.61	0.41	0.61

So the actual sampling rates for B-Sampling algorithms can be closer to the target sampling rates by further reducing such time interval. SGS algorithm is not adaptive, and hence it cannot maintain a pre-defined sampling rate. As presented in the second column in Table 34, SGS results in a high actual sampling rate (20%) in our monitored network.

2. The remaining columns report the sampling rates related to different types of botnet-related packets, where we "zoom" in the sampled packets and evaluate the actual sampling rates for packets of each botnet. For example, the column of $SR_{IRC-A/B}$ reports the actual sampling rates for packets in Bot-IRC-A and Bot-IRC-B using different sampling algorithms. We can find that B-Sampling captures a higher percentage of botnet packets, compared to FlexSample, and random sampling algorithms. For example, considering the second row (target sampling rate is 0.025), B-Sampling achieves a sampling rate of 0.93 ($SR_{IRC-A/B}$, B column) while FlexSample achieves that of 0.002 ($SR_{IRC-A/B}$, $Flex$ column) for packets in `Botnet-IRC-A`, where the C&C flows are large flows. The remaining columns report a comparison of B-Sampling and FlexSampling on the sampling rates for other botnets. As we can see, B-Sampling achieves higher sampling rate for botnet-related packets, compared to other sampling algorithms. The reason is that the feature of flow size and server in-degree are not intrinsic for

Table 34: Packet sampling rates for SGS

SR_T	SR_{Actual}	$SR_{IRC-A/B}$	$SR_{HTTP-A/B/C}$	SR_{Storm}	$SR_{Waledac}$
-	0.2	0.007/0.093	0.96/0.96/0.96	0.58	0.89

Table 35: Packet sampling rates for the random sampling algorithm

SR_T	SR_{Actual}	$SR_{IRC-A/B}$	$SR_{HTTP-A/B/C}$	SR_{Storm}	$SR_{Waledac}$
0.01	0.01	0.01/0.01	0.01/0.01/0.01	0.01	0.01
0.025	0.025	0.025/0.025	0.025/0.025/0.025	0.025	0.025
0.05	0.05	0.05/0.05	0.05/0.05/0.05	0.05	0.05
0.075	0.075	0.075/0.075	0.075/0.075/0.075	0.075	0.075
0.1	0.1	0.1/0.1	0.1/0.1/0.1	0.1	0.1

botnets and different botnets can diverse greatly regarding these features. Random sampling algorithms cannot focus on capturing packets that are likely to be related to botnets, and therefore they give the same sampling probability to all packets.

We evaluated the parameters, C and $step_{up}$, in the B-Sampling algorithm in Section 4.3.2.1. Given $SR_T = 0.05$, we report the experimental results in Table 36. The results demonstrate that the results of B-Sampling are stable over these values.

Table 36: Packet sampling rates using different parameters

C, $step_{up}$	SR_{Actual}	$SR_{I-A/B}$	$SR_{H-A/B/C}$	SR_{Storm}	$SR_{Waledac}$
10, 0.8	0.051	0.97/0.96	0.71/0.96/0.96	0.50	0.49
10, 0.5	0.051	0.96/0.95	0.61/0.96/0.95	0.51	0.51
10, 1.2	0.052	0.96/0.96	0.77/0.96/0.96	0.46	0.46
5, 1	0.052	0.96/0.96	0.74/0.96/0.96	0.46	0.46
15, 1	0.052	0.96/0.96	0.74/0.96/0.96	0.48	0.48

4.4.3.2 Evaluation of Flow Correlation

In this section, we will evaluate the detection accuracy and scalability of our cross-epoch correlation technique. We set $M = \llcorner \frac{N}{2} \lrcorner$ ($N = 7, M = 3$), which means that two hosts sharing similar communication patterns for any 3 out of 7 epochs will be labeled as suspicious.

Given SR_T and Per_{Exp}, each cell in Table 37 shows the detection rate of bots(/23) and percentage of noises(/1460) identified by Flow-Correlation using B-Sampling. The results show that Flow-Correlation can achieve high detection rate with low

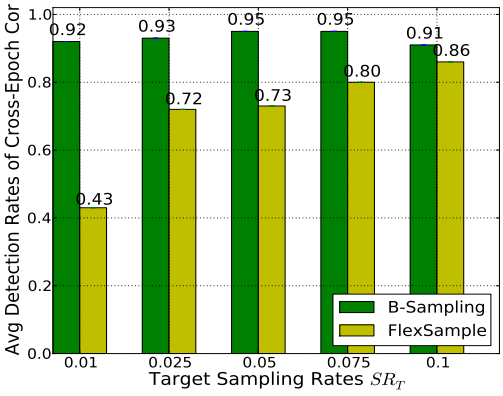


Figure 25: The average detection rate for cross-epoch correlation, over different values of Per_{Exp}

Per_{Exp}. For example, with $Per_{Exp} \geq 5\%$, for all the SR_T evaluated, Flow-Correlation can successfully identify all the bots. While for the very low Per_{Exp} (e.g., 2% and 3%), more than half of the bots were still captured. We also compared the detection rate of Flow-Correlation using B-Sampling to that of Flow-Correlation using FlexSample (in Table 38). The comparison results for using B-Sampling algorithm and using the random sampling algoirithm is presented in Table 39. Figure 25 illustrates the average detection rates over different Per_{Exp} for each target sampling rate. The comparison results show that by using B-Sampling, Flow-Correlation can achieve higher detection rate compared to both FlexSample and the random sampling algorithm.

Figure 26 presents the time consumption (in a 4G memory and 2-core CPU computer) for cross-epoch correlation and the C-Plane clustering of BotMiner as the number of C-flows increases. We configured `Birch` to run $MaxRound = 50$ to simulate the process of identifying up to Per_{Exp} suspicious hosts. The exponential time increment for C-Plane clustering of BotMiner indicates its limited scalability. The cross-epoch correlation shows linear pattern and its linear regression model is $t = 0.0035x$.

Figure 27 presents the mean and standard deviation for detection rates by Flow-Correlation with B-Sampling for different M, given Per_{Exp} (5% or 10%) for all SR_T. First, the results demonstrate the effectiveness of cross-epoch correlation. When no

Table 37: Detection rates of cross-epoch correlation using B-Sampling

SR_T	For each Per_{Exp}, TP(bots/23), FP(noises/1460)									
	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.1
0.01	48%, 0.1%	83%, 0.5%	96%, 1%	96%, 2%	100%, 3%	100%, 4%	100%, 5%	100%, 6%	100%, 6%	100%, 8%
0.025	52%, 0%	87%, 0.5%	100%, 1%	100%, 2%	100%, 3%	100%, 4%	100%, 5%	100%, 6%	100%, 7%	100%, 8%
0.05	48%, 0.1%	100%, 0.3%	100%, 1%	100%, 2%	100%, 3%	100%, 4%	100%, 5%	100%, 5%	100%, 7%	100%, 7%
0.075	48%, 0.2%	100%, 0.3%	100%, 1%	100%, 2%	100%, 3%	100%, 4%	100%, 5%	100%, 6%	100%, 7%	100%, 8%
0.1	39%, 0.3%	78%, 0.8%	100%, 1%	100%, 2%	100%, 3%	100%, 3%	100%, 5%	100%, 5%	100%, 7%	100%, 8%
1	30%, 0.5%	65%, 0.8%	96%, 1%	100%, 2%	100%, 3%	100%, 4%	100%, 5%	100%, 5%	100%, 7%	100%, 8%

Table 38: Detection rates of cross-epoch correlation using FlexSample

SR_T	For each Per_{Exp}, TP(bots/23), FP(noises/1460)									
	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.1
0.01	22%, 0.6%	30%, 2%	30%, 2%	39%, 3%	52%, 4%	52%, 5%	52%, 6%	52%, 7%	52%, 8%	52%, 8%
0.025	22%, 0.6%	39%, 1%	52%, 2%	87%, 3%	87%, 3%	87%, 5%	87%, 6%	87%, 7%	87%, 7%	87%, 8%
0.05	17%, 0.6%	43%, 1%	70%, 2%	87%, 3%	87%, 4%	87%, 4%	87%, 5%	87%, 7%	87%, 7%	87%, 7%
0.075	30%, 0.4%	57%, 1%	83%, 2%	87%, 3%	87%, 3%	87%, 4%	87%, 6%	96%, 6%	96%, 7%	96%, 8%
0.1	22%, 0.3%	65%, 1%	83%, 2%	96%, 2%	96%, 3%	100%, 4%	100%, 5%	100%, 6%	100%, 7%	100%, 8%

Table 39: Detection rates of cross-epoch correlation using the random sampling algorithm

SR_T	For each Per_{Exp}, TP(bots/23), FP(noises/1460)									
	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.1
0.01	13%, 0.6%	13%, 1%	22%, 2%	22%, 3%	22%, 4%	22%, 5%	22%, 6%	22%, 7%	35%, 8%	35%, 9%
0.025	17%, 0.7%	30%, 1%	30%, 2%	30%, 3%	39%, 4%	39%, 5%	39%, 6%	43%, 7%	43%, 8%	43%, 9%
0.05	22%, 0.6%	34%, 1%	40%, 2%	40%, 3%	40%, 4%	52%, 5%	52%, 6%	52%, 7%	52%, 8%	52%, 9%
0.075	22%, 0.6%	26%, 1%	43%, 2%	52%, 3%	52%, 4%	52%, 5%	52%, 6%	52%, 7%	52%, 8%	52%, 9%
0.1	17%, 0.7%	43%, 1%	52%, 2%	52%, 3%	52%, 4%	52%, 5%	52%, 6%	52%, 7%	52%, 8%	52%, 9%

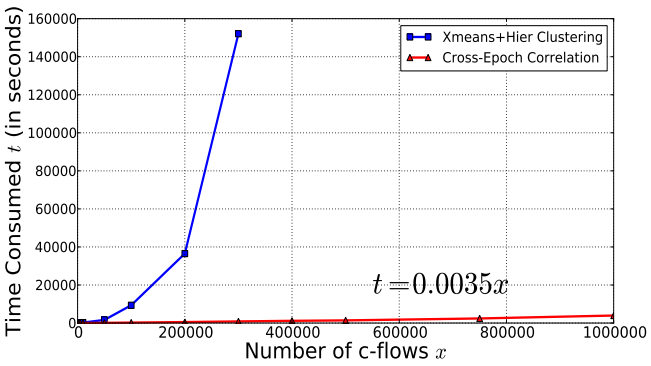


Figure 26: Scalability of cross-epoch correlation

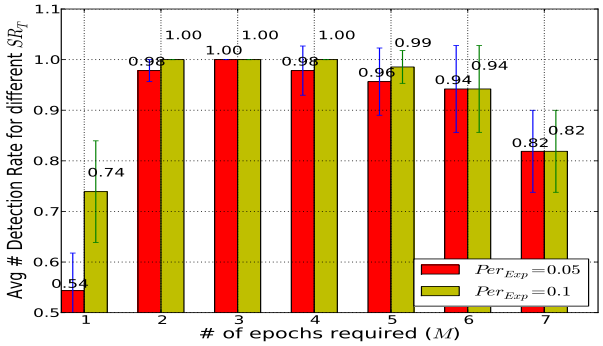


Figure 27: The average detection rate (over SR_Ts) of cross-epoch correlation using B-Sampling

Table 40: Detection rates of fine-grained detectors

SR_T	For each Per_{Exp}, TP(bots/23), FP(noises/1460)									
	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.1
0.01	48%, 0	83%, 0	96%, 0	96%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0
0.025	52%, 0	87%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0
0.05	48%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0
0.075	48%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0
0.1	39%, 0	78%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0
1	30%, 0	65%, 0	96%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0	100%, 0

cross-epoch correlation is used ($M = 1$), many legitimate IPs show stronger similarity than bots in a single epoch. Therefore, given a certain Per_{Exp}, more than 50% bots are missed. While cross-epoch correlation can effectively eliminate these legitimate IPs that show strong similarity in one epoch but do not have persistently similar patterns. For example, cross-epoch correlation with $M = 2$ can successfully detect most bots. Second, the results indicate that cross-epoch correlation is not sensitive to the value of M. For example, for $M = 3/4/5$, the cross-epoch correlation achieves similar detection rate. Such observation also indicates that $\frac{N}{2}$ is a good value for M.

4.4.3.3 Botnet Detection

Fine-grained botnet detector inspects all the packets related to suspicious IPs detected by Flow-Correlation. Using 1.5hr trace mixed with botnet traces, we evaluated the detection rate and performance of the fine-grained detector.

By analyzing the similarity among IRC messages, "IRC Message Correlation" component in our detector detected bots in `Bot-IRC-A/B`. Other bots were detected by the "Correlation" component. For example, Bots in `Bot-HTTP-B/C` trigger alerts when they scan the local network. `Bot-HTTP-A` bots trigger alerts when they make update requests. `Storm` and `Waledac` trigger alerts when they discover peers. These bots were detected by correlating such activities/alerts with corresponding pairs of IPs from Flow-Correlation. Table 40 presents the detection rates and false positive rates for the fine-grained detector for different SR_Ts and Per_{Exp}s. The corresponding cells in Table 42 present the percentage of packets that our fine-grained detector needs

Table 41: Performance of fine-grained detector (in seconds)

	With Flow-Corr ($Per_E = 5\%, M = 3$)						direct
SR_T	0.01	0.025	0.05	0.075	0.1	1	
Per of Pkts	1.7%	2.9%	2.1%	3%	4.3%	2%	100%
$Time$	33s	39s	35s	40s	49s	33s	858s

to inspect. For most combinations of SR_T and Per_{Exp}, our framework can reduce traffic volume by more than 90% for fine-grained detector but still keep high detection rates and low false positives. For example, for $SR_T = 0.01$ and $Per_{Exp} = 0.05$, the fine-grained detector can detect all bots with false positive of 0, and it only needs to focus on 1.7% percentage of packets.

Table 41 presents the performance comparison, including the percentage of packets inspected and the processing time of the fine-grained detector in two situations: i) the detector is directly applied, ii) the detector is applied with Flow-Correlation and B-Sampling ($Per_{Exp} = 0.05$ and $M = 3$). By using Flow-Correlation, fine-grained detector to reduce 95% time to process off-line traces, indicating a great workload reduction in real time.

Table 42: The percentage of packets investigated by fine-grained detectors based on DPI

SR_T	For each Per_{Exp}, Percentage of Packets									
	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.1
0.01	0.1%	0.4%	1%	1.5%	1.7%	3.3%	3.5%	4.1%	4.2%	5%
0.025	0.2%	0.7%	1.2%	2.6%	2.9%	3.5%	3.8%	4%	4.2%	6%
0.05	0.6%	0.6%	1%	1.8%	2.1%	2.2%	2.7%	2.7%	3.3%	3.5%
0.075	0.6%	0.6%	2%	3%	3.2%	3.8%	4.5%	4.5%	4.4%	5%
0.1	0.2%	0.9%	1.3%	3.7%	4.3%	4.3%	4.6%	4.6%	5.5%	6.2%
1	0.7%	0.6%	1%	1.7%	1.9%	3.4%	3.3%	3.3%	4.9%	6.1%

4.5 Discussion

To answer the question "how high speed networks our approach can handle?", we consider the performance of two key components, B-Sampling and cross-epoch correlation. B-Sampling is intended to be implemented with hardware support, where we can design the Counting-Sketch and Sampling-Sketch in fast memory (e.g., SRAM)

while the SID and PSPC in slow memory (e.g., DRAM). The system can periodically but parallel read the data from SRAM to DRAM for identifying synchronized hosts and computing sampling probabilities, and then write the sets of IPs to SRAM. And for Counting-Sketch, recent study has shown the hardware implementation of a specific hash function with a throughput of over 10Gbps [19], indicating the potential performance of 10Gbps of B-Sampling with hardware implementation. Given an expected time consumption of 2hr for cross-epoch correlation, the linear model $t = 0.0035x$ (in seconds) implies 2M C-flows. If we assume the number of C-flows is proportional to the traffic volume (e.g., 200K C-flows in our experiment is corresponding to 200Mbps), 2M C-flows correspond to a network with speed of 2Gbps. Since 2Gbps is less than the potential performance of 10Gbps of B-Sampling, such results indicate that our approach can be used in 2Gbps networks (e.g., campus backbone networks) and has the potential to be deployed in faster network as the expected time consumption of cross-epoch correlation increases.

Because of our assumptions on the persistent use of coordinated C&Cs in a botnet, any evasion attempts that violate our assumptions will likely succeed if the botmaster knows our algorithms, similar to any evasion attacks against an IDS. Bots may intentionally manipulate their communication patterns to decrease sampling probabilities or evade cross-epoch correlation. For example, bots can randomize communication patterns (e.g., number of packets per flow) to evade the syn-client/server detection. One potential solution is to dynamically tune the parameters used for identifying syn-servers and syn-clients for each round (e.g., randomly select $\frac{1}{4}$, $\frac{3}{4}$ quantiles or medium value of variances for identifying syn-server, and choose R and C from a pre-defined set of values/ranges for identifying syn-clients). Another solution is for B-Sampling to incorporate information from other systems. For example, we can set a category of IPs in rouge networks [16] or malicious fast-flux networks, which are likely related to

botnets, to sample more related packets. For cross-epoch correlation, we can incorporate more detection features (e.g., using packet payload information for some tight clusters to do light-weight content checking) to make the evasion more difficult. Due to the nature of the arms race in existing intrusion detection and evasion practice, we should always study better and more robust techniques as a defender. Combining different complementary detection techniques to make the evasion harder is one possible future direction. We leave a deeper and more extensive study to handle these evasion attempts as future work.

4.6 Summary

Botnet detection in high-speed and high-volume networks is a challenging problem. Given the severity of botnets and the growing interest from ISPs to defend against botnets, research on botnet detection in high-speed and high-volume networks is important. In this chapter, we have described a solution to this problem, which includes a botnet-aware adaptive packet sampling algorithm and a scalable spatial-temporal flow correlation approach. The adaptive packet sampling technique uses network characteristics of botnet C&Cs to capture more packets related to bots and adaptively tune the sampling probabilities to keep a target sampling rate. The flow correlation approach exploits the essential properties of botnets and detects bots by identifying hosts with persistently similar communication patterns. Evaluation using real-world network traces shows that our proposed solution yields good performance. The sampling algorithm can capture more botnet packets in comparison to pre-defined sampling rate and outperforms the state-of-the-art adaptive sampling algorithms. Based on the sampled packets, the correlation algorithm can successfully and scalably pinpoint various types of bots (including IRC-based, HTTP-based, and P2P-based). This approach will help the fine-grained botnet detectors to focus on inspecting packets from a smaller amount of suspicious traffic, thus allowing them to

operate on increasingly more high-speed networks.

CHAPTER V

CONCLUSION AND FUTURE WORK

5.1 Conclusion

Serving as platforms responsible for the vast majority of large-scale and coordinated cyber attacks, botnets have been recognized as one of the most serious threats against Internet security. Botnet detection therefore becomes fundamentally important. A number of network-based botnet detection systems have been consequently proposed. However, these systems are faced with two new challenges, which may significantly limit their practical use. First, botnets tend to be increasingly stealthy on performing cyber attacks. As a result, botnet detection systems that rely on the observation of attacks in network traffic may lose their effectiveness. Second, the volume of network traffic grows fast, requiring botnet detection systems to process a huge amount of information efficiently. However, existing botnet detection systems mainly rely on deep packet inspection for detection, thereby suffering from limited scalability. My dissertation focuses on addressing these two challenges by buidling three novel systems.

First, we have designed a system, named *ARROW* [47], to detect drive-by download attacks, which represent one of the most significant and popular methods for botnet infection. *ARROW* boosts the effectiveness of existing detection systems by detecting a large number of stealthy drive-by download attacks that are missed by them. By designing this system, we made the following contributions:

- We provide a method to identify malware distribution networks from millions of individual drive-by download attacks.
- By *correlating* individual drive-by download attacks, we propose a novel method

to generate regular expression signatures of central servers of malware distribution networks to detect drive-by downloads.

- We have built a system to automatically generate regular expression signatures of central servers of MDNs and evaluate the effectiveness of these signatures.

The next piece of our work concentrates on building system to detect P2P botnets. Our system aims to effectively detect P2P botnets, even in the case that their attacks cannot be observed in the network traffic. Our system is also capable of detecting P2P botnets even if the bot-infected hosts are running legitimate P2P applications simultaneously. Our work made multiple contributions:

- We propose a novel flow-clustering-based analysis approach to identify and profile hosts that are engaging in P2P communications.

- We design new features and algorithms to detect P2P botnets independent of the observation of botnets' malicious activities.

- We have developed a scalable detection system based on efficient clustering algorithm and load-balance workload distribution. We also evaluate the effectiveness and efficiency of our system.

The third piece of our work focuses on boosting the scalability of existing network-based botnet detection systems. Our system can effectively and efficiently identify a small portion of hosts that are likely to be bots, and then suggest existing detection systems to perform DPI-based analysis on packets related to those suspicious hosts instead of all hosts. Consequently, the existing detection systems only need to investigate a small portion of network traffic and their scalability is thus boosted. In this piece of work, we made the following contributions:

- We introduce a botnet-aware adaptive packet sampling technique based on group similarity, an intrinsic characteristic of botnets, to sample packets that are likely related to C&C communications with high probability.

- We propose a new scalable spatial-temporal correlation method to identify hosts in a network that share persistently similar communication patterns, which is one of the main characteristics of botnets.

- We have implemented a proof-of-concept version of our system, and evaluated it using real-world legitimate and botnet-related network traces. Our experimental results show that the proposed approach is scalable and can effectively detect bots with few false positives, which can be further reduced by fine-grained botnet detection systems.

5.2 Future Work

While a number of problems to build effective and scalable network-based botnet detection systems have been addressed in this dissertation, our research work also suggested new research directions discussed as follows:

- **Botnet Detection Based on Heterogeneous Information Correlation** The severity of cyber-attacks has accelerated the deployment of various detection and monitoring systems, such as antivirus software, DNS monitoring system, network flow collection system, and spam traps. A growing body of information collected from diverse systems makes possible the opportunity to study and detect botnets from multiple perspectives. Therefore, building an integrated detection system that can correlate information from these monitoring systems is a promising direction. However, it is a challenging task because of the heterogeneous nature and sheer volume of the information. We believe our work on leveraging DNS and network flow for P2P botnet detection (Chapter 3)

and designing scalable botnet detection systems (Chapter 4) will provide useful information to explore this direction.

- **Botnet Mitigation** A natural step after botnet detection is mitigation. How to strike the balance between minimizing the botnet threats and maintaining the usability of bot-infected hosts remains a challenging problem. Designing novel mitigation techniques that can integrate mitigation strategies from different levels (e.g., process, system, and network) would be a promising direction.

- **Botnet Detection in Emerging Infrastructures** Driven by incredible profits, attackers are always actively expanding their malicious ecosystems by contaminating emerging infrastructures. A number of botnets that operate on smart phones have been identified, introducing great threats against personal data and cellular networks. Stuxnet [63], a bot targeted industrial control systems, was identified in 2009 and reportedly interfered with the operation of Iran's nuclear facility. As a consequence, leveraging our knowledge learnt from detecting bontets in Internet to proactively defeat botnets and other malware in emerging infrastructures will demonstrate great importance and deserves exploration in the future.

5.3 Closing Remarks

This dissertation addressed important research problems in botnet detection. Specifically, our work focused on addressing new challenges presented to existing botnet detection systems, which are introduced by the evolution of both botnets and the Internet. We first built a novel system to detect drive-by downloads, which serve as the primary way for botnet infection. Further, we proposed a new P2P botnet detection system. We finally presented a new traffic analysis framework that can boost the scalability of existing botnet detection systems. Together, our contributions boost the effectiveness and scalability of existing botnet detection systems. Due to the nature

of arms race in existing security area, attackers and their attacks are always actively evolving, which requires us to identify and act on emerging challenges. We believe the solutions presented in this dissertation will be helpful for future research in this direction.

REFERENCES

[1] "ARGUS: Auditing Network Activity." `http://www.qosient.com/argus/`.

[2] "AutoIt Script." `http://www.autoitscript.com/autoit3/index.shtml`.

[3] "DNSCAP: DNS traffic capture utility." `https://www.dns-oarc.net/tools/dnscap`.

[4] "TCP SYN Flooding Attacks and Common Mitigations." RFC 4987.

[5] "CAIDA Analysis of Code-Red." `http://www.caida.org/research/security/code-red/`, 2001.

[6] "Hackers Use Twitter API To Trigger Malicious Scripts." http://blog.unmaskparasites.com/2009/11/11/hackers-use-twitter-api-to-trigger-malicious-scripts/, 2009.

[7] "Zeus Spreading Through Drive-by Download." http://www.scmagazine.com/zeus-spreading-through-drive-by-download/article/158691/, 2009.

[8] "Drive-By Download Attacks Were the Biggest Online Threat Last Month." http://news.softpedia.com/news/Drive-By-Download-Attacks-Were-the-Biggest-Online-Threat-Last-Month-170525.shtml, 2010.

[9] A. Karasaridis, B. Rexroad, and D. Hoeflin, "Wide-Scale Botnet Detection and Characterization," in *Proc. USENIX HotBots*, 2007.

[10] A. Kumar and J. Xu, "Sketch Guided Sampling – Using On-Line Estimates of Flow Size for Adaptive Data Collection," in *Proc. IEEE INFOCOM*, 2006.

[11] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy, "A Crawler-based Study of Spyware on the Web," in *Proc. NDSS*, 2006.

[12] A. Ramachandran, N. Feamster, and D. Dagon, "Revealing Botnet Membership Using DNSBL Counter-Intelligence," in *Proc. USENIX SRUTI*, 2006.

[13] A. Ramachandran, N. Feamster, and S. Vempala, "Filtering Spam with Behavioral Blacklisting," in *Proc. ACM CCS*, 2007.

[14] A. Ramachandran, S. Seetharaman, and N. Feamster, "Fast Monitoring of Traffic Subpopulations," in *Proc. ACM IMC*, 2008.

[15] A.W. Moore and D. Zuev, "Internet Traffic Classification Using Bayesian Analysis Techniques," in *Proc. ACM SIGMETRICS*, 2005.

[16] B. Stone-Gross, A. Moser, C. Kruegel, E. Kirda, and K. Almeroth, "FIRE: FInding Rogue nEtworks," in *Proc. ACSAC*, 2009.

[17] 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 *Proc. ACM CCS*, 2009.

[18] B. Stone-Gross, R. Abman, R. Kemmerer, C. Kruegel, D. Steigerwald, and G. Vigna, "The Underground Economy of Fake Antivirus Software," in *Proc. WEIS*, 2011.

[19] B. Yang, R. Karri, and D.A. McGrew, "Divide and Concatenate: An Architectural Level Optimization Technique for Universal Hash Functions," in *Proc. of the Design Automation Conference*, 2004.

[20] B.B. Kang, E.C. Tin, and C.P. Lee , "Towards Complete Node Enumeration in a Peer-to-Peer Botnet," in *Proc. ACM ASIACCS*, 2009.

[21] Brad Miller, Paul Pearce, Chris Grier, Christian Kreibich, and Vern Paxson, "What's Clicking What? Techniques and Innovations of Today's Clickbots," in *Proc. DIMVA*, 2011.

[22] C. Hu, S. Wang, J. Tian, B. Liu, Y. Cheng, and Y. Chen, "Accurate and Efficient Traffic Monitoring Using Adaptive Non-linear Sampling Method," in *Proc. IEEE INFOCOM*, 2008.

[23] C. Livadas, R. Walsh, D. Lapsley, and W. T. Strayer, "Using Machine Learning Techniques to Identify Botnet Traffic," in *Proc. IEEE WoNS*, 2006.

[24] C. Mullaney, "Android.Bmaster: A Million-Dollar Mobile Botnet." http://www.symantec.com/connect/blogs/androidbmaster-million-dollar-mobile-botnet, 2012.

[25] C. Seifert and R. Steenson, "Capture - Honeypot Client (Capture-HPC)." https://projects.honeynet.org/capture-hpc, 2006.

[26] C. Seifert, I. Welch, and P. Komisarczuk, "HoneyC - The Low-Interaction Client Honeypot," in *Proc. NZCSRCS*, 2007.

[27] C. Seifert, R. Steenson, T. Holz, B. Yuan, and M. A. Davis, "Know Your Enemy: Malicious Web Servers." http://www.honeynet.org/papers/mws/, 2007.

[28] C. Whittaker, B. Ryner, and M. Nazif, "Large-Scale Automatic Classification of Phishing Pages," in *Proc. NDSS*, 2010.

[29] C. Zou, W. Gong, and D. Towsley, "Code Red Worm Propagation Modeling and Analysis," in *Proc. ACM CCS*, 2002.

[30] D. Stutzbach and R. Rejaie, "Understanding Churn in Peer-to-Peer Networks," in *Proc. ACM IMC*, 2006.

[31] F. Yu, Y. Xie, and Q. Ke, "SBotMiner: Large Scale Search Bot Detection," in *Proc. ACM WSDM*, 2010.

[32] G. Bartlett, J. Heidemann, C. Papadopoulos, and J. Pepin, "Estimating P2P Traffic Volume at USC," Tech. Rep. ISI-TR-2007-645, USC/Information Sciences Institute, 2007.

[33] G. Gu, J. Zhang, and W. Lee, "BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic," in *Proc. NDSS*, 2008.

[34] G. Gu, M. Sharif, X. Qin, D. Dagon, W. Lee, and G. Riley, "Worm Detection, Early Warning and Response Based on Local Victim Information," in *Proc. ACSAC)*, 2004.

[35] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, "BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation," in *Proc. USENIX Security*, 2007.

[36] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection," in *Proc. USENIX Security*, 2008.

[37] G. Sinclair and C. Nunnery, and B. B. Kang, "The Waledac Protocol: The How and Why," in *Proc. Intl. Conf. Malicious and Unwanted Software*, 2009.

[38] G. Stringhini, T. Holz, B. Stone-Gross, C. Kruegel, and G. Vigna, "BOTMAGNIFIER: Locating Spambots on the Internet," in *Proc. USENIX Security*, 2011.

[39] J. Cheng, "Report: Botnets Sent Over 80% of All June Spam." http://arstechnica.com/security/news/2009/06/report-botnets-send-over-80-of-all-spam-in-june.ars, 2009.

[40] J. Goebel and T. Holz, "Rishi: identify bot contaminated hosts by IRC nickname evaluation," in *Proc. USENIX HotBots*, 2007.

[41] J. Nazario, "PhoneyC: A Virtual Client Honeypot," in *Proc. USENIX LEET*, 2009.

[42] J. Newsome, B. Karp, and D. Song, "Polygraph: automatically generating signatures for polymorphic worms," in *Proc. IEEE Symposium on Security and Privacy*, 2005.

[43] J. P. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi, "deSEO: Combating Search-Result Poisoning," in *Proc. USENIX Security*, 2011.

[44] J. P. John, F. Yu, Y. Xie, M. Abadi, and A. Krishnamurthy, "Searching the Searchers with SearchAudit," in *Proc. USENIX Security*, 2010.

[45] J. R. Binkley and S. Singh, "An Algorithm for Anomaly-based Botnet Detection," in *Proc. USENIX SRUTI*, 2006.

[46] J. W. Stokes, R. Andersen, C. Seifert, and K. Chellapilla, "WebCop: Locating Neighborhoods of Malware on the Web," in *Proc. USENIX LEET*, 2010.

[47] J. Zhang, C. Seifert, J. W. Stokes, and W. Lee, "ARROW: Generating Signatures to Detect Drive-By Downloads," in *Proc. WWW*, 2011.

[48] J. Zhang, R. Perdisci, W. Lee, U. Sarfraz, and X. Luo, "Detecting Stealthy P2P Botnets Using Statistical Traffic Fingerprints," in *Proc. IEEE DSN-DCCS*, 2011.

[49] J. Zhang, X. Luo, R. Perdisci, G. Gu, W. Lee, and N. Feamster, "Boosting the Scalability of Botnet Detection using Adaptive Traffic Sampling," in *Proc. ACM ASIACCS*, 2011.

[50] K. Wang, and S. J. Stolfo, "Anomalous Payload-based Network Intrusion Detection," in *Proc. RAID*, 2004.

[51] L. Lu, V. Yegneswaran, P. Porras and W. Lee, "BLADE: An Attack-Agnostic Approach for Preventing Drive-By Malware Infections," in *Proc. ACM CCS*, 2010.

[52] Lee, C. P., *Framework for Botnet Emulation and Analysis*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, Nov. 2008.

[53] M. A. Rajab, L. Ballard, P. Marvrommatis, N. Provos, and X. Zhao, "The Nocebo Effect on the Web: An Analysis of Fake Anti-Virus Distribution," in *Proc. USENIX LEET*, 2010.

[54] M. Antonakakis, R. Perdisci, W. Lee, N. Vasiloglou II, and D. Dagon, "Detecting Malware Domains at the Upper DNS Hierarchy," in *Proc. USENIX Security*, 2011.

[55] M. Bailey, J. Oberheide, J. Andersen, Z. Morley Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in *Proc. RAID*, 2007.

[56] M. Cova, C. Kruegel, and G. Vigna, "Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code," in *Proc. ACM WWW*, 2010.

[57] M. Cova, C. Leita, O. Thonnard, A. Keromytis, and M. Dacier, "An Analysis of Rogue AV Campaigns," in *Proc. RAID*, 2010.

[58] M. Halkidi, Y. Batistakis, and M. Vazirgiannis, "On Clustering Validation Techniques," *J. Intell. Inf. Syst.*, vol. 17, no. 2-3, pp. 107–145, 2001.

[59] M.A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis, "A Multifaceted Approach to Understanding the Botnet Phenomenon," in *Proc. ACM IMC*, 2006.

[60] M.P. Collins and M. K. Reiter, "Finding Peer-to-Peer File Sharing Using Coarse Network Behaviors," in *Proc. ESORICS*, 2006.

[61] N. Anderson, "Vint Cerf: One Quarter of All Computers Part of a Botnet." http://arstechnica.com/old/content/2007/01/8707.ars, 2007.

[62] N. Daswani and M. Stoppelman, "The Anatomy of Clickbot.A," in *Proc. USENIX HotBots*, 2007.

[63] N. Falliere, L. O. Murchu, and E. Chien, "W32. Stuxnet Dossier," tech. rep., Symantec Security Response, 2011.

[64] N. Provos, J. McClain, and K. Wang, "Search Worms," in *Proc. ACM WORM*, 2006.

[65] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All Your Iframes Points to US," in *Proc. USENIX SECURITY*, 2008.

[66] P. Porras, H. Saidi, and V. Yegneswaran, "A multi-perspective analysis of the Storm (Peacomm) worm," in *Computer Science Laboratory, SRI International, Technical Report*, 2007.

[67] P. Porras, H. Saidi, and V. Yegneswaran, "Conficker C Analysis." http://mtc.sri.com/Conficker/addendumC/index.html, 2009.

[68] R. Lemos, "Bot software looks to improve peerage." http://www.securityfocus.com/news/11390, 2006.

[69] R. Perdisci, I. Corona, D. Dagon, and W. Lee, "Detecting Malicious Flux Service Networks through Passive Analysis of Recursive DNS Traces," in *Proc. ACSAC*, 2009.

[70] R. Perdisci, W. Lee, and N. Feamster, "Behavioral Clustering of HTTP-based Malware and Signature Generation using Malicious Network Traces," in *Proc. USENIX NSDI*, 2010.

[71] S. Hao, N. A. Syed, N. Feamster, A. G. Gray, and S. Krasser, "Detecting Spammers with SNARE: Spatio-temporal Network-level Automatic Reputation Engine," in *Proc. USENIX Security*, 2009.

[72] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov, "BotGrep: Finding P2P Bots with Structured Graph Analysis," in *Proc. USENIX Security*, 2010.

[73] S. Sen, O. Spatscheck, and D. Wang, "Accurate, Scalable In-Network Identication of P2P Traffic Using Application Signatures," in *Proc. WWW*, 2004.

[74] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated Worm Fingerprinting," in *Proc. USENIX OSDI*, 2004.

[75] S. Staniford, V. Paxson, and N. Weaver, "How to 0wn the Internet in Your Spare Time," in *Proc. USENIX Security*, 2002.

[76] S. Stover, D. Dittrich, J. Hernandez, and S. Dietrich, "Analysis of the Storm and Nugache trojans: P2P is here," in *USENIX; login, vol. 32, no. 6*, 2007.

[77] Symantec, "The State of Spam, A Monthly Report September 2008." `http://eval.symantec.com/mktginfo/enterprise/other_resources/b-state_of_spam_report_09-2008.en-us.pdf`, 2008.

[78] T. F. Yen and M. K. Reiter, "Traffic Aggregation for Malware Detection," in *Proc. DIMVA*, 2008.

[79] T. F. Yen and M. K. Reiter, "Are Your Hosts Trading or Plotting? Telling P2P File-Sharing and Bots Apart," in *Proc. ICDCS*, 2010.

[80] T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling, "Measuring and Detecting Fast-Flux Service Networks," in *Proc. NDSS*, 2008.

[81] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling, "Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm," in *Proc. USENIX LEET*, 2008.

[82] T. Karagiannis, A.Broido, M. Faloutsos, and Kc Claffy, "Transport layer identification of P2P traffic," in *Proc. ACM IMC*, 2004.

[83] T. Karagiannis and K. Papagiannaki, and M. Faloutsos , "BLINC: Multilevel Traffic Classification in the Dark," in *Proc. ACM SIGCOMM*, 2005.

[84] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An Efficient Data Clustering Method for Very Large Databases," in *Proc. ACM SIGMOD*, 1996.

[85] The Honeynet Project, "Know Your Enemy: Fast-Flux Service Networks; An Ever Changing Enemy." http://www.honeynet.org/papers/ff/, 2007.

[86] U. Bayer, P. Milani, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, Behavior-Based Malware Clustering," in *Proc. NDSS*, 2009.

[87] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha, "An Architecture for Generating Semantics-Aware Signatures," in *Proc. USENIX SECURITY*, 2005.

[88] W. Fang and L. Peterson, "Inter-AS Traffic Patterns and Their Implications," in *IEEE Global Internet Symposium*, 1999.

[89] W. T. Strayer, R. Walsh, C. Livadas, and D. Lapsley, "Detecting Botnets with Tight Command and Control," in *Proc. IEEE LCN*, 2006.

[90] X. Hu, M. Knysz, and K. Shin, "RB-Seeker: Auto-detection of Redirection Botnets," in *Proc. NDSS*, 2009.

[91] 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 *Proc. NDSS*, 2006.

[92] Y. Xie, F. Yu, K. Achan, R. Panigraphy, G. Hulten, and I. Osipkov, "Spamming Botnets: Signatures and Characteristics," in *Proc. ACM SIGCOMM*, 2008.

[93] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Applications," in *Proc. ACM IMC*, 2004.

[94] Y Zhao, Y. Xie, F. Yu, Q. Ke, and Y. Yu, "BotGraph: Large Scale Spamming Botnet Detection," in *Proc. USENIX NSDI*, 2009.

[95] Z. Li, A. Goyal, Y. Chen, and A. Kuzmanovic, "Measurement and Diagnosis of Address Misconfigured P2P Traffic," in *IEEE INFOCOM*, 2010.

[96] Z. Li, A. Goyal, Y. Chen, and V. Paxson, "Automating Analysis of Large-Scale Botnet Probing Events," in *Proc. ACM ASIACCS*, 2009.

[97] Z. Li, M. Sanghi, B. Chavez, Y. Chen, and M. Kao, "Hamsa: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience," in *Proc. IEEE Symposium on Security and Privacy*, 2006.