PROTECTING NETWORKED SYSTEMS FROM MALWARE THREATS

A Dissertation

by

SEUNG WON SHIN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Guofei Gu
Co-Chair of Committee,	Narasimha Reddy
Committee Members,	Alex Sprintson
	Shuguang Cui
Head of Department,	Chanan Singh

August 2013

Major Subject: Computer Engineering

ABSTRACT

Currently, networks and networked systems are essential media for us to communicate with other people, access resources, and share information. Reading (or sending) emails, navigating web sites, and uploading pictures to social medias are common behaviors using networks. Besides these, networks and networked systems are used to store or access sensitive or private information. In addition, major economic activities, such as buying food and selling used cars, can also be operated with networks. Likewise, we live with networks and networked systems.

As network usages are increasing and popular, people face the problems of network attacks. Attackers on the networks can steal people's private information, mislead people to pay money for fake products, and threaten people, who operate online commercial sites, by bothering their services. There are much more diverse types of network attacks that torture many people using networks, and the situation is still serious.

The proposal in this dissertation starts from the following two research questions: (i) what kind of network attack is prevalent and how we can investigate it and (ii) how we can protect our networks and networked systems from these attacks. Therefore, this dissertation spans two main areas to provide answers for each question.

First, we analyze the behaviors and characteristics of large-scale bot infected hosts, and it provides us new findings of network malware and new insights that are useful to detect (or defeat) recent network threats. To do this, we investigate the characteristics of victims infected by recent popular botnet - Conficker, MegaD, and Srizbi. In addition, we propose a method to detect these bots by correlating network and host features.

Second, we suggest new frameworks to make our networks secure based on the new network technology of Software Defined Networking (SDN). Currently, SDN technology is considered as a future major network trend, and it can dynamically program networks as we want. Our suggested frameworks for SDN can be used to devise network security applications easily, and we also provide an approach to make SDN technology secure.

ACKNOWLEDGEMENTS

This is to thank Dr. Guofei Gu and Dr. Narasimha Reddy for their thoughtful help and support during this stint. With their careful advising, patience, and intellectual support, I have been able to complete my Ph.D research. I would also express my thanks to my wife for her support, and my mother who has been an inspiration as always. My committee has also been pivotal in steering my work and this is to thank Dr. Spritson, Dr. Cui, and Dr. Hu. My mentors at SRI International have also been greatly helpful in shaping my work and thoughts and inputs are really appreciated. This is to thank Phil Porras and Vinod Yegneswaran. I have also had the pleasure of sharing my office and thoughts with guys who are more like friends than officemates. Thank you Zhaoyan, Chao, Jialong, and Haopei. I would also like to thank Tammy Carda for her support at all times. Last but not the least, I would like to thank all my colleagues, teachers, mentors, roommates, and friends, who have in some way touched my life and shaped this dissertation. I am indebted to them.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Currently, networks and networked systems are intertwined with our lives. A student sends an email to deliver his report to a professor; a man shares his recent photos with his parents, who live on different continents, through a social network service; and a manager, who owns his e-commerce site, sells new basketball shoes through the Internet. All of these activities can be observed easily around us, and the networks and networked systems make our lives more productive and efficient.

As many devices that we use daily are connecting to network systems, we face new threats that attack these systems. Attackers conduct different malicious operations to steal money or just to show off their abilities. For example, an attacker may reveal hosts that can be reached through a network and have certain vulnerabilities, and can steal sensitive information, such as bank account numbers and confidential documents, from the hosts. Moreover, an attacker can make web services fail in handling requests from normal clients by sending a great number of fake network packets to the services

Currently, such network attacks are major hurdles in building network environments that can make our lives better. If these attacks become more widespread and common, people will hesitate to use network services. This will be a serious obstacle in the development of networks in our lives.

Our research is motivated by this problem, and we ultimately intend to stop these network attacks and make our networks more secure. To achieve this goal, we first need to know about network attacks; thus, we must investigate how attackers operate, infect hosts, and conduct malicious operations. By analyzing network attacks, we can understand them, and this represents the starting point of research with the goal

of defending our networks. However, it is very hard to survey all network threats because they are so diverse. Therefore, in this research, we have decided to focus on the analysis of some specific, serious network threats. Although this limits the scope of the study, we believe that this can cover mainstream network threats and it can help us reduce the effects of the most serious ones.

With the considerations mentioned above, we have selected *bot malware* as our main target. Bot malware infects victim hosts, and it makes them be controlled by a malware writer (we call this writer a bot-master). The bot-master controls infected hosts to conduct different malicious operations. For example, a bot-master can steal sensitive information from infected hosts, and he performs distributed denial of service (DDoS) attacks to crash network services by controlling them. Even worse, a bot-master can rent these infected hosts to another people who want to engage in malicious operations. Renting bot-infected hosts can help a bot-master to make money, and this is a great motivation to malware writers. Thus, recently, many serious network threats have been initiated by bot-infected hosts controlled by a bot-master.

Some previous studies have investigated bot malware, mostly focusing on analysis of bot malware binaries, which is important work [91, 102]. However, since bots have infected so many victims and have so much potential to damage the Internet, they deserve much deeper study. By analyzing the state-of-the-art botnet, we can gain more knowledge of current malware, for example, how it differs from previous generation malware and whether such differences represent future trends. Such deeper investigations could also provide new insights into developing new detection and defense mechanisms for current and future malware.

In this context, we first provide deep analysis on bot-infected victims in Chapter I. In this chapter, we analyze infected victims by looking at recent popular bot

malware - Conficker [96], Srizbi [95], and Megad [81], and we provide our findings and new insights based on them. This chapter also compares the results of analysis concerning each bot to provide more interesting results. We perform an in-depth cross-analysis of different botnet types and show what similarities and differences exist between them. Slightly contradictory to the hypothesis stated above, we report that both types of botnets have a large portion of overlapped victims, and the overall victim distributions in IPv4 space are quite similar. At the same time, they show several distinct characteristics. To obtain a fine-grained understanding of these similarities and differences, we further perform an in-depth set of large-scale passive and active measurement studies from several perspectives, such as IP geographical location, IP address population/density, network openness, and IP address dynamism. The results reveal many interesting characteristics that could help explain the similarities and differences between the two botnet infection types. More interestingly, we demonstrate empirically that even if we only know some information about one botnet (e.g., past botnet data), we can predict unknown victims of another botnet (e.g., a future emerging botnet) with reasonably high accuracy, given that both botnets use the same infection type (e.g., web-exploit). This sheds light on the promising power of cross-analysis and cross-prediction.

Based on the results of the analysis, we devise a promising bot malware detection system, and its prototype, called *EFFORT*, is presented in the second part (Chapter II). This system detects bot malware by correlating the network- and host-level characteristics of bot malware. Based on the intrinsic characteristics of bots, we propose a multi-module approach to correlate information from different host- and network-level elements, and design a multilayered architecture to efficiently coordinate modules to perform heavy monitoring only when necessary. We implement our proposed system and evaluated real-world benign and malicious programs run-

ning on several different real-life office and home machines for several days. The final results show that our system can detect all 15 real-world bots (e.g., Waledac, Storm) with low false positives (less than 0.7%) and minimal overhead. We believe that EFFORT raises the bar in the malware battle, and therefore this host-network integrated design represents a timely effort in the right direction.

Understanding network threats and devising a malware detection system are both important and necessary endeavors, which make many contributions when it comes to removing network threats. However, they also have some limitations. We need to keep investigating new network threats, and we should keep designing new defense systems for new threats that emerge. Considering the relationships between attacker and defender, this kind of approach may not be avoidable. Attackers keep developing new attack methods, and defenders keep finding ways to defeat them. Is it possible to stop this arms race, or at least reduce the burden of defenders?

It will be very hard to stop the progression of malware attacks and defense. However, if we change our network environments with a focus on security, this can help defenders to devise more effective and efficient systems to prevent attacks. This is the second goal of this thesis; thus, we try to provide a new network architecture that affords better security services than existing network environments.

Changing network architectures is not easy, and thus we need to identify an efficient way do this. In this context, we find that software-defined networking (SDN) is a good candidate technology, as it enables us to design new network environments easily. SDN has quickly emerged as a new promising technology for future networks. With the separation of the control plane from the data plane, thereby enabling the easy addition of new, creative, powerful network functions/protocols, SDN has attracted significant attention from both academia and industry. In academia, since the publication of OpenFlow [62], which is a key component in realizing the SDN

concept, many research ideas based on SDN/OpenFlow have been proposed [67] [73] [87] [9] [38] [85] [15] [88]. In industry, SDN is widely considered to be the new paradigm for future networks, and many companies are deploying or plan to deploy such technology in order to strengthen their network architectures, reduce operational costs, and enable new network applications/functions.

Likewise, SDN has potential as a future networking technology, and we have decided to use it to design our new network architecture with an emphasis on security. Thus, we introduce a new security application development framework called *FRESCO*, which represents our first step in making networks more secure. The aim of FRESCO is to address several key issues that can accelerate the composition of new SDN-enabled security services. It exports a scripting API that enables security practitioners to code security monitoring and threat detection logic as modular libraries. These modular libraries represent the elementary processing units in FRESCO, and may be shared and linked together to provide complex network defense applications.

FRESCO currently includes a library of 16 commonly reusable modules, which we intend to expand over time. Ideally, more sophisticated security modules can be built by connecting basic *FRESCO modules*. Each FRESCO module includes five interfaces: (*i*) input, (*ii*) output, (*iii*) event, (*iv*) parameter, and (*v*) action. By simply assigning values to each interface and connecting necessary modules, a FRESCO developer can replicate a range of essential security functions, such as firewalls, scan detectors, attack deflectors, or IDS detection logic.

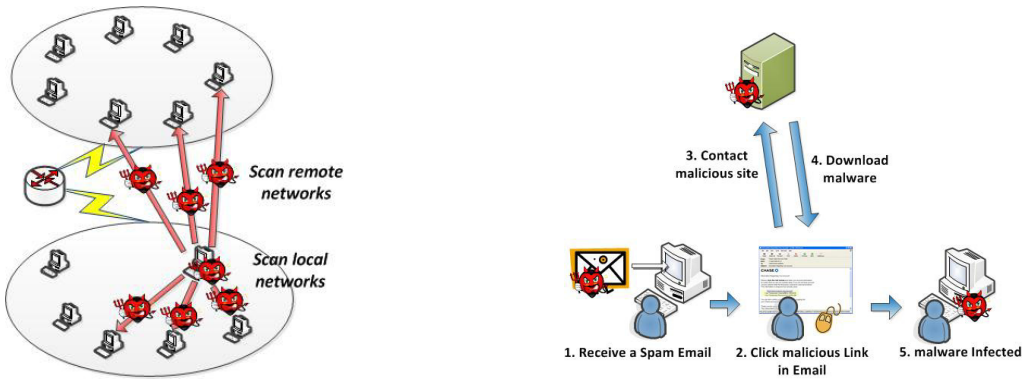
FRESCO modules can also produce flow rules, and thus provide an efficient means to implement security directives to counter threats that may be reported by other FRESCO detection modules. Our FRESCO modules incorporate several security functions ranging from simple address blocking to complex flow redirection procedures (dynamic quarantine, or reflecting remote scanners into a honeynet, and

so on). FRESCO also incorporates an API that allows existing DPI-based legacy security tools (e.g., BotHunter [34]) to invoke FRESCO's countermeasure modules. Through this API, we can construct an efficient countermeasure application, which monitors security alerts from a range of legacy IDS and anti-malware applications and triggers the appropriate *FRESCO* response module to reprogram the data planes of all switches in the SDN network.

2. UNDERSTANDING BOT MALWARE

2.1 Introduction

Botnets have become serious threats to the Internet. They send huge amount of spam emails and perform DDoS attacks on popular web servers [41]. To perform these attacks more efficiently and effectively, botmasters, who control botnets, keep trying to recruit new victims. In their infection trials, two methods are popular: network scanning and web-exploit. These two methods are quite different from each other. Figure 2.1 (a) shows a botnet using the network scanning method (randomly or with some local preference) to find new victims who have the same network service vulnerabilities. If successful, it will try to install malicious binaries on them. This approach is active and does not require any user operations (user-independent). In addition, it can be considered a server-side vulnerability exploitation, because it depends on opened (vulnerable) network services. In this work, we define this type of botnet infection as *Type I*.



(a) Type I infection of scanning botnets (b) Type II infection of web-exploit botnets

Figure 2.1: Infection approaches of scanning and web-exploit botnets.

In the case of a botnet using the web-exploit method (Figure 2.1 (b)), it finds new victims by sending spam emails (or other social messages) which include URL links pointing to malicious sites. If the recipient clicks the URL link, malware could exploit browser vulnerabilities and be installed on his host automatically (a.k.a drive-by download attack). This approach highly depends on user interactions (i.e. the user has to click/visit malicious URLs in order to be infected). Thus, we consider this web-exploit type of botnet infection user-dependent and relatively passive. It can also be regarded as a client-side infection, because its infection depends on client behaviors and browser vulnerabilities. In this work, we define this type of botnet infection as *Type II*.

Several previous studies have made attempts to understand the characteristics of botnets, becuase of their serious threats [107, 94, 55, 14]. These studies provide interesting observations and insights. However we find that most these studies focus on one botnet or one type of botnets (i.e., focusing on analysis of multiple botnets employing the same infection approach).

This tendency lets us invoke a question of *"why is it hard to find some studies which compare multiple botnets employing different infection methods from each other, when we can infer that botnets using different infection approaches show quite different characteristics?"*. The main reason might be difficulty of collecting and analyzing large-scale data of multiple botnets. To collect large-scale botnet data, we have to set up a data collecting system and it requires a lot of servers, network devices, and other related equipments (e.g., power supply). It is not s simple job and costs too much. Even if we can set up the data collecting system and gather enough data, analyzing the data is another problem, because it is very hard to find meaningful information from the data.

The main goal of this work is to provide a large scale cross-analysis of multiple

botnets, whose infection approaches are different from each other. In particular, we intend to answer the following questions:

- Do these two types of botnets have different infection patterns, such as the distribution of their victims? And how similar or different are they? From the above description, it is reasonable to guess that they might have different infection patterns given the fact that their infection vectors/methods are so different. However, there is no prior work to confirm whether this hypothesis is true or not.

- Why do these two types of botnets exhibit similar/different infection patterns? If they have similarities, who are the common victims (that are extremely vulnerable to both infection types)? If they have differences, why are some victims more vulnerable to one infection type than the other one?

- What can we learn from cross-analysis? What new implications and insights can we gain? How can they guide us towards new malware defense strategies/techniques?

2.2 Data Collection and Term Definition

In this section, we provide information of data that we have analyzed and we define several terms used in this work.

2.2.0.1 Data Collection

To understand the characteristics of different types of botnets, we have collected data for three major botnets: Conficker, MegaD, and Srizbi. Conficker [96] is a recent popular botnet known to have infected several million Internet machines. It propagates automatically through network scanning. It first scans random networks to find new victims and if it infects a host successfully, it scans neighbor networks

of the host to find victims nearby [19]. Thus it is a representative example of *Type I botnets*. The MegaD [81] and Srizbi [61] botnets are two recent botnets known for sending large volume of spam since 2008. In particular, it is mentioned that MegaD was responsible for sending about 32% of spam worldwide [81] and Srizbi was responsible for sending more than half of all the spam in 2008 [72]. They are representative examples of *Type II botnets* because they spread by drive-by-download [81, 61] or pay-per-install methods [20].

The Conficker botnet data has been collected by setting up sinkholing servers because Conficker uses domain-fluxing to generate C&C domain names for victims to contact [96]. With the help of *shadowserver.org*, we have collected a large dataset of Conficker infection including about 25 million victims [86]. The *shadowserver.org* has set up several sinkhole servers and registered the domain names same as the Conficker master servers to redirect queries of the Conficker bots to the sinkhole servers. Then, the sinkhole servers capture the information of hosts contacting them and the hosts can be considered as the Conficker infected victims.

Botnet	*Data Source*	*Main Infection Vector*	*# of Victims*	*Collection Date*
Conficker	Sinkhole server [84]	network scanning	24,912,492	Jan. 2010
MegaD	Spam trap [8]	drive-by-download or PPI	83,316	Aug. 2010
Srizbi	Spam trap [8]	drive-by-download	106,446	Aug. 2010

Table 2.1: Data summary of collected botnets.

The MegaD and Srizbi botnet data has been collected through the *botlab project* [8], of which spam trap servers were used to gather information of hosts sending spam emails. The detailed summary information regarding our collected data is presented in Table 3.2. The *botlab project* captures spam emails from spam-trap servers and

further investigates the spam emails through various methods such as crawling URLs in the spam emails and DNS monitoring. From correlating the investigation results, the *botlab project* finally reports which hosts are considered as infected by spam-botnets such as MegaD and Srizbi.

2.2.0.2 Term Definition

Before we perform cross-analysis on the data, there are several important issues to be addressed which can bias our result. The first thing is the *dynamism* of the IP address of a host. Many ISPs use dynamic IP address re-assignment to manage their assigned IP addresses efficiently [106]. This makes it hard to identify each host correctly. This may cause some biases in measuring the population or characteristics of the botnet [75]. Second, we are not likely to collect the *complete* data of certain botnets but only parts of the data (e.g., MegaD and Srizbi), and this can also cause some biases.

To account for these issues, instead of basing our analysis unit granularity on the individual IP address level, we generalize our analysis to examine at the network/-subnet level by grouping adjacent IP addresses. This will help mitigate the effect of *dynamism*, because it is common that dynamic IP addresses of a host come from the same address pool (subnet). Also, we believe that it is sufficient to examine subnets because even if only one host in the network is infected, the neighbor hosts are likely to be vulnerable or be infected soon [86].

In this work, we define our base unit for analyzing, i.e., *"infected network"*, as the /24 subnet which has at least one malware infected host. Thus, if a sub-network is infected by a *Type I botnet*, we call the subnet a *Type I infected network* and a similar definition is also applicable to *Type II infected networks*. In addition, we define a *Common infected network* as an *infected network* which has victims of both types of

botnets. There may be some *infected networks* that are exclusively infected by either *Type I* or *Type II*, which are defined as *Type I EX* or *Type II EX infected networks*, respectively.

In our data set, we found 1,339,699 *infected networks* in the case of the Conficker botnet, 71,896 for the MegaD botnet, and 77,934 for the Srizbi botnet. Thus, we have data for around 1,339,699 *infected networks* for the *Type I botnet* and 137,902 *infected networks* for the *Type II botnet*[*]. From this we have identified 97,290 *Common infected networks*.

2.3 Cross-Analysis of Botnet Victims

In this section, we provide detailed cross-analysis results of two types of botnets.

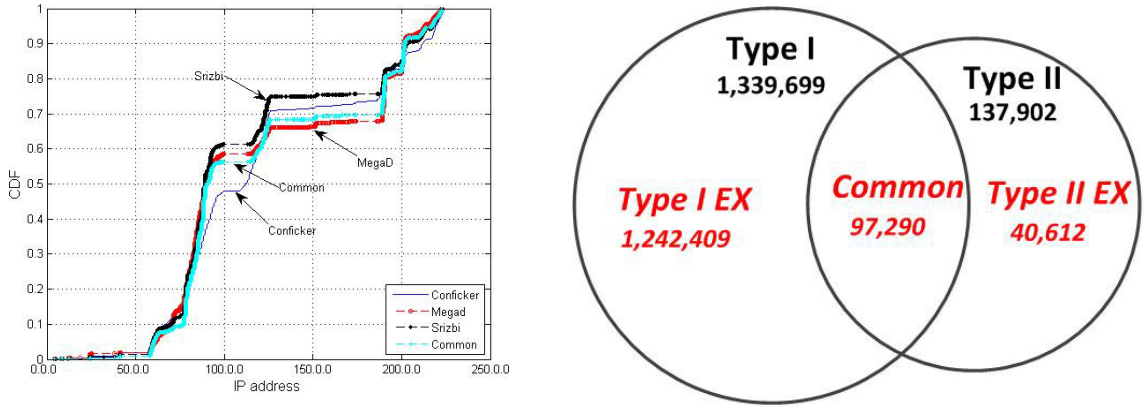
2.3.1 Point of Departure

We start our analysis with the following *Hypothesis 1* that we proposed in Section 2.2.

Hypothesis 1. *Since the two types of botnets have very different infection vectors, they may exhibit different infection patterns (e.g., distributions of their infected networks).*

To verify this hypothesis, we measure how many *infected networks* are shared by both types of botnets and how they are different from each other. The basic measurement results are shown in Figure 2.2. Figure 2.2(a) shows the distribution for *infected networks* of each type of botnet over the IP address spaces (*Type I (Conficker), II (MegaD and Srizbi), and Common infected networks*). Interestingly, the distributions of *Type I* and *Type II botnets* are very similar to each other. Specifi-

[*]There are 11,928 *infected networks* in common between MegaD and Srizbi.



(a) Infected network distributions (b) Infected network diagram

Figure 2.2: Infected network distributions and diagram.

cally, the IP address ranges of (77.* - 96.*), (109.* - 125.*), and (186.* - 222.*) are highly infected by both types of botnets and their shared regions (*Common*) are also distributed in the similar ranges.

To investigate how many *infected networks* are *"really"* shared between them, we draw a diagram which represents the number of *infected networks* of each type of botnet and networks that they share in common in Figure 2.2(b). There are 97,290 *Common infected networks*, 1,242,409 *Type I EX networks*, and 40,612 *Type II EX networks*.

Contrary to our expectation, the two types of botnets are distributed over similar IP address ranges and there are many *Common infected networks* between them. However, this observation is only about the distribution over the IP address space and it is very hard to find semantic meanings such as their physical locations from this result. For instance, even though we know a /24 subnet *111.111.111/24* is an infected network, we may not understand who are using the subnet and where the subnet is located. More importantly, why is the subnet more likely to be infected

by certain type (or both types) of botnets? In addition, the ranges are too broad to comprehend clearly. We show range (77.* - 96.*) is highly infected, but that does not mean that all IP addresses in the range are infected, we need more fine-grained investigation. Besides that, we also find that there are some differences between them (i.e., *Type I EX and II EX infected networks* are still significant) and they also need to be understood, because they can show which ranges are more vulnerable to which type of botnet. Only considering IP address ranges might not clearly show these differences.

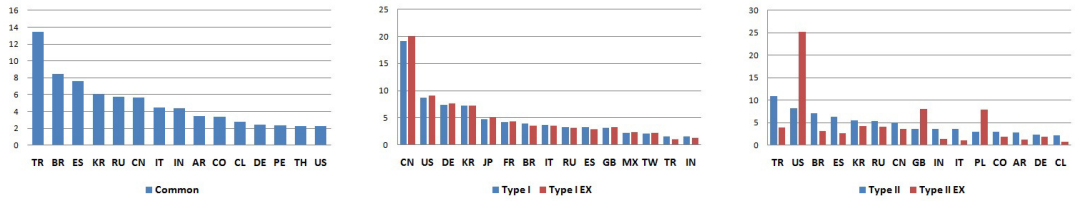
Thus, we are motivated to consider more viewpoints that provide us some understandable meanings with fine-grained level semantic information. We have selected four interesting viewpoints (we call them *categories*): (i) geographical distribution of infected networks, which lets us identify more (or less) vulnerable locations and their correlation with certain types of infections, (ii) IP address population/density, which helps us understand relationships between the number of assigned IP address to the country and the number of infected networks of the country, (iii) remote accessibility of networks, which shows us how open (and thus possibly prone to infection) the networks are and whether there is a correlation with certain infection types, and (iv) dynamism of IP addresses, which tells us whether vulnerable networks use more dynamic IP addresses and the correlation with infection type. In each category, we build a hypothesis based on some intuition and then we perform a large scale passive or active measurement to verify the hypothesis and gain some insights.

Insight 1. *Interestingly, the two types of botnets are distributed in similar IP address ranges despite of their different infection types. In addition, the ranges are continuous and it might imply that vulnerable networks are close to each other. More fine-grained analysis over the ranges might help us find new results and insights.*

2.3.2 Geographical Distribution of Infected Networks

In our first test, we have observed that two types of botnets seem to have similar distributions over the IP address space. Thus, we could infer that the distributions of two different types of botnets over geographical locations are similar to each other. From this intuition, we make the following hypothesis.

Hypothesis 2. *Type I and Type II infected networks are mainly distributed over similar countries.*



(a) Common (b) Type I and I EX (c) Type II and II EX

Figure 2.3: Infected network distributions over the countries (x-axis for country code, y-axis for percentage)

To verify this hypothesis, we investigate how each type of infected network is distributed over countries. When we observe the overall distribution of each type of botnet over the countries, we find that all *Common, Type I, Type I EX, Type II, and Type II EX infected networks* spread all over the world (with the exception of Africa), but there are some concentrated areas. To analyze the result in detail, we select the top 16 countries of each case and show their distributions in Figure 2.3. Results are sorted by the number of *infected networks* of the countries. Here, X-axis represents

the country code and Y-axis represents the percentage of each infection type, e.g., if there are 100 *Common infected networks* overall and 14 *infected networks* are located in Turkey (its country code is TR[†]), the percentage of Turkey is 14%.

In Figure 2.3(a), *Common infected networks* are mainly distributed in Asia (e.g., Turkey, Korea, Russia, China, and India) with more than 35%. Figure 2.3(b) also presents that *Type I and I EX infected networks* are mainly distributed over Asia. The distributions of *Type I EX infected networks* are quite similar to that of *Type I*. The distributions of *Type II and II EX infected networks* are shown in Figure 2.3(c). Here we still observe more than 30% as being located in Asia.

From the observations, we find two interesting things. First, the set of countries that are highly infected are not very different for each type of botnet (i.e., if some countries are highly infected by *Type I botnet*, they are also likely to be infected by *Type II botnets*). This implies that these countries are more prone to be infected regardless of infection methods. Second, there are some countries that are highly vulnerable to one type of botnet over the other. China is a good example of this. China has a lot of *Type I infected networks*. However, it has relatively small portions of *Type II infected networks*. We presume that most of the networks in China are accessible from remote scanning botnets because *Type I botnets* usually use network scanning techniques to find new victims. We will test this in section 2.3.4 and show whether our presumption is correct.

Insight 2. *There are some countries which are prone to be infected by both types of botnets. However, some other countries are more likely to be infected by one type*

[†]Each country code represents followings; AR Argentina, AU Australia, BR Brazil, CA Canada, CL Chile, CN China, CO Colombia, DE Germany, ES Spain, FR France, GB Great Britain, IN India, IT Italy, JP Japan, KR South Korea, MX Mexico, NL Netherlands, PE Peru, PL Poland, RO Romania, RU Russian Federation, SE Sweden, TH Thailand, TR Turkey, TW Taiwan, US United States, VN Vietnam

of botnet. Management policies of networks (e.g., network access control) could affect malware infection of the country.

2.3.3 IP Address Population

From the previous result, we know that the *infected networks* of each type of botnet are concentrated mainly within several countries but the infection rates between them are different. Why is the infection rate between them different? Are there any possible answers or clues that might explain this? To find out some clues, we first focus on the number of IP addresses assigned to each country.

IP addresses are not assigned evenly over networks or locations [40] [43]. In terms of the IPv4 address space, there are some IP address ranges which have not been assigned to users but registered only for other purposes, e.g., (224.* - 239.*) is assigned for multicast addresses [40]. In addition, IP addresses have been assigned differently over locations, e.g., more than 37% of IP addresses are assigned to the United States, while Turkey only has less than 0.5% [43]. From this fact, we can easily infer that countries that have more IP addresses could have more chances to be infected by malware leading to *Hypothesis 3*. Here, we will use the term of *IP address population* to represent the number of assigned IP addresses and we define *high IP address population country* as the country ranked in the top 30 in terms of the number of assigned IP addresses, and *low IP address population country* as the country ranked below 30. All ranking information is based on [43].

Hypothesis 3. *Countries with more IP addresses (high IP address population countries) might contain more of both types of infected networks than low IP address population countries.*

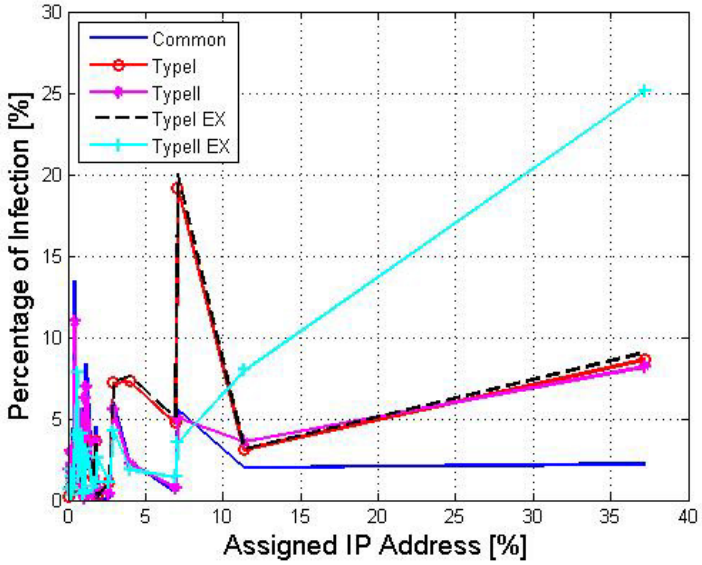


Figure 2.4: Infected network distribution versus IP address population (x-axis for percentage of assigned IP addresses to a country, y-axis for percentage of infection of each type of botnet in the country)

To verify this hypothesis, we compare the number of *infected networks* of each type of botnet with the number of IP addresses assigned to each country. The comparison results are shown in Figure 2.4. We can see that the number of *infected networks* of the *Type I, II, I EX, II EX botnets* are relatively proportional to the *IP address population* (i.e., the more IP addresses a country has, the more *infected networks* it contains). However, in the case of *Common infected networks*, they are *NOT* proportional to *IP address population*. On the contrary, they are mainly distributed over some *low IP address population countries*.

Intuitively, countries with more IP addresses have more chances to be infected. Thus, we can easily accept the results of *Type I, II, I EX, II EX*. However, why do some *high IP address population countries* have less *Common infected networks* while some *low IP address population countries* have more? There may be several possible

reasons for this. For example, the security education/knowledge of people may play a role. People may open some vulnerable services or click suspicious URLs without serious consideration, if they do not have enough education/knowledge of security in some countries. Another possible reason is in regards to network management. If networks in a country are well managed and protected very carefully, it is harder for malware to find chances to infect the networks. Thus, malware infection rate would not be proportional to the number of IP addresses in the country.

The other interesting point is the *percentage of infected networks over all networks of the country* (e.g., if a country has 100 networks and if 10 networks among them are infected, the percentage of *infected networks* of the country is 10%). We have observed that *high IP address population countries* are likely to have more infected networks. However, it does not mean that most (or a high percentage) of networks in the country are infected. For example, even though the United States has more number of *Type II infected networks* than other countries (except Turkey), the *infected networks* may only cover small percentage of all networks in the United States, because the country has around 38% of IP addresses of the world. This can reveal some *low IP address population countries* whose networks are more vulnerable (in terms of percentage) than other countries and they could be ignored if only considering the absolute number of *infected networks*.

To investigate the percentage of *infected networks* of each country, we have used the data from the *IP2Location.com* report [43]. In the report, we find that 2,505,141,392 IP addresses have been observed in the world. This may not cover all observable IP addresses in the world. However we believe that it is close to the real value. Their report also shows the percentage of IP addresses that each country has out of all observed IP addresses.

We use this data to calculate the number of IP addresses assigned to each country.

Then, we calculate the number of /24 sub-networks of each country by dividing the number of IP addresses assigned to the country by 256. At this time, we make an assumption that *"IP addresses are assigned to each country with the minimum unit size of /24 subnet"* to make our calculation easy. And we calculate the ratio of *infected networks* in each country with it and the number of infected /24 subnets. This scenario can be formalized as follows.

- Θ = the number of all IP addresses in the world (i.e., 2,505,141,392)
- ϵ_j = the percentage of assigned IP addresses to the country j
- α_j = the number of /24 subnets in country j
- γ_i = the number of *infected networks* of type i botnet (e.g., γ_1 represent the number of *infected networks* of *Type I botnet*)
- η_i = the percentage of *infected networks* of type i botnet in each country

Our goal is to calculate the value of η of each country, and this can be obtained by the following formula (here $j \in \{1, 2, ..., 240\}$, and 240 denotes the number of countries which have observable IP addresses).

- $\alpha_j = \frac{\Theta}{256} * \epsilon_j$
- $\eta_i = \frac{\gamma_i}{\alpha_j} * 100$, where $i \in \{1, 2\}$

The distribution of the values of η over some selected countries are shown in Figure 2.5 2.7, and 2.7. This result is quite different from the previous result (in Figure 2.3). In the case of *Common* (Figure 2.5), some top ranked countries in Figure 2.3 show quite low η values. Russia, Korea, China, and the United States are examples of this case, however Turkey still represents high η value. From the results,

we can understand which countries are more vulnerable (i.e., high η value). Peru is an interesting case. It has not been known as a country containing large number of *infected networks* in our previous results. However large portions of its networks in the country seem to be infected. *Type I, I EX, II, and II EX* also show similar characteristics to the *Common* case and the results are shown in Figure 2.6 and 2.7. Based on these results, we may focus on some vulnerable countries (e.g., Turkey and Peru) to study infection trends of botnets. They may be good candidates for monitoring in order to comprehend the infection trends of botnets.

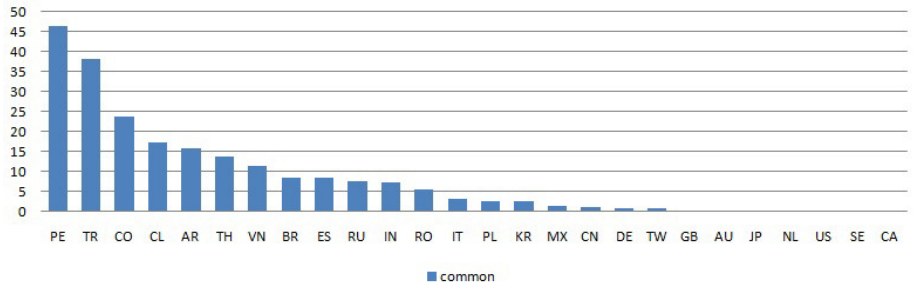


Figure 2.5: Common case: η values of selected countries (x-axis for country code, y-axis for η value)

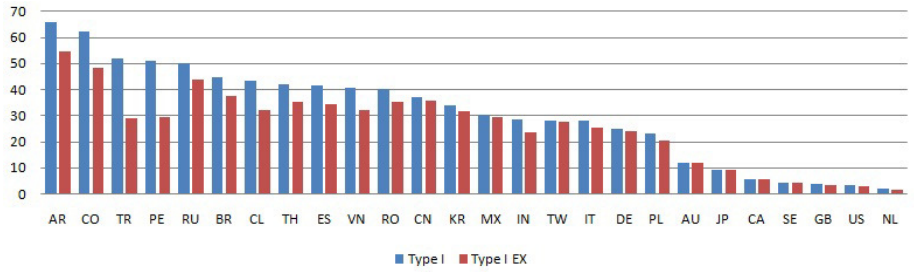


Figure 2.6: Type I and EX case: η values of selected countries (x-axis for country code, y-axis for η value)

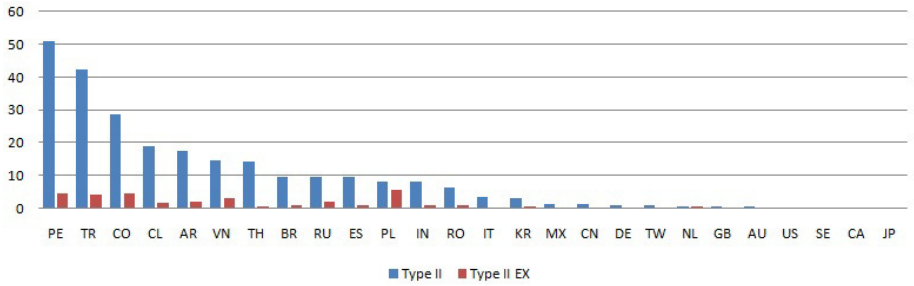


Figure 2.7: Type II and EX case: η values of selected countries (x-axis for country code, y-axis for η value)

We try to reveal the reason why Turkey and Peru show high η values. From our investigation, we find a possible reason. It can be caused by *geopolitical reasons*. Some previous work pointed out that Turkey has been suffered from large cyber attacks generated by its neighbor countries such as Russia [5]. This explanation is also applicable to Peru, because it is surrounded by several countries that have a lot of malware infected networks such as Brazil and Mexico.

Insight 3. *To understand malware distributions, we might put our focus on not only high IP address population countries with large number of infected networks, but also some low IP address population countries where large portions of their networks seem to be infected. Malware infection of these low IP address population countries could be affected by geographical neighbors.*

2.3.4 Remote Accessibility

Another category that we consider is the network openness or remote accessibility (i.e., whether a host can be directly accessed from remote hosts or not). As we described in the previous section, one major infection vectors of the *Type I botnet* is scanning remote hosts (or networks). Enterprise networks are usually protected by

several perimeter defending systems such as firewalls, in an attempt to block malicious threats from remote hosts. However, *not* all networks are protected as such and if they are not protected, malware can infect internal unguarded hosts more easily. From this intuition, we build the following hypothesis.

Hypothesis 4. *Networks that are more open (more directly accessible from remote hosts) might have more infected networks of Type I botnets than that of Type II botnets*.

We have tested the network accessibility by sending several *Ping* packets (i.e., testing ICMP reachability) to 5 randomly selected hosts in a network. If any of our *Ping* queries is successful in selected hosts, we regard that the network is reachable from remote hosts, otherwise we regard that the network is unreachable. This test has been already used before to understand the network reachability by previous work [12]. Note that this test may only show the *lower bound* of reachable networks, because some perimeter defending systems (e.g., firewalls) block incoming ICMP packets, or our randomly selected hosts may be not alive during testing. In this test, we assume that each /24 subnet have the same network access control policy (i.e., if one of the host in the same /24 subnet is accessible from the remote host, we consider that all hosts in the same /24 subnet might also be accessible).

In our test, we can access 54.32% of *Type I infected networks*, which is more than half. This indeed shows that *Type I infected networks* are more open (remote accessible). It confirms our hypothesis, although we presume this ratio could be higher for *Type I*. This could be probably explained by (a) our network reachability test is only a low-bound estimation, and (b) more networks are aware of malware scanning attacks and thus more (previously open) networks installed firewalls. In the

case of the result for *Type II*, it shows 46.85% networks are accessible, which is much less than *Type I*. This is probably because the infection vectors of *Type II botnets* do not depend on remote accessibility.

The result of *Common* is interesting, because it shows more than 60% of networks are accessible. This implies that remote accessible networks are much more vulnerable to malware attacks. It might be reasonable, because even though network accessibility may not help *Type II botnets* infect hosts, at least it helps *Type I botnets*.

In addition, we measure the remote accessibility of networks of three countries: Turkey, China and the United States. These countries show somewhat interesting patterns (e.g., China has a lot of *Type I infected networks*, but has relatively small number of *Type II infected networks*). In our measurement, we find that 64.09% of networks in China are accessible from remote hosts. This corresponds with our previous prediction (i.e., networks in a country that has a lot of *Type I infected networks* might be more accessible from remote hosts) in section 2.3.2. We discover that 51.8% of networks are accessible in the case of Turkey and 40.92% of the United States. This result seems to be reasonable, because these countries are more vulnerable to *Type II* than *Type I botnets*.

Insight 4. *Open (remote accessible) networks are more likely to be infected, particularly by Type I infection. However, it does not mean that inaccessible networks are much more secure, because malware (Type II infection) can still infect hosts in protected networks by several smart attack methods such as social engineering.*

2.3.5 Dynamism of IP Address

Previous work has shown that a lot of bots used dynamic IP addresses [106]. We want to investigate whether the networks with more dynamic IP addresses are more

vulnerable than those with static IP addresses for both types of botnet infections.

Hypothesis 5. *Places (or networks) with more dynamic IP addresses are more prone to be infected by both types of botnets.*

To understand this, we have analyzed how many infected networks are using dynamic IP addresses. For the analysis, we apply the technique of finding dynamic IP addresses proposed by Cai et al. [12]. In their analysis, they used reverse DNS PTR records of each host. They believed that the reverse PTR record can represent the status of a host and if some keywords of a reverse PTR record represent dynamism of IP address, the host is likely to use dynamic IP address. For instance, if a reverse PTR record of a host A is *dynamic-host.abcd.com*, it is very likely for the host A to use dynamic IP address, because its reverse PTR record has a keyword of *dynamic-host*. Note that this test only shows the lower bound of dynamic networks due to the limitation of reverse DNS lookup and selected keywords. Even though this test can not show all networks using dynamic IP addresses, it could give us information of which type of botnet has more dynamic IP addresses. Based on this idea, we use the same keywords mentioned in [12] to find hosts (and finally networks) which are likely to use dynamic IP addresses. If we find any host in a subnet using keywords representing the dynamism, we simply consider that the subnet uses dynamic IP addresses.

We have measured how many *infected networks* use dynamic IP addresses and the results are summarized in Table 2.2. The results are quite interesting. In the case of *Type I, I EX, and II EX* we find that around 50% of *infected networks* use dynamic and other 50% of *infected networks* use static IP addresses. However, in the case of *Common and Type II*, *infected networks* use more dynamic IP addresses

Type	*Dynamic IP*	*Static IP*
Common	62%	38%
Type I	50.1%	49.9%
Type II	58.4%	41.6%
Type I EX	49.08%	50.92%
Type II EX	51.87%	48.13%

Table 2.2: Comparison of the percentage of dynamic or static IP addresses of each type.

than static IP addresses.

The result of *Common* matches the previous result [106] which mentioned dynamic IP addresses are more vulnerable. However, the result of *Type I* does not fully match the previous result, i.e., *Type I botnet* infection does not have noticeable preference on networks with more dynamic addresses. This is actually reasonable because *Type I botnets* locate a remote victim by scanning the IP address space regardless whether the target address is dynamic or static. In the case of *Type II botnet infection*, we do observe infection preference on networks with more dynamic addresses. This is also reasonable because there are probably more home users in these (dynamic) address space who have less security awareness and potentially more vulnerable computers and web browsing patterns.

Insight 5. *Networks with more dynamic IP addresses are more vulnerable to malware attacks. This is more noticeable in the case of Type II botnet infection than Type I.*

2.4 Neighborhood Correlation of Botnet Victims

In this section, we provide a prediction approach based on our insights obtained in the previous section.

2.4.1 Watch Your Neighbors

Insight 1 in Section 2.3.1 points out that both types of botnets have heavily uneven distributions of infected networks and there are several heavily (continuous) infected areas in some part of the IPv4 space. This implies that *infected networks* of both types of botnets might be close to each other, i.e., it is very likely for them to be located in the same or similar physical locations and neighbor networks (e.g., belonging to the same /16 networks). This intuition has already been discussed before and verified in some previous work for some *Type I botnet* [19] [55] [86]. An interesting thing is that one of the previous work provides an approach of predicting unknown victims based on the intuition and it predicts unknown victims with more than 90% accuracy with only employing a simple method (e.g., K-Nearest Neighbor classification) [86]. However, this work has only focused on the case of *Type I botnets*.

The reason for strong neighborhood (network) correlation of *Type I botnets* is intuitive, because *Type I botnets* will very likely scan neighbor networks to recruit new victims. Then, can we apply a similar prediction approach to *Type II botnets*? At first glance, this might not be the case because *Type II botnets* have very different infection vectors/types from *Type I botnets*. However, we have also shown in the previous section that the distributions of both types of botnets are continuous and seems to be close to each other (in Figure 2.2(a)). Thus, it is hard to immediately draw a conclusion whether similar neighborhood correlation could be found in *Type II botnets* or not. Next, we plan to empirically verify this myth.

The previous work [86] has used the K-Nearest Neighbor (KNN) classifier which is a very popular machine learning algorithm and it uses neighbor information for classification. We also apply the KNN algorithm and select the same features for the KNN classifier used in [86]: /24 subnet address and physical location of *infected*

networks.

To perform this experiment, we first prepare data for representing the class of *benign* and *malicious* networks. At this time, the *infected networks* of *Type II botnets* can be used to represent the *malicious class*. However, since we do not have data for the *benign* class, we also collect many (at the same scale as malicious networks) clean networks[‡] to represent it. When we collect benign networks, we intentionally choose those which are close to *infected networks* in terms of the IP address and physical location, and they could be also neighbors of *infected networks*.

After the preparation, we divide each *Type II botnet* data (MegaD and Srizbi) into two sets for training/testing. And then, we apply the KNN classifier to predict unknown *infected networks*.

As shown in Table 2.3, the prediction results are quite interesting. Even though the prediction accuracy is lower than the case of *Type I botnet* (i.e., [86] reported around 93% of accuracy), our predictor for *Type II botnet* (in both MegaD and Srizbi cases) shows more than 88% accuracy with some reasonably small number of false positives.

Botnet	K	Prediction Accuracy	False Positive Rate
MegaD	1	88.35%	7.35%
	3	88.25%	7.36%
	5	88.14%	7.54%
Srizbi	1	88.20%	6.23%
	3	87.70%	6.04%
	5	88.30%	5.77%

Table 2.3: Botnet prediction results.

[‡]We checked whether they are clean or not by looking up several DNS blacklists.

The results imply that *Type II botnets* also have the similar characteristics as *Type I botnets* (i.e., if a host is infected, its neighbors are also likely to be infected). Then, why does this happen? It may be very hard to find concrete answers or clues for this question (unlike the intuitive explanation for *Type I* infection).

From our investigations, we could provide a possible answer. It may be caused by its infection media. As we described before, one promising infection method of *Type II botnets* is drive-by-download, which typically uses spam emails containing links to compromised web sites, to trick people into downloading malicious binaries. Thus, the infection pattern of *Type II botnet* might highly depends on who receives spam emails. We find articles describing how spammers harness email addresses [76] [26], and they point out that collecting mailing lists is one of their main tasks. It is likely for mailing lists to contain email addresses belonging to similar locations (e.g., same company and same university). It implies that spam emails are delivered to people who are likely to be close to each other and thus victims infected by spam emails might also be close to each other.

2.4.2 Cross-Bonet Prediction

We have shown that if a host is infected by a *Type II botnet*, its neighbor networks are also likely to be infected by this *Type II botnet*. When we perform this test, we treat data of MegaD and Srizbi separately. However we know that these two botnets are very similar in terms of infection vectors. To confirm the similarity of their infected networks, we calculate a *manhattan distance* between the distribution of the two types of botnets. The *manhattan distance* between two items is the sum of all feature value differences for each of the all features in the item, and it is frequently used to denote whether two data distributions are similar or not (e.g., if a distance between data distributions of A and B is smaller than between that of A and C, A

and B are closer to each other than C). It can be formalized as the following equation (assuming that there are two items/distributions of x and y, and they both have n elements).

$$Manhattan\ Distance = \sum_{i=1}^{n} |x_i - y_i|$$

We use the probability distributions of infected networks of Conficker, MegaD, Srizbi over IP address spaces to measure the *manhattan distance* and we find that the *manhattan distance* between Conficker and MegaD is 1.1427, Conficker and Srizbi is 1.1604, and MegaD and Srizbi is 0.8404. From the results, we can easily see that the distance between the *Type I* and *Type II botnet* distributions is larger than the distance between the two *type II botnets* distributions. This result shows that the distributions of infected networks with the same infection type are closer to each other than that of different types of botnet (i.e., infected networks of botnets in the same type show very similar distribution patterns).

This result gives us another insight that *if two botnets share the same infection vectors (i.e., they are of the same type), we might predict unknown infected networks of one botnet (e.g., a future botnet) with the help of the information of the other botnet (e.g., historic data).* This insight can be verified with a similar test that we have done before. We can perform a test by simply changing the training and testing data set to cross botnets. In the previous test, we extract the training and testing data from the same botnet. However in this case, we use data from botnet A for training and data from botnet B for testing. For example, when we predict (unknown) *infected networks* of the Srizbi botnet, we use data of the MegaD botnet for training.

The cross-prediction results are quite surprising. As denoted in Table 2.4, this

Botnet	K	Prediction Accuracy	False Positive Rate
MegaD(train), Srizbi(test)	1	87.80%	7.41%
	3	86.75%	7.49%
	5	86.45%	7.69%
Srizbi(train), MegaD(test)	1	84.09%	6.53%
	3	83.89%	6.31%
	5	83.65%	5.09%

Table 2.4: Botnet cross-prediction results.

approach can predict unknown *infected networks* of the other botnet with more than 83% accuracy. This prediction accuracy is slightly less than what we observed previously. We believe that these results show us that even if we have no knowledge of some botnets (e.g., a future emerging botnet), if we have some information of a botnet whose infection vector is very similar to them[§], we may be able to predict unknown *infected networks*. To show a realistic example of application of the neighborhood correlation, let us first assume that a network administrator knows historic infected networks by Srizbi botnets. Then, he gets to know that the MegaD botnet starts spreading but he does not have any information of which networks are and will be infected. In this case, he can use the information of Srizbi botnet information (e.g., victim distribution). Based on the physical location and IP address of victims of Srizbi, he can predict future victim networks that will possibly be infected by MegaD with a reasonably high probability.

2.5 Limitations and Discussions

Like any measurement/analysis work, our empirical study has some limitations or biases. Even though we have collected a large amount of Conficker botnet data,

[§]Note that this is a very reasonable assumption because fundamental infection types of botnets are very limited and do not change frequently.

we have a relatively smaller amount of data for the MegaD and Srizbi botnets. This might cause some bias in our measurement results and subsequent analysis. In addition, the dynamism of IP addresses may lead to some over-estimation from the collected data. To reduce some of the side effects, we generalize our analysis over a network consisting of several adjacent IP addresses (i.e., measuring/analyzing over /24 subnets instead of each individual host).

To discover interesting insights, we leverage some previous work. For example, we use previous work to obtain how dynamic IP addresses are distributed over countries, but the information is not complete, i.e., it does not cover all countries. However, the provided information may help to uncover interesting cases (e.g., countries which are highly infected by botnets), hence the information is still useful.

When we perform the test to find networks with dynamic IP addresses through looking up reverse DNS PTR records of hosts in the networks, we may not collect reverse PTR records from all hosts because registration of a reverse PTR record is not always necessary. However previous work already verified the feasibility of such kind of test [12], lending credibility to these results (at least providing a good low-bound estimation).

2.6 Related Work

There are several studies of measurement or analysis of the *Type I botnet* victims. CAIDA provides basic information about the victim distribution of the Conficker botnet in terms of their IP address space and physical location [14]. In [55], Krishnan et al. conducted an experiment to detect infected hosts by Conficker. Weaver [103] built a probabilistic model to understand how the Conficker botnet spreads via network scanning. These studies provided useful and interesting analysis of the Conficker botnet. Shin et al. provided a large scale empirical analysis of the Conficker

botnet and presented how victims are distributed [86]. However, our work is different from them in that we perform cross-analysis of different botnets and propose an early warning approach based on cross-prediction. Even though [86] observed neighbor correlation in Conficker, this work differs in that we empirically verified similar neighborhood correlation in *Type II botnets*. In addition, we have proposed and verified cross-botnet prediction techniques to predict unknown victims of one botnet from the information of the other botnet if they have similar infection vectors.

Measurement studies of the *Type II botnet* were also conducted. In [61], Mori et al. performed a large scale empirical study of the Srizbi botnet. John et al. set up a spam trap server to capture botnets sending spam emails [46]. This work also showed the distribution of victims in terms of their IP addresses. Even though these studies provided detailed analysis of some *Type II botnet(s)*, they still differ from our work in that they concentrate on a single (or one type of) specific botnet.

Some interesting studies from the analysis of *Type II botnets* have been also proposed. In [20], Cho et al. analyzed the MegaD botnet and showed how it works. Caballero et al. provided an interesting technique to infiltrate the MegaD botnet and performed an analysis of its protocol [11].

Cai et al. measured how IP addresses are distributed over the world through several interesting sampling techniques [12]. Our work leverages some of its results but is different from their work in the main purpose.

2.7 Summary of this Chapter

In this chapter, we have collected a large amount of real-world botnet data and performed cross-analysis between different types of botnets to reveal the differences/similarities between them. Our large scale cross-comparison analysis results allow us to discover interesting findings and gain profound insights into botnet

victims. Our results show fine-grained infection information and nature of botnet victims. They show some interesting relationships between geopolitical issues and malware infection, which might be the first work shedding light on this correlation. This study can guide us to design better botnet prediction or defense systems.

3. DETECTING BOT MALWARE*

3.1 Introduction

In the previous chapter, we show the analysis results of bot infected hosts and some new insights based on our findings. These analysis results and insights can guide us to devise a new bot malware detection system, and in this chapter, we describe an approach of detecting bot malware based on our experiences and knowledge from our previous research (in Chapter 2). Before talking about the detection system, we briefly survey the existing bot malware detection systems to understand their weak and strong points.

To eradicate the threats posed by bot malware, a lot of research has been proposed so far, and they fall into two main categories: (i) *network-level detection* and (ii) *host-level detection*. Network-level detection approaches focus on network behaviors of bots/botnets. They typically concentrate on finding common patterns of network flows between bots and their botmasters (a.k.a C&C channels) [34, 32, 35]. Host-level detection approaches investigate bot runtime behaviors mainly using system call monitoring and/or data taint analysis [53, 92].

Both approaches have their own advantages and disadvantages in detecting bots. Network-level detection approaches can detect different types of bots without imposing overhead to the hosts, because they mainly monitor network traffic. However, their limitations appear when they need to detect a bot communicating through encrypted messages or with evasion attempts [93]. Host-level detection approaches, on the contrary, analyze suspicious runtime program behaviors in the host, so that they

*Reprinted with permission from "EFFORT: A new hostnetwork cooperated framework for efficient and effective bot malware detection" by Seungwon Shin, Zhaoyan Xu, Guofei Gu, 2013. Computer Networks, Copyright [2013] by Elsevier.

can detect a bot even if it uses an encrypted communication channel. However, they typically suffer from performance overhead because they need to monitor all invoked system calls [53] at realtime and/or taint memory locations touched by the program for information flow taint analysis [92].

After surveying both approaches, we have the following questions: (i) *Is it possible to design solutions consisting of strengths from both approaches?* (ii) *What kinds of features/heuristics of each approach are helpful to build an effective system?* and (iii) *Given these features, how do we combine them in an efficient way?* If we can answer questions of (ii) and (iii), then we could build a system that potentially answers the first question.

We start with thoroughly examining prevalent bots to deeply understand their intrinsic characteristics that they have regardless of their various implementations, operations, and C&C communications. Knowledge obtained when we perform the research described in the previous chapter, we observe the following invariants that hold true for almost all bots. First, they are automated programs that are non-human driven at the host side. Second, in order to be flexible and robust for C&C purposes and detection evasion, they heavily use DNS tricks for rallying. For example, they use dynamic DNS service, fast-flux service networks [39], or even domain fluxing [91]. And again their DNS queries are non-human driven unlike most normal programs. Third, in order to be useful, bots have to heavily access system resources. For example, they will attempt to read/steal information in file systems, change critical registry entries, or create new files/sockets. Finally, if we treat a networked program as a communication information processing unit, most normal client programs (e.g., browsers) are intent to gain information. However, bot programs tend to be more information leaking/losing oriented for most of their malicious activities (and they tend to minimize incoming communication to minimize possible detection

probabilities).

Observing the characteristics described above gives us new insights to detect bot malware more effectively and robustly. For each characteristic observed, we further investigate possible features or heuristics that contribute to effective and efficient detection. Naturally, we need to collect, combine, and correlate essential information at both host and network level. In particular, one unique feature of our approach is the tight combination/correlation of both host and network information. For example, we correlate network communications to corresponding programs at host level. Thus, different from all existing network-based approaches with basic analysis unit at *per-host* (IP) level, our granularity of network analysis can attribute to more fine-grained *per-process* level. This itself provides many advantages such as more accurate source attribution, and potentially more accurate detection results because it avoids the possible ambiguity and unwanted aggregation/mixing of traffic from all local programs.

To be more exact, at host level, we provide lightweight human-process-network correlation analysis. We correlate human-computer interaction with each program, and record correlated clues between network connections and running programs. Further more, we can monitor *system resource exposure patterns* of suspicious processes (e.g., those have non-human driven network communications). Our intuition is that the exposure surface of system resources, such as files, registries, and network sockets, is different between benign programs and bots.

At network level, we extract several different types of features. First, we build a reputation engine to characterize a process' reputation through examining "who you are" and "whom you are talking to". Our intuition is that the reputation of a process could possibly be decided by the reputation of its social contact surface, i.e., reputations of communicated remote servers/hosts. This is intuitively sound because

bot malware will communicate with "bad" server/host while good software tends to communicate with "good" ones. Although a pre-established host reputation database (or blacklist) is helpful, we do not require it as prior knowledge. Instead, we can use anomaly-based features from DNS registration information. More importantly, we want leverage community-based knowledge and intelligence by using public search engines to locate information about certain communicated targets and then infer their reputation. The interesting use of Google search engine was initially proposed in [97] with great success for traffic measurement. We differentiate ourselves from it in our different goal (for security purpose) and different use of features. Finally, we also analyze network information trading rate for any process to infer how likely the networked program is information gaining oriented or information leaking/outgoing oriented.

In short, this chapter makes the following contributions.

- We propose a new host-network cooperated framework for bot malware detection with correlative and coordinated analysis. This design demonstrates an important step from current state-of-the-art toward both *effective* and *efficient* botnet detection.
- We implement *EFFORT*, a prototype system containing several novel modules (e.g. process reputation analysis and system resource exposure analysis) to cover bot invariants at both host and network levels. We investigate multiple heuristics and features in these modules and we believe that they can capture many of bots' intrinsic characteristics.
- We evaluate our system on real-world data collected on several real-life lab/office and home machines for several days. Our results show that we can detect all 30 malicious operations from 15 tested bots with no false positives and the

overhead of our modules is negligible in our evaluation.

3.2 System Design

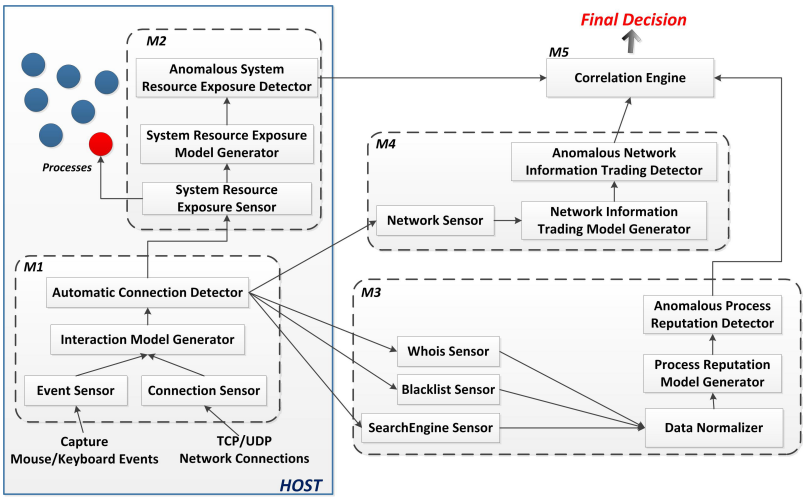


Figure 3.1: EFFORT design architecture, (M1 is for *human-process-network correlation analysis module*, M2 for *system resource exposure analysis module*, M3 for *process reputation analysis module*, M4 for *network information trading analysis module*, and M5 for *correlation engine*)

Our system consists of five modules to cover previously mentioned bot characteristics: (i) a *human-process-network correlation analysis module*, which analyzes the interaction and correlation between human activity, process, and network connection, i.e., to know whether it is a human-driven network connection or bot-driven; (ii) a *process reputation analysis module*, which characterizes reputation of a process from the process itself (who you are) and its social contact surface (the communicated targets, i.e., whom you are talking to); (iii) a *network information trading analysis module*, which monitors incoming/outgoing network traffic in order to infer the information gain/loss without heavy load; (iv) a *system resource exposure*

analysis module, which examines the system resource access patterns of a suspicious process in detail; and (v) a *correlation engine*, which collects all analysis results and correlates them to make a final decision whether the process is a malicious bot or not.

Every module except for the *correlation analyzer* has a similar architecture. They mainly consist of three elements; (i) sensor, which monitors network or process activity, (ii) model generator, which creates a model of process or network behavior, and (iii) detector, which decides whether each new action of a process or network is anomalous or not. The overall architecture is shown in Figure 3.1 and each item in Figure 3.1 will be explained in the following section.

3.2.1 Human-Process-Network Correlation Analysis

Since most running processes are benign, it is relatively inefficient to monitor all of them in fine-grained detail (e.g., monitor system call level activities) all the time to detect bots. If we can filter out all or most of the benign programs, we will not waste our resource/time in meaningless investigation. Our *human-process-network correlation analysis* module is designed to do this, i.e., sifting benign programs out. This is based on an intrinsic characteristic of bot malware, i.e., bots are automated programs that run without user interaction/notice, while most normal programs need some human interactions.

Human-Process Interactions Monitoring: Keyboard and mouse are the basic components that link human and the computer. If someone wants to send an email using an email client program, he will type an email address and contents by keyboard and click the send button through the mouse. Based on this intuition, we monitor keyboard and mouse events of the host to understand which program has human activity/interaction. To do this, our *event sensor* hooks Windows system calls

related to keyboard and mouse events, and also determines which program generates those events.

However, if a bot knows about our policy, it might imitate human behaviors and create virtual events to confuse our sensor. To resolve this problem, our sensor also checks whether the events come from actual physical devices or not. If the events are resulted from physical devices connected via PS2 or USB interface, it trusts them; otherwise it regards them as suspicious. Note that in current implementation, we trust operating system and we believe it provides true information, a common assumption widely used in this line of work. Of course, some malware (e.g., rootkit) may infect operation system and deliver fake information. This issue could be solved by using Hypervisor or TPM, as we discuss in Section 3.7.

Process-Network Interactions Monitoring: A *connection sensor* records outgoing network connections from processes in a host. In particular, it cares about one special network connection, DNS query. As briefly discussed before, botnets heavily rely on using DNS for flexible, efficient, and evasive C&C rallying. They can use fast-flux service networks [39] to frequently change the IP addresses associated with one domain name, or even use domain fluxing [91] to frequently change domain names. By monitoring these DNS queries, we can obtain valuable information later in detection analysis, e.g., we can determine if they are human driven or not, and furthermore we can even detect if there is fast-flux in use.

Interaction Model Generation and Automatic Connection Detection : Combining information from the *event sensor* and the *connection sensor*, an *interaction model generator* creates a model to describe which process has which network correlations so that later we can link any given network flow to a specific program/process in the host. The model simply uses time difference between the time when a process issues a DNS query and the time when a process produces a

mouse/keyboard event. We think that if the time difference is very small, the event and the query can be considered as continuous operations and therefore human is issuing the query. More formally, a DNS query time of a process can be defined as a variable d and a recent keyboard/mouse event time of the process can be defined as a variable e. Then we can create a model as following (if we assume that there are n processes in the host).

if $\epsilon > d_i - e_i > 0$, a DNS query of the process p_i is human-driven, otherwise bot-driven, where i is an integer $\in \{1, 2, ..., n\}$ and ϵ is a threshold

However, in practice, this intuitive model may not work for all DNS queries (but work well for IP address case). The reason is because some operating systems provide helper functions of relaying a DNS query for other processes, e.g., Windows uses the *svchost.exe* process for this purpose. Thus, with the above approach, we will frequently find that DNS queries are sent from the *svchost.exe* instead of the original program. To address this issue, we maintain a returned IP address(es) from a DNS query (sent by helper processes), and observe successive outgoing network connections to wait for the actual (original) program to connect to the returned IP address(es)[†].

This problem can be simply addressed by modifying the above model. If a DNS query is issued by a helper process (the query time d), we will wait until we find a process which contacts returned IP address(es) and apply the recent event time e of the process.

[†]At this time, we do not need to monitor all network connections, we only monitor first packet of each connection.

3.2.2 Detecting Malicious Processes

With the help of the previous module, we can filter out benign programs and only focus on some suspicious processes. However, there might be still some benign programs that send automatic DNS queries. For instance, *googleupdate.exe* will automatically contact servers [30] to check update status [29]. We should differentiate those benign processes from malicious processes using other features.

To do this, we perform a set of independent and parallel checks. First, we check the reputation of the process and its social contact surface (the reputation of targets it has communicated with). Second, we investigate the system resource access patterns of the process. Third, we observe network information trading of the process. We detail our design of these three modules as follows.

3.2.2.1 Process Reputation Analysis Module

A quick intuitive observation is that we could determine the reputation of a process by not just looking at "who you are", but also referring to "whom you are talking to". A bot malware will *automatically* contact some "bad" servers/peers in order to be useful or controlled. Benign programs are relatively unlikely to connect to "bad" targets *automatically*. Thus the problem of determining the reputation of a process could be inferred by contacting social surfaces of the process and it can be approximately reduced to the accumulation of "bad" communication targets (domains). In the context of domain names, then we need to determine the reputation of a domain name.

Domain Information Collection: We collect reputation information of the domain by employing three types of sensors. First, we employ a *whois sensor* to detect some anomaly features in its *registration information*, such as domain creation date. Second, we use a *blacklist sensor* to investigate its *previous records* in well-

known blacklists (e.g., SpamHaus [90], Google Safe Browsing [1]), which give us relatively clear clues whether the domain has a history of malicious activity. Finally, since blacklists might not be complete, we apply a *search engine sensor* to get another heuristic which can leverage community-based knowledge and intelligence, i.e., ask a search engine to infer the reputation of given domain name (IP address could work too). It is motivated by the googling idea in [97].

At this time, we do not need to monitor all domains, since there are a lot of domains which can be considered benign (e.g., google.com, bing.com and yahoo.com). Thus, we can maintain a list of benign sites and our sensors do not need to check connections to those domains for efficiency.

Data Normalization: Before applying collected data to a model creation, we express it numerically and normalize it.

In terms of a domain registration information, we consider the following heuristics: (i) whether the domain registration date is very recent (a lot of phishing and botnet C&C domains fall into this type), (ii) whether the domain expiration date is very soon (malicious domains typically come and go quickly and they do not register for a long time), (iii) number of registrars (typically very small for malicious domains). We simply use numeral values to represent each feature as shown in Table 3.1.

In the case of the blacklist, it is very obvious that if a domain can be found in blacklists, it is suspicious. We give "0" if it is in blacklists, otherwise "1".

To leverage community knowledge by using search engines like Google, we consider the following simple heuristics: (i) whether the domain name is well-indexed (thus returning many results), (ii) in the top returned web page results, whether the domain name and the process name are frequently used in a malicious context, e.g., they are surrounded by malicious keywords such as bot, botnet, malware, DDoS,

attack, spam, identity theft, privacy leak, command and control (C&C). Again we use numeric values to represent these heuristics as shown in Table 3.1. Typically, contents of returned search results include three different types of information: (i) the title, (ii) the URL, and (iii) the relevant snippet of the returned web page. We treat each type as different features, and we assign "1" if returned contents (title, URL, and summary) include the queried domain name. We inspect the returned results to see whether there are any malicious keywords or not. If there are any, we give "0" for its value.

Feature	*Numerical Values*
Domain Creation Date	Current Date - Creation Date
Domain Expiration Date	Expiration Date - Creation Date
Number of Registration	Number of Registration
Blacklist	if NOT in any blacklist it is 1, otherwise 0
Web Search Engine Results (title, URL, summary, malicious keywords)	The title, URL, and summary are 1 if we find the domain/process name in each item, otherwise 0. The malicious keyword is 1 if we can not find any malicious keywords from the returned pages, otherwise 0

Table 3.1: Numerical value of the selected features of domain reputation analysis module.

The features related to returned results by a search engine and blacklist are already normalized, i.e., their values are between "0" and "1". However, features of domain registration can be varied dynamically. To make their values range between "0" and "1", we employ a *gaussian normalization* approach. It regards a distribution of data as gaussian function and maps every data point into the probability of gaussian function.

Process Reputation Model Creation: Now, we have normalized values of

the features, then all that remains is to build a model based on them. We employ a Support Vector Machine (SVM) classifier [21] for the *process reputation model*. The SVM classifier maps training examples into feature spaces and finds (a) hyperplane(s) which can best separate training examples into each class.

Here we briefly talk about the SVM classifier.

To start with the simplest case, we assume that there are two classes and they can be separated by a linear function. More formally, given training examples x_i and a classifier y_i, if we assume that those two classes are denoted as 1 and -1 (i.e. $y_i \in \{-1, 1\}$), the training examples which lie on the hyperplane satisfy the following equation.

$w \cdot x + b = 0$, where w is a normal vector and $b/||w||$ is a perpendicular distance from the hyperplane to the origin.

From the above equation, we can find the hyperplanes which separate the data with maximal margin by minimizing $||w||$ under the constrains of $y_i(x_i \cdot w + b) - 1 \geq 0$. To solve this equation, we will apply a *Lagrangian formulation*, and then we will have a primal form - L_p - of the *Lagrangian* [10]. It is described as the following equations.

$$L_p \equiv \frac{1}{2}||w||^2 - \sum \alpha_i y_i (x_i \cdot w + b) + \sum \alpha_i \tag{3.1}$$

, where α_i is a *Lagrangian multiplier* and $\alpha_i \geq 0$.

Now, we have to minimize L_p with respect w and b, and it gives us two conditions of $w = \sum \alpha_i y_i x_i$ and $\sum \alpha_i y_i = 0$. In addition, we can substitute these conditions into L_p, since they are equality in the dual formulation. Thus, we can get dual form - L_d - of the *Lagrangian* like the following equation.

$$L_d = \sum \alpha_i - \frac{1}{2} \sum \alpha_i \alpha_j y_i y_j x_i \cdot x_j \tag{3.2}$$

Finally, we can get our SVM classifier through maximizing L_d. If we can not separate the data by a linear function, we have to extend the original set of training examples x_i into a high dimensional feature space with the mapping function $\Phi(x)$. Suppose that training examples $x_i \in R^d$ are mapped into the euclidean space H by a mapping function $\Phi : R^d \rightarrow H$, we can find a function K such that $K(x_i, x_j) = \Phi(x_i) \cdot \Phi(x_j)$ (a.k.a. "kernel function"). We can replace the inner-product of the mapping function by the kernel function and solve the problem with similar approach of the linearly separable case.

In this model, we consider that the normalized features, which are mentioned above, are training examples. In addition, we define that there are two classes - benign and malicious - in this model, thus the normalized features will represent one of the two classes.

Finally, we will find (a) hyperplane(s) which can best separate training examples into each class. Then, we can obtain a SVM classifier for the *process reputation model*.

Anomalous Process Reputation Detection: After creating a model of process reputation, we apply contacting domains of testing processes to the model. It is very frequent that a process contacts several different domains during a certain period. Thus, we apply all contacting domains to the model, and determine whether (a) "bad" domain(s) (i.e. classified as a malicious domain) exists or not. If there is (are), we consider the process reputation as bad (malicious), otherwise it is good (benign). More formally, if a process visits m different domains during T seconds, the detection result is represented as following (We define that y_i is for a classifier,

+1 for a "benign class", and −1 for a "malicious class").

If there is any domain d_i satisfying $y_i(d_i) = -1$ in the contacting domains of a process, the process is malicious (where $i \in \{1, 2, .., m\}$).

3.2.2.2 System Resource Exposure Analysis

If a bot infects a host, it usually tries to do something useful for its master (to make profit), e.g., stealing information, sending spam, and launching DDoS attacks [42]. Common characteristics of these operations are that they will consume system resources - memory, cpu and network - of the host, read/modify some system files or registries, and/or steal sensitive information of the owner [57]. If we can monitor how system resources (e.g., files, registries and network sockets) are exposed to this process (and to what degree), we could infer these anomaly access patterns.

System Resource Exposure Patterns Monitoring: A *system resource exposure sensor* monitors resource access activities of a process, which is considered to issue an automatic outgoing connection, and stores this information. It mainly observes how the process access files, registries, and network sockets.

System Resource Exposure Model Creation: To build this model, we use the following heuristics: (i) typically normal processes rarely access files in other user's folders and system directories, (ii) typically normal processes do not modify critical registries (with a few exceptions), and (iii) typically normal processes do not create a large number of sockets in a short time period. These heuristics are not perfect, i.e., some normal processes might have some of these patterns. Our goal of this module is not to have zero false positive, instead, we want to detect most of these system-resource-consuming malware (since for a botmaster, the purpose of controlling a bot-infected machine is to turn the machine into his resource for profit).

Thus, we think our heuristics are reasonable.

More specifically, these heuristics can be represented as following events Y_i of a process:

- Y_1, access files in other user's folders
- Y_2, access files in system folders
- Y_3, modify critical registries
- Y_4, create a new process
- Y_5, create network sockets more than threshold θ

At this time, most events are very clear to understand, however the event Y_5 needs a clearer definition to avoid confusion. We employ a time window, whose size is w seconds, for event Y_5, and we measure the number of network socket creations (defined as m) during the time window. Then, we calculate the unit number of network socket creations by dividing m by w. We will use this unit value when we measure the event Y_5 (i.e. the event that unit value is larger than σ).

To build a system resource exposure model, we also employ a SVM classifier. However, this model is different from the previous model of a *process reputation*. In the case of the *process reputation model*, we have two classes (benign and malicious), since we could easily obtain both classes of information (i.e. benign domains from contacting domains of normal users and malicious domains from several third parties, such as [60]). Here we mostly have one side of information - benign processes. To get ground truth information of the system resource usages of malware is tricky, some malware may refuse running or behave normally. In addition, they may behave differently, when they are not in control by a bot-master. Thus, even though we obtain the information of malware, it may not represent its behavior clearly. To

address this issue, we only use the processes of known typical (benign) programs. Hence, in this model we consider that we may have only one class of data.

The One-Class SVM (OCSVM) has been proposed to create a model with only one side of information [80]. In this work, we use OCSVM to build the *system resource exposure model*. The OCSVM maps training examples into a feature space and finds a hyperplane which can best separate training examples from the origin. For instance, given training examples x_i, if we assume that there are two planes denoted as "+" and "-" across an origin, the OCSVM will assign all known examples x_i into an one of the planes (i.e. "+" plane or "-" plane).

Similar to generic multi-class SVM classifier, the OCSVM needs to find a hyperplane with maximal geometric margin and it is described as solving the *Lagrangian* equations of (1) and (2) in the above section (more details about the OCSVM can be found in [80]). In this model, we will find a hyperplane to assign all benign examples (i.e. the Y_i features of the benign processes) into the "+" plane and anomaly examples into the "-" plane.

In this model, we will find a hyperplane to assign all benign examples (i.e. the Y_i features of the benign processes) into the "+" plane and anomaly examples into the "-" plane.

Anomalous System Resource Exposure Detection: After creating the model of the system resource exposure, we apply system resource exposure patterns of new processes to the model. If a value of the mapped result of testing data is assigned to "-" plane, we regard the process as anomalous, otherwise it is normal.

3.2.2.3 Network Information Trading Analysis

Typically, most programs will act as clients rather than servers, and most programs will try to gather information rather than distribute information. That is, if

we treat a program as a communication information processing unit, normal client programs are more likely to be an information gaining process. However, a bot will behave differently. Usually, the data that a bot receives is a command from a bot-master, therefore the amount of the data may be small (to minimize the chance of being detected), however the data, which a bot typically sends, will be relatively large as it performs malicious operations in the network. Information theft, DDoS attack, and massive spam sending are good examples.

Lightweight Network Traffic Monitoring: To observe network information trades, a *network sensor* captures network packets between a process and a contacting domain and stores them. An important thing in here is that this sensor monitors network traffic generated by a process not by a host. It could give us more fine-grained observations of network information trading. Our sensor is very simple and lightweight, since it does not need to analyze payload contents and it is not affected by encryption used by bots.

In addition, we monitor the host level network connections to obtain aggregated-view of network information trading. At this time, the sensor only measures the number of outgoing connection trials (i.e. TCP SYN packets and first UDP packets). We believe that this aggregated view gives us a good clue to find DDoS, network scan, or massive spam mail sending.

Network Information Model Creation: We use a simple method to model the network information trade rate, i.e., the ratio of incoming and outgoing packets/bytes exchanged between a process and a remote site in a certain period. We define the number of incoming and outgoing packets as θ_1 and θ_2, and the number of incoming and outgoing bytes as δ_1 and δ_2. Thus, each ratio can be represented as $\frac{\theta_1}{\theta_2}$ and $\frac{\delta_2}{\delta_2}$.

To observe an aggregated-view, we employ a time window w_i for each host i. We

measure how many network connection trials happen in the time window.

Anomalous Network Information Trading Detection: In the case of the fine-grained view, if one or both of the predefined ratio values of a process is (are) smaller than some threshold γ_1 (for packet) and γ_2 (for bytes), we consider the process anomalous, otherwise normal. Also, we consider the network behavior of the host is anomalous, if a host creates network connection trials more than a threshold τ.

3.2.2.4 Correlation Engine

After each module makes its own decision, the *correlation engine* will combine these results and make a final decision using a weighted voting system. At this time, we should determine the weights of the decision of each module.

We can also employ a SVM classification technique to let us know which element (i.e. decision result of the module) is more important (i.e. should have more weight).

To apply the SVM technique, we need training examples of both sides - malicious and benign. However, here is also same issue with the *system resource exposure model* creation mentioned in Section 3.2.2.2. It would be relatively difficult to collect the information of the malicious side. Thus, we decide to employ OCSVM to determine the weight. The way how to determine the weights is same as the method explained in Section 3.2.2.2.

3.3 System Implementation

In this section, we will explain how we implement each module.

3.3.1 Host-Level Modules Implementation

Our *human-process-network correlation analysis module* captures the mouse and keyboard events using Windows system functions. Basically, Windows provides func-

tions to capture the events from external devices [66]. Using these APIs, we implemented the *event sensor* which identifies which process generates the events. We also added the function to store captured information (prcess, event time) to the Shared Memory area.

To capture the outgoing DNS queries, TCP SYN, and UDP packets, we used the WinPcap library [104]. It provides functions to collect raw level network packets on the Windows OS, with little overhead. Moreover, *connection sensor* does not monitor all network packets, but monitor only DNS, TCP SYN and UDP packets. It also reduces the overhead, since those packets comprises a small portion of the all network packets.

Whenever there are network events we should capture, our module also identifies which process produces them and verifies whether the process is related to the human actions or not. However, if a process uses a helper process for a DNS query, we could not directly use it sometimes. To address this problem, we check the process which produces the DNS query automatically and if it is a helper process (e.g., *svchost.exe*), the module waits a DNS reply which contains a IP address of the domain. Then, if there is an automatic connection from a process to that IP address after a DNS query, the module sees that the process issues a DNS query. We use `GetExtendedTcpTable` and `GetExtendedUdpTable` functions to recognize which process creates the connections. If we see the TCP or UDP connection, we will call these functions to identify which process acquires the source port number of the connection.

We implemented the *system resource exposure analysis module* based on EasyHook [24]. EasyHook is a successor and enhanced version of Detours [22], and it provides an interface letting us perform Windows API hooking. The hooking ability allows us to observe how a process works and which system calls are invoked by the process. We selected 28 system calls to understand the exposure patterns of the

process. The selected system calls are related to the access of the system resources, such as files, registries, network sockets, and creation of a process. In addition, we employ TinySVM library [56] to create the *system resource exposure model*.

3.3.2 Network-Level Modules Implementation

To gather network features, the *process reputation analysis module* should utilize multiple network services such as, whois services, blacklist identification services, and web searching services. Whenever the module receives a suspicious process and its contacting domains, it sends a query to multiple network services to gather network features. Also, we use TinySVM library [56] to create the *process reputation model*.

For a *network information trading analysis module*, we build a function to capture network packets using pcap library.

3.3.3 Correlation Engine Implementation

We have implemented the *correlation engine* as an independent process and it will wait for a message from each detection module, and finally decides whether the process is malicious or not. We also use TinySVM library [56] for the *correlation engine*.

3.4 Benign Data Collection and Detection Model Training

In this section, we will show how we collect real world benign data and how we train detection models based on collected data.

3.4.1 Data Collection and Usage

We have installed our modules into six different real-life hosts which are used in the office or home and collected the information of process activities and network behaviors for several days. The collection has been done in working hours on business days. We carefully examine to make sure that there are no malicious programs

(especially bot) in the hosts, thus we consider that the collected data can be used for benign examples. The summary of the data set is represented in Table 3.2.

As shown in the Table 3.2, during the data collection, 86 programs have initiated 85,462 connection trials in total and each host initiated from 37 to 331 connections per hour. In addition, we have collected the system call traces of 78 running processes. Since collecting system call traces can make a system slow, we do not collect system call traces all the time, we randomly select current working processes in each time interval (mainly business hours) and record their system call traces.

We have split the whole data set into two parts based on time-line; (i) first 70% for training our models - **SET-1** and (ii) later 30% for testing of false positives - **SET-2**. **SET-1** contains 62 processes and 84,614 connections and **SET-2** has 16 processes and 848 connections. In this Section, we only use **SET-1** for training and verifying our detection models. We will use **SET-2** in Section 3.5.3 for false positive testing.

Host	*Usage*	*Programs*	*Network Connection Trials*	*Collection Time*
1	Office	7	252	3 hours
2	Office	16	7,859	69 hours
3	Office	19	5,740	147 hours
4	Office	9	5,098	139 hours
5	Home	27	55,586	168 hours
6	Home	8	10,927	83 hours

Table 3.2: Data set summary. (Programs represent the number of programs related to network connection trials)

3.4.2 Automatic Connection Analysis

The *human-process-network correlation analysis module* detects 18,144 automatic connections out of 84,614 connections in total and they are 21.44% of all network connections. When we detect automatic connections, we set the threshold ϵ of the *human-process-network correlation analysis module* to 1 second.

We investigate which process generates automatic connections, and we observe that most automatic connections are issued by Windows system processes or web browser processes. The *svchost.exe*, *spoolsv.exe*, and *taskeng.exe* are the Windows system processes handling network or printing services and they generate 37.73% of automatic connection trials. In the case of the web browsers, the *iexplore.exe*, which is an instance of the Internet Explorer browser, and *chrome.exe*, which is an instance of the Google Chrome browser, are the main sources, and they produce 29.48% of automatic connection trials.

Next, we examine why they generate large number of automatic connection trials. The automatic connections from the Windows network service processes are mainly for sharing resources such as network printer and sharing folders. In the case of the browser processes, they mainly contact well-known web sites or trustworthy web mail sites, such as Google, Yahoo, Google mail (Gmail) and university mail sites to check updated information. Interestingly, they also contact on-line advertisement sites such as *AdSide* [2] very frequently.

Besides this, there are some other processes producing automatic connections frequently. The *vmnat.exe*, which is an instance of VMWare virtual machine [99], relays network requests issued by its guest OS, and most requests are generated by browsers in the guest OS. The *devenv.exe* is a process of Visual Studio 2010 program whose main purpose is developing Windows application programs, but it contacts

Microsoft Developer web sites very frequently. The *ExpressService.exe* is a process of a network storage application program, and it frequently contacts storage hosting sites.

After surveying, we think that we could ignore some automatic connections contacting explicitly benign remote hosts. In the case of the Windows system processes, we can ignore automatic connections heading to local networks, since it is nearly impossible that a bot-master runs a server in the local network. In addition, we may ignore automatic connections visiting trustworthy web sites such as *Google*, *Yahoo*, and *Gmail*.

Based on this idea, we create a whitelist of domain names to reduce unnecessary costs of our system. If an automatic connection tries to contact domains in the whitelist, we consider it benign. To reduce the effect of the whitelist, we only maintain 10 well-known web sites. It has 5 search engine web sites, 3 web mail sites, 1 university site and 1 multimedia site. All sites except for 1 university site are ranked in the top 400 by Alexa [6].

By removing connections to local networks and web sites in the whitelist, we finally have 5,309 automatic connections to 131 different domains, which is 6.27% of all network connection trials.

3.4.3 Process Reputation Model

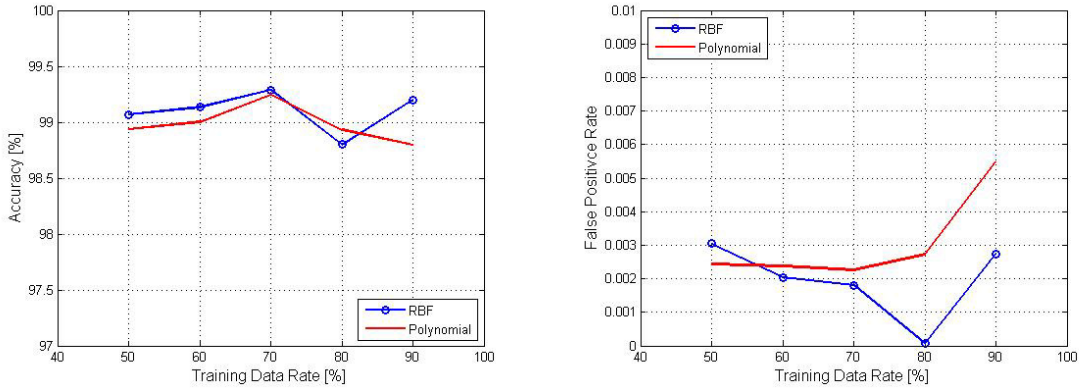
From the collected data, we find that these processes have contacted 7,202 different domains. In order to create the *process reputation model*, we extracted features, which are listed in Table 3.1, from these domains. At this time, we consider that all collected domains are benign, hence we will use them to represent the benign class. We also need malicious domains to represent the malicious class. For the malicious class, we have collected recent 150 malicious domains from [60], and also extracted

features.

Using collected features, we train a SVM classifier for the model. When we train the SVM classifier, we use a RBF (radial basis function) and polynomial as the kernel function.

We validate the model to comprehend how it works. To do this, we first divide our collections of benign and malicious data into training and testing set. Then, we create a model by applying training set and evaluate the model by applying testing set. We change the rate of training and testing data set to evaluate our model more clearly.

Figure 3.2 presents a detection rate and a false positive rate of the *process reputation model*. As shown in Figure 3.2, our model can detect both malicious and benign domains around 99% of rates with very low false positive rates (sometimes 0% of false positive).



(a) detection (b) false positive

Figure 3.2: Process reputation model - detection rate and false positive rate

From this result, we decide that we use the RBF function for the kernel function

for our *process reputation model*, since it shows around 99% detection rate and has very low false positive rates.

3.4.4 System Resource Exposure Model

We analyzed system resource exposure patterns of normal (benign) processes to create the *system resource exposure model*. Here, we will only use information of benign processes and employ a OCSVM classifier to build the model. To do this, we use 62 collected benign processes information. The representative benign processes are the instances of browsers (Chrome, Firefox, and Internet Explorer), mp3 playing programs (Winamp, Gom Player), and p2p client program (Emule). We have extracted each feature defined in Section 3.2.2.2 from them.

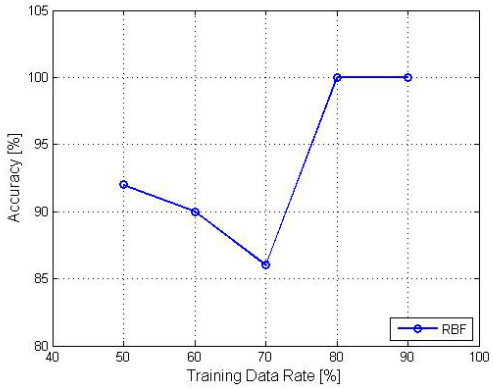
Like the approach mentioned in the *process reputation model* creation, we also divide our data set into training and testing set. The training set is used to build an OCSVM classifier and the testing set is applied to evaluate the obtained OCSVM classifier. We use a RBF function for a kernel function.

We measure a detection rate and a false positive rate of the *system resource exposure model* and present results in Figure 3.3. When we use more than 80% of the collected data for a training, our *system resource exposure model* can classify all the testing data without any error. Thus, our classifier has no false positive in this case.

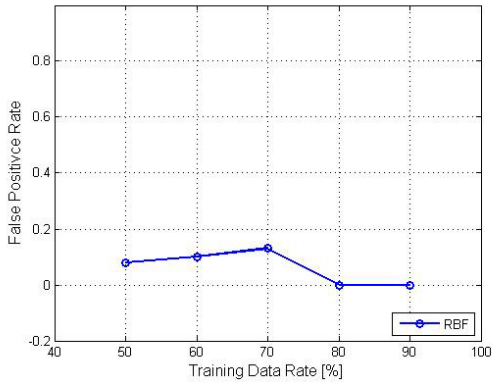
3.4.5 Network Information Trading Model

We analyzed the network flow patterns and verified that most benign programs act as clients not as servers. We measure the ratio between incoming packets (or bytes) and outgoing packets (bytes).

When investigating our data set, we found 92.6% of flows get more packets from the remote host. In the case of bytes, only 81.87% of the connections grab more byte



(a) detection



(b) false positive

Figure 3.3: System resource exposure model - detection rate and false positive rate

from the remote host. The processes, which send more data to remote servers, are applications of updating data and browsers. We tried to understand why they send more information to remote servers, however unfortunately, we could not reveal the reason behind this, since we do not capture payload contents (because of privacy issues). However, we can infer the reason from their main purpose, we think that they may send current status of applications for update.

We can apply the ratio of the incoming and outgoing packets to discriminate a malicious process from a benign process. It is very obvious that if a bot delivers its own information to a master, the ratio can detect them easily.

We investigate the number of network connection trials of a host. To do this, we find what is the maximum network connection trials of a host in a certain time window. We set 10 seconds for a time window and investigate our data set. We find that 21 connection trials (in 10 seconds) are the maximum number. Based on this result, we can set the threshold τ, in this experiment we will use 42 (double the experimental result) for the threshold τ.

3.4.6 Correlation Engine

To calculate weights for the *correlation engine*, we select 27 benign processes, which produce automatic connections frequently. Most of them are processes of browser, multimedia application, and p2p client programs. We collect their detection results which are performed by our detection modules. Then, we train a OCSVM classifier using the collected results and determine the weights.

3.5 Evaluation

In this section, we will provide the results of evaluating the EFFORT system. In order to present the effectiveness clearly, we test each individual module and show how our system finally combine results of each module and make a final decision. In addition, we measure the performance of each module to determine the efficiency of our system.

3.5.1 Test Environment and Data Set for Botnet Detection

We build an isolated virtual environment to run our test. The environment consists of three virutal machines which individually served as an infected host, a controller and a monitor machine. All of them install `Windows XP SP3` operating system with basic software installed, such as Internet Explorer browser and Microsoft Messenger.

The virtual network environment consists of three virtual machines which individually served as an infected host, a controller, and a monitor machine, respectively. At the infected host, we create independent snapshot for each individual malware instance to ensure no cross-infection between different malware. Our host-based modules are also installed to collect data we need. At the monitor machine, we install a fake DNS server to redirect all the DNS queries. At the controller side, we

install various malware controllers we could find to manipulate the infected machine. We intend to reconstruct realistic attack scenarios that a botmaster sends commands to his zombie army.

We have used 15 botnets for evaluation and they are summarized in Table 3.3. Since we just have binary samples of most botnets except three (B1, B2, and B5), we install and simply run them. Among them, 3 botnets (B1, B2, and B5) use IRC protocol, 2 botnets (B4 and B10) use HTTP protocol, 2 botnets (B3 and B4) use P2P protocols, and other 9 botnets use customized protocols. In addition, three botnets (B3, B4, and B7) use encrypted protocols to evade network-level detection. In terms of their actively spreading time in the wild (i.e. when they infect victims highly), it varied from 2003 (B7) to recent (B4). Since these collected botnets can cover diverse cases (e.g. from old one to currently working one, different types of protocols including encryption, and various kinds of malware functionalities), we believe that they can verify our system's effectiveness and efficiency well.

3.5.2 Botnet Detection Results

We begin our evaluation with the test of the *human-process-network correlation analysis module* and it is followed by the results of other modules.

3.5.2.1 Detection Results of Automatic Connections

First, we test whether a bot infected host really generates automatic connections to remote servers and use the *human-process-network correlation analysis module* to detect them. To test this, we installed each bot in a host and leave it without any intervention. After a while, we find that all installed bots issue automatic connections to remote servers to be controlled. All of the automatic connections are reported by our *human-process-network correlation analysis module* and the detected information is delivered to other modules - *process reputation analysis module*, *system exposure*

ID	*Name*	*Protocol*	*Sample Functionalities*
B1	PhatBot	IRC	Steal Key, Spam Mail Send, Network Scan
B2	JarBot	IRC	Kill Process, Steal Key
B3	peacomm	P2P $*$	Other
B4	Waledac	HTTP, P2P $*$	Other
B5	PhaBot.α5	IRC	Other
B6	Flux	Custom	Operate/Modify File, Kill Process, Capture Desktop/Screen,
B7	nuclearRat	Custom $*$	Download Update
B8	BiFrost	Custom	Operate File, Kill Process, Capture Screen, Steal Key
B9	Cone	Custom	Operate file
B10	Http-Pentest	HTTP	Operate File, Kill Process, Capture Screen
B11	Lizard	Custom	Capture Screen, DDoS
B12	PanBot	Custom	Flooding
B13	Penumbra	Custom	Operate File, Create Shell
B14	SeedTrojan	Custom	Download Update
B15	TBBot	Custom	Capture Screen, Create Shell

Table 3.3: Botnets for evaluation (Custom denotes a botnet uses its own protocol and $*$ represents the protocol is encrypted).

analysis module, and *network information trading analysis module*.

3.5.2.2 Detection Results of the Process Reputation Model

The *process reputation analysis module* will receive domain names which a process contacts. Then, the module analyzes the reputation of contacted domains. Since a bot contacts multiple domains in a short time, we analyzed all contacting domains. If the module finds any malicious domain from the contacted domains, it considers the process malicious.

The detection results are shown in Table 3.4. As shown in the Table, the *process reputation analysis module* detects 12 bots and misses 3 bots (B2, B3, and B4).

We investigate why our module missed them. In the case of B2 (peacomm) and

ID	*Contacting Domains*	*Detected Domains*	*ID*	*Contacting Domains*	*Detected Domains*
B1	1	1	B9	2	2
B2	1	-	B10	1	1
B3	2	-	B11	1	1
B4	1	-	B12	1	1
B5	6	2	B13	2	2
B6	3	2	B14	1	1
B7	2	1	B15	1	1
B8	3	2	-		

Table 3.4: Detection results of automatic connections

B3 (Waledac), both bots only contacted the remote server using direct IP addresses instead of domain names. Of course, we can also apply the IP addresses to our module. However, unfortunately, their contacting targets are either private IP addresses (192.168.X.X) or some hosts for which we could not get any useful information from the third parties.

B4 (JarBot) contacts a regular IRC-server and the server has been operated for several years and we could not find any malicious keyword from search results.

3.5.2.3 Detection Results of the System Resource Exposure Model

Receiving the information of an automatic connection trial from the *human-process-network correlation analysis module*, the *system exposure analysis module* began examining the target process.

When we tested the functionality of each malware listed in Table 3.3, we found that the *system exposure analysis module* detects most of the malicious operations. The detection results are summarized in Table 3.5.

It only misses 2 malicious operations of "B6 (Flux)", the first operation is to *operate file* which moves/creates a file in the host, and the second operation is to

Function	*B1*	*B2*	*B3*	*B4*	*B5*	*B6*	*B7*	*B8*	*B9*	*B10*	*B11*	*B12*	*B13*	*B14*	*B15*
Operate file						N		S	S,N	S,N			S,N		
Modify file						S									
Kill process		S,N				S		S		S,N					
Capture Desktop						S,N									
Capture screen						N		S,N		S,N	S,N				S
DDoS											S,N				
Flooding												S,N			
Create Shell													S,N		S,N
Download update							S							S	
Steal key	S,N	S,N						S,N							
Spam Mail Send	S,N														
Network Scan	S,N														
Other Operation			S,N	S,N	S										

Table 3.5: Detection results of the System Resource Exposure and Network Information Trading Module (shaded cells represent functionalities provided by malware. Each "S" and "N" denotes each *system resource exposure analysis* and *network information trading analysis module* detect the functionalities, respectively).

capture screen which captures current screen.

When we analyze their resource exposure patterns, we find that their operations are very similar to normal programs. In the case of *operate file*, malware just creates a file under its permission and reports its success to remote server. In the *capture screen* case, malware captures the current screen, saves its local folder, and delivers captured screen information to a remote server. Both operations (in the point of host view) are very similar to resource exposure patterns of normal applications - creates a file and saves it in its local folder. However, we believe that these operations will be detected by the *network information trading analysis module*, since they distribute more information to the outside. We will show this result later.

3.5.2.4 Detection Results of the Network Information Model

After notifying an automatic connection, the *network information trading analysis module* captures network traffic between a process (not a host) that issued an automatic connection and a remote server. If a process sends more packet/bytes than receives packets/bytes, our module considers it anomalous.

As listed in Table 3.5, the *network trading information analysis module* detects most malicious operations. It misses 7 malicious operations related to *download updates* and *file modification or operation*. In the case of the *download updates*, an infected host gains more data, so that our module can not detect an anomaly. In addition, sometimes a bot-master server sends commands frequently to an infected host, but does not require an answer. In this case, a bot also obtains more data.

In terms of the aggregated-view, our module detects all massive outgoing connection trials, such as DDoS and flooding.

3.5.2.5 Corrleated Detection Results

If the *process-reputation analysis*, *system exposure analysis* and *network information trading analysis* modules determine their decisions, these decision results are delivered to the *correlation engine*. Based on delivered results, the *correlation engine* makes a final decision for a process.

When we test malicious operations, the *correlation engine* can detect all malicious operations and the results are shown in Table 3.6. As we discussed before, even though a module misses an anomaly of an attack, other modules will complement it, thus our combined results can detect all attacks.

Functionality	*B1*	*B2*	*B3*	*B4*	*B5*	*B6*	*B7*	*B8*	*B9*	*B10*	*B11*	*B12*	*B13*	*B14*	*B15*
Operate file						C		C	C	C			C		
Modify file						C									
Kill process		C				C		C		C					
Capture Desktop						C									
Capture screen						C		C		C	C				C
DDoS											C				
Flooding												C			
Create Shell													C		C
Download update							C							C	
Steal key	C	C						C							
Spam Mail Send	C														
Network Scan	C														
Other Operation			C	C	C										

Table 3.6: Detection results of Correlation Engine (shaded cells represent functionalities provided by malware. "C" denotes that correlation engine detects the attack).

3.5.3 False Positive Test Results of Benign Programs

In order to determine whether our modules misjudge benign processes as malicious or not, we have tested 16 benign processes in **SET-2**, which is reserved for false positive test in Section 3.4. They are general Windows applications programs such as browsers, network management programs, office programs and multimedia programs. Even if there are some programs, which produce automatic connections frequently, in this test set, we add 8 more processes in the data set in order to validate our modules more clearly. The added programs are browsers (Internet Exploere and Chrome), p2p client (Emule) and multi-media application (Gom Player). Finally, we have 24 processes.

When we tested 24 processes, 16 processes (8 from **SET-2** and 8 from added ones) produced automatic connections and all of these connections were detected by the *human-process-network correlation analysis module*.

The 16 processes contacted 106 different domains automatically, and the domains were examined by the *process reputation analysis module*, and our module decided that all domains were benign so that it finally determined that the reputations of processes are good.

In addition, the *system exposure analysis module* investigated these 16 processes in detail, and decided that they were all benign. In detail, we found that some of them touched Windows system folders (specifically, browsers and a network storage program), however we could not observe any process that accesses critical registries or creates a lot of network sockets.

In the case of the *network information trading analysis module*, it detects 6 processes as anomalous and they are all browsers. Unfortunately, since we do not have packet payloads (because of privacy issues), we can not understand why it happens clearly. However, we presume that they send current status of web applications (in the host) to remote servers.

Even though one of our modules considers some connection trials anomalous, the *correlation analyzer* makes the right decision for all cases (i.e. all are benign). Since the *correlation analyzer* combines classification information from three different modules and makes a decision, even if one module makes a wrong decision, it can be complemented by other modules.

3.5.4 Performance

We have measured the overhead of each module to verify the efficiency of the proposed system. In this measurement, we want to show how our modules affect the system and other applications.

To do this, we used two metrics in measuring the overhead; *memory usage* and *program delay*. The *memory usage* presents how our modules consume the resources of memory and the *program delay* represents how our modules make other programs slow when our modules are running. To measure the *program delay*, we selected three types of test programs, Internet Explorer which produces network connections frequently, Calculator which mainly uses CPU, and Notepad which produces some

disk operations. We compared the running time of these programs between when our modules are running and not[‡]. In the case of the Internet Explorer, we simply visit one web site and close. We just divide some numbers using Calculator and read a file and save it to other file using Notepad.

We will start performance measurements for the *human-process-network correlation analysis* and *system exposure analysis* modules, since they are installed into a host and affect the performance of the host directly.

Overhead of Human-Process-Network Correlation Analysis Module: At first, we have measured the overhead of the *human-process-network correlation analysis module*. As shown in Table 3.7, the overhead of this module is 1.35% at maximum and even 0% (i.e. our module does not affect some programs at all). In addition, this module only consumes 1.81 MB of memory. Thus, we believe that the overhead of this module is nearly ignorable.

Item	*w/o module*	*with module*	*overhead (%)*
Internet Explorer	177 (ms)	179.4 (ms)	1.35%
Notepad	4,206 (ms)	4,218 (ms)	0.29%
Calculator	26 (ms)	26 (ms)	0%

Table 3.7: Overhead of human-process-network correlation analysis module.

Overhead of System Exposure Analysis Module: In addition, we have measured the overhead of the *system exposure analysis module*. This module will show relatively high overhead. Since it has to monitor a lot of system calls which are frequently called by a process, it is very hard to reduce the overhead of this module.

[‡]When we performed this test, we run a test program 5 times and found the average value

When we measure the overhead, we observe that it consumes 9.18 MB of memory and produces overhead around 6% at maximum and 1 % at minimum, as presented in Table 3.8.

The overhead seems to be not so high and even very low in a case. In addition, our module does *not* need to monitor processes *all* the time. The *system exposure analysis module* only investigates a process when the process issues automatic connections to untrustable remote sites. As we discussed in Section 3.4, the automatic connections to untrustable sites happen very rarely, at most 8 - 9 connections/hour in our dataset. Hence, we also consider that the overhead from this module is low.

Item	*w/o module*	*with module*	*overhead (%)*
Internet Explorer	177 (ms)	185.1 (ms)	4.51%
Notepad	4,206 (ms)	4,463 (ms)	6.12%
Calculator	26 (ms)	26.3 (ms)	1.15%

Table 3.8: Overhead of system exposure analysis module.

Overhead of Other Modules: Unlike *human-process-network correlation* and *system exposure* modules, the other modules of the *process reputation analysis module*, the *network information trading analysis module*, and the *correlation analyzer* exists in the other host(s) and they mainly observe network traffic or third party information. Thus they do not affect the performance of the host directly.

3.5.5 Summary of Evaluation Results

We will summarize the evaluation results of our system by showing how our system is effective and efficient. In terms of the effectiveness, we evaluate that our system detects all malicious operations from the various kinds of botnets without any

false positive. Even if botnets employ encrypted messages for their C& C channels to evade detection, our system can detect them with the help of combined results from each module. In terms of the efficiency, we evaluate that our system does not put high overhead on the hosts. Thus, we believe that our system can be used as a real-time monitoring/detecting system.

3.6 Related Work

There are several approaches to detect bots at the network level. Karasaridis et al., proposed an approach of detecting bots based on network traffic pattern analysis [49]. In [108], the authors proposed a system detecting malware (also bots) through aggregating network flows. Gu et al., proposed several promising approaches which detect bots through network behavior correlated analysis [34, 32, 35]. Our work is different from the above work, since we design several new network level sensors and we consider host level features as well. The host-network cooperation allows us to detect malware more effectively.

Detecting bots in the host level is also popular due to its effectiveness. In [11, 92], the authors tainted all memories and resources touched by a process to determine whether it was malicious or not. In [53], Kolbitsch et al., provided an way of detecting malware through examining the system call sequences/graphs. Although they detected malware accurately, they could cause high overhead in the host. Our work designs several new host level sensors without analyzing *all* running processes *all* the time, only investigating the process when necessity.

In [57], Lanzi et al., provided an approach of detecting malware at the host by investigating the access patterns of the process. Our work differs from [57], since we use different features in host level (e.g., network socket creation) and detection models. Moreover, our work analyzes the binary only when necessary, and it can

significantly reduce the overhead.

There is also an approach to detect bots combining information obtained from both of the host and network level [110]. This work uses network sensors to trigger host analysis, thus it suffers from the same limitations of network-based detection approaches. If a bot can evade the network level monitoring, it evades their detection system. Our work differs from their work in that we use different features/models and our coordination is triggered from host level features.

In [97], Trestian et al., uses the Google search engine to comprehend the network traffic, and our approach of identifying reputation of a process also employs search engines. Our work differs in its main goal (for security) and detection models. Also, while they only use IP address for their query, we use the process and domain name as well.

3.7 Limitations

Like many detection systems, our system is not perfect. In current implementation, we do not consider the protection of our host modules yet. In addition, some kernel malware like rootkit could fool our host modules such as faking human-driven events as OS level. This problem can be solved by employing hardware/TPM [37] or hypervisor-based introspection techniques [28, 45], which is our future work.

Our reputation module mainly assumes that bots will use DNS to contact their master. However not all bots may use DNS, some bots use IP address directly. Our reputation model is easily to be extended to handle IP address as well.

3.8 Summary of this Chapter

It is a very challenging problem to detect bots effectively and efficiently. In this work, we studied various kinds of features in the network and the host level and chose promising features that enable to detect bots effectively. Then, we proposed

a novel detection approach with correlative and coordinated analysis of each feature to detect bots efficiently. In our evaluation on real world data, we show that our features are feasible to detect bots effectively, and verify our system can detect bots accurately without any false positive.

4. SECURING FUTURE NETWORK ENVIRONMENTS

4.1 Introduction

In this chapter, we present how we design a future network architecture with security. It is not easy to redesign a whole network architecture at initial stage, but we can provide a new framework for security on a possible network technology that is considered as a future network. This chapter suggests a possible solution that we can make a future network secure, and it contains several possible example cases for security applications.

To design a new secure network architecture, we employ the technology of Software Defined Networking (SDN). SDN enabled networks distinguish themselves from legacy network infrastructures by dramatically rethinking the relationship between the data and control planes of the network device. SDN embraces the paradigm of highly programmable switch infrastructures [62], enabling software to compute an optimal flow routing decision on demand. For modern networks, which must increasingly deal with host virtualization and dynamic application migration, SDN may offer the agility needed to handle dynamic network orchestration beyond that which traditional networks can achieve.

For an SDN enabled switch, the data plane is made programmable, where flows are dynamically specified within a *flow table*. The flow table contains a set of *flow rules*, which specify how the data plane should process all active network flows. In short, the control plane provides the basic instructions that govern how to forward, modify, or drop each packet that traverses the SDN enabled switch. The switch's control plane is simplified to support a protocol (e.g., OpenFlow protocol), which allows the switch to communicate statistics and new flow requests to an external

control plane (e.g., *controller* in OpenFlow). In return, it receives flow rules that extend its flow table ruleset.

The control plane is situated above a set of SDN enabled switches, often on lower-cost commodity hardware. It is the coordination point for the network's flow rule production logic, providing necessary flow rule updates to the switch, either in response to new flow requests or to reprogram the switch when conditions change. As a controller may communicate with multiple SDN enabled switches simultaneously, it can distribute a set of coordinated flow rules across the switches to direct routing or optimize tunneling in a way that may dramatically improve the efficiency of traffic flows. The controller also provides an API to enable one to develop *SDN applications*, which implement the logic needed to formulate new flow rules. It is this application layer that is our central focus.

From a network security perspective, SDN technology offers researchers with an unprecedented singular point of control over the network flow routing decisions across the data planes of all OF-enabled network components. Using SDN technology, an *SDN security app* can implement much more complex logic than simplifying halting or forwarding a flow. Such applications can incorporate stateful flow rule production logic to implement complex quarantine procedures, or malicious connection migration functions that can redirect malicious network flows in ways not easily perceived by the flow participants. Flow-based security detection algorithms can also be redesigned as SDN security apps, but implemented much more concisely and deployed more efficiently, as we illustrate in examples within this chapter.

We introduce a new security application development framework called *FRESCO*. *FRESCO* is intended to address several key issues that can accelerate the composition of new SDN enabled security services. *FRESCO* exports a scripting API that enables security practitioners to code security monitoring and threat detection logic as

modular libraries. These modular libraries represent the elementary processing units in *FRESCO*, and may be shared and linked together to provide complex network defense applications.

FRESCO currently includes a library of 16 commonly reusable modules, which we intend to expand over time. Ideally, more sophisticated security modules can be built by connecting basic *FRESCO modules*. Each *FRESCO* module includes five interfaces: (*i*) input, (*ii*) output, (*iii*) event, (*iv*) parameter, and (*v*) action. By simply assigning values to each interface and connecting necessary modules, a *FRESCO* developer can replicate a range of essential security functions, such as firewalls, scan detectors, attack deflectors, or IDS detection logic.

FRESCO modules can also produce flow rules, and thus provide an efficient means to implement security directives to counter threats that may be reported by other *FRESCO* detection modules. Our *FRESCO* modules incorporate several security functions ranging from simple address blocking to complex flow redirection procedures (dynamic quarantine, or reflecting remote scanners into a honeynet, and so on). *FRESCO* also incorporates an API that allows existing DPI-based legacy security tools (e.g., BotHunter [34]) to invoke *FRESCO's* countermeasure modules. Through this API, we can construct an efficient countermeasure application, which monitors security alerts from a range of legacy IDS and anti-malware applications and triggers the appropriate *FRESCO* response module to reprogram the data planes of all switches in the OpenFlow network.

Contributions. In summary, our primary contribution is the introduction of *FRESCO*, which simplifies the development and deployment of complex security services for SDN networks. To this end, we describe

- *FRESCO*: a new application development framework to assist researchers in

prototyping new composable security services in SDN enabled networks. *FRESCO* scripts can be defined in a manner agnostic to SDN control plane implementation or switch hardware (an important feature given the rapid evolution of the protocol standard).

- A collection of SDN security mitigation directives (*FRESCO* modules) and APIs to enable legacy applications to trigger these modules. Using *FRESCO*, security projects could integrate alarms from legacy network security DPI-based applications as inputs to *FRESCO* detection scripts or as triggers that invoke *FRESCO* response scripts that generate new flow rules.

- Several exemplar security applications demonstrate both threat detection and mitigation in a SDN network, including scan detectors [48, 82, 47] and BotMiner [33]. We further show that existing security applications can be easily created with the use of *FRESCO*. For example, our *FRESCO* implementations demonstrate over 90% reduction in lines of code when compared to standard implementations and recently published another SDN implementations [63].

- A performance evaluation of *FRESCO*, which shows promise in developing SDN security services that introduce minimal overhead for use in live network environments.

4.2 Term Definition

Note that in this chapter, we use OpenFlow [62] network environment for our SDN testbed, and we use the terms of both OpenFlow and SDN when we talk about SDN technology. OpenFlow is the most common technology for SDN, and many projects and products in both of academia and industry are implemented with OpenFlow.

Thus, we believe that it is acceptable to use OpenFlow in developing our framework for SDN.

In addition, we use the term of controller to denote the control plane and the term of OpenFlow/SDN enabled switch for the data plane supporting SDN technology. The term of controller is commonly used in OpenFlow specification, and it is widely used term. The term of application is used to denote a network application running on the control plane, and it denotes a software program that tries to control network flows with the help of SDN technology. Sometimes, we use OF instead of OpenFlow, and it is the short form of OpenFlow.

4.3 Motivation

Our intent is to design an application framework that enables the modular design of complex SDN enabled network security services, which can be built from smaller sharable libraries of security functions. Before presenting *FRESCO*'s design, we first review some of the challenges that motivate the features of our framework.

4.3.1 The Policy Enforcement Challenge

The first challenge, which we call the policy enforcement challenge, stems from the fact that SDN provides no inherent mechanisms to reconcile rule conflicts as competing SDN applications assert new rules into a switch. For example, a set of rules designed to quarantine an internal compute server might subsequently be overridden by a load balancing application that may determine that the quarantined host is now the least loaded server. One needs a mechanism to ensure that flow rules produced by a security application will take precedence over those produced from non-security aware applications. SDN also incorporate a packet alteration functions (i.e., the `set` action), specifiable within its flow rule format. This feature enables virtual tunnels between hosts, such that a virtual tunnel can be used to circumvent a

flow rule that was inserted to prevent two hosts from establishing a connection [74].

To address this challenge, we have designed and implemented a security enforcement kernel (SEK), which is integrated directly into the control plane upon which *FRESCO* operates. In [74], we present the design of our SEK, along with a prototype implementation called *FortNOX*, which we integrated into the open-source NOX [36] controller. FortNOX offers several important features upon which *FRESCO* relies to ensure that flow rules derived from security service are prioritized and enforced over competing flow rules produced by non-security-critical applications:

- Rule source identification: the SEK introduces a trust model that allows *FRESCO* applications to digitally sign each candidate flow rule, thus enabling the SEK to determine if a candidate flow rule was produced by a FRESCO security module, an SDN application, or by a network administrator.

- Rule conflict detection: To detect conflicts between a candidate rule set and the set of rules currently active in the switch. The SEK incorporate an inline rule conflict analysis algorithm called *alias set rule reduction*, which detects flow rule conflicts, including those that arise through set actions that are used to produce virtual tunnels.

- Conflict resolution: When a conflict arises, the SEK applies a hierarchical authority model that enables a candidate rule to override (replace) an existing flow rule when the digital signature of the rule source is deemed to possess more authority than the source whose rule is in conflict.

4.3.2 The Information Deficiency Challenge

The control planes in SDN do not uniformly capture and store TCP session information, among other key state tracking data, which is often required to de-

velop security functionality (e.g., TCP connection status, IP reputation). We call this an information deficiency challenge. The *FRESCO* architecture incorporates a database module (*FRESCO*-DB) that simplifies storage and management of session state shared across applications. *FRESCO* also exports a high-level API in the *FRESCO* language that abstracts away complexities relating to switch management and specific controller implementations. This abstraction is a critical feature to enable module sharing across SDN network instances that may vary in the control plane

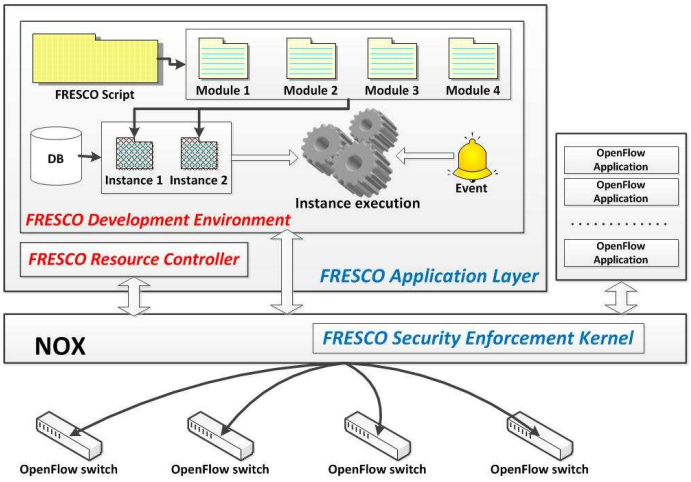


Figure 4.1: High-level overview of the *FRESCO* architecture.

4.3.3 The Security Service Composition Challenge

The *FRESCO* framework incorporates a modular and composable design architecture, inspired by the Click router architecture [52], which fosters rapid and collaborative development of applications through module composition. For example, a security module design to recognize certain traffic patterns that may represent a threat should be easily linkable to a variety of potential threat mitigation mod-

ules that, when triggered by the detection module, produce appropriate flow rule responses. *FRESCO* incorporates a scripting language that enables the linking of modules through data sharing and event triggering. Another important challenge is the need to provide an API that can facilitate flow rule production decisions using information produced from legacy DPI-based security applications (such as IDS or anti-malware applications).

4.3.4 The Threat Response Translation Challenge

The SDN technology enables the controlling software layer to communicate flow handling instructions to the data plane. However, while network security technologies do indeed produce threat alerts applicable to responses for individual flows, these technologies also have a need to express more complex (even stateful) security response directives that may span many flow rules, or even address network-wide attack scenarios. We call this the threat response translation challenge.

For example, one may wish to define a notion of host quarantine, in which all flows from an infected internal machine are blocked, with the exception that the machine's web traffic should be redirected to a web server that returns quarantine notification pages to the machine's user. One might also wish to define redirection directives that will silently redirect flows from a detected hostile external entity away from an internal production network and into a honeynet for analysis. One might even want to produce a network-wide response to shun malicious traffic, or alternatively, incorporate high-priority flow rules to ensure that emergency administrative flows succeed during a DOS attack.

Such security directives may require a complex set of flow rule production logic, which is also ideally sharable as a countermeasure library that could be coupled with many different detection algorithms.

4.4 FRESCO Design

The *FRESCO* framework consists of an application layer (which provides an interpreter and APIs to support composable application development) and a security enforcement kernel (SEK, which enforces the policy actions from developed security applications), as illustrated in Figure 4.1. Both components are integrated into NOX, an open-source OpenFlow controller.

FRESCO's application layer is implemented using NOX python modules, which are extended through FRESCO's APIs to provide two key developer functions: (*i*) a *FRESCO* Development Environment [**DE**], and (*ii*) a Resource Controller [**RC**], which provides FRESCO application developers with OpenFlow switch- and controller-agnostic access to network flow events and statistics.

Developers use the *FRESCO script* language to instantiate and define the interactions between the NOX python security modules (we present *FRESCO's* scripting language in Section 4.5.3). These scripts invoke *FRESCO*-internal modules, which are instantiated to form a security application that is driven by the input specified via the FRESCO scripts (e.g., TCP session and network state information) and accessed via *FRESCO's* DE database API. These instantiated modules are triggered (executed) by *FRESCO* DE as the triggering input events are received. FRESCO modules may also produce new flow rules, such as in response to a perceived security threat, which are then processed by the controller's security enforcement kernel [**SEK**] (Section 4.6).

4.5 FRESCO Application Layer

The basic operating unit in the *FRESCO* framework is called a *module*. A module is the most important element of *FRESCO*. All security functions running on *FRESCO* are realized through an assemblage of modules. Modules are defined as

Python objects that include five interface types: (*i*) *input*, (*ii*) *output*, (iii) *parameter*, (iv) *action*, and (v) *event*. As their names imply, *input* and *output* represent the interfaces that receive and transmit values for the module. A *parameter* is used to define the module's configuration or initialization values. A module can also define an *action* to implement a specific operation on network packets or flows. An *event* is used to notify a module when it is time to perform an action.

A module is implemented as an event-driven processing function. A security function can be realized by a single module or may be composed into a directed graph of processing to implement more complex security services. For example, if a user desires to build a naive *port comparator* application whose function is to drop all HTTP packets, this function can be realized by combining two modules. The first module has *input*, *output*, *parameter*, and *event*. The *input* of the first module is the destination port value of a packet, its *parameter* is the integer value 80, an *event* is triggered whenever a new flow arrives, and *output* is the result of comparing the *input* destination port value and parameter value 80. We pass the *output* results of the first module as *input* of the second module and we assign drop and forward *actions* to the second module. In addition, the second module performs its function whenever it is pushed as an *input*. Hence, the *event* of this module is set to be *push*. A module diagram and modules representing this example scenario are shown in Figure 4.2.

An *action* is an operation to handle network packets (or flows). The actions provided by *FRESCO* derive from the actions supported by the NOX OpenFlow controller. The OpenFlow standard specifies three required actions, which should be supported by all OpenFlow network switches, and four optional actions, which might be supported by OpenFlow network switches [69]. OpenFlow requires support for three basic actions: (*i*) *drop*, which drops a packet, (*ii*) *output*, which forwards a packet to a defined port (in this work, we sometimes use the term *forward* to denote

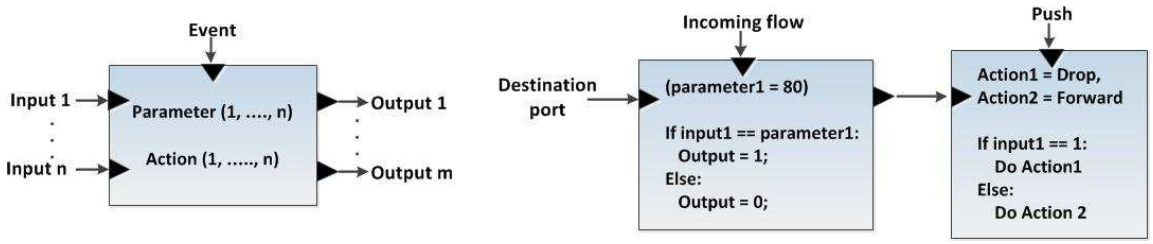


Figure 4.2: Illustration of *FRESCO* module design (left: model diagram; right: naive port comparator application)

the output action), and (*iii*) *group*, which processes a packet through the specified group. As these actions must be supported by all OpenFlow network switches, *FRESCO* also exports them to higher-level applications.

One optional action of interest is the *set action*, which enables the switch to rewrite a matching packet's header fields (e.g., the source IP, destination port) to enable such features as flow path redirection. Because one of the primary goals of *FRESCO* is to simplify development of security functions, *FRESCO* handles possible issues related to the *set action* by breaking the *set action* into three more specific actions: *redirect*, *mirror*, and *quarantine*. Through the *redirect action*, an application can redirect network packets to a host without explicitly maintaining state and dealing with address translation. *FRESCO* offloads session management tasks from applications and automatically changes the source and destination IP address to handle redirects. The *mirror action* copies an incoming packet and forwards it to a mirror port for further analysis. The functionality may be used to send a packet to a feature or other packet analysis systems. The *quarantine action* isolates a flow from the network. Quarantine does not mean dropping a particular flow, but rather, *FRESCO* attaches a tag to each packet to denote a suspicious (or malicious) packet. If a packet has the tag, then this packet can traverse only to allowed hosts (viz., a

FRESCO script can fishbowl an infected host into an isolated network using packet tags).

4.5.1 FRESCO Development Environment

The *FRESCO development environment (DE)* provides security researchers with useful information and tools to synthesize security controls. To realize this goal, we design the *FRESCO* DE with two considerations. First, this environment must export an API that allows the developer to detect threats and assert flow constraints while abstracting the NOX implementation and OF protocol complexities. Second, the component must relieve applications from the need to perform redundant data collection and management tasks that are common across network security applications. The *FRESCO* development environment provides four main functions: (*i*) script-to-module translation, (*ii*) database management, (*iii*) event management, and (*iv*) instance execution.

Script translation: This function automatically translates *FRESCO* scripts to modules, and creates instances from modules, thus abstracting the implementation complexities of producing OF controller extensions. It is also responsible for validating the registration of modules. Registration is performed via a registration API, which enables an authorized administrator to generate a *FRESCO* application ID and an encryption key pair. The developer embeds the registered application ID into the *FRESCO* script, and then encrypts the script with the supplied private key. The naming convention of *FRESCO* applications incorporates the application ID, which is then used by *FRESCO* to associate the appropriate public key with the application. In addition to registering modules, the module manager coordinates how modules are connected to each other and delivers input and event values to each module.

Database management: The DB manager collects various kinds of network and switch state information, and provides an interface for an instance to use the information. It provides its own storage mechanism that we call the *FRESCO-DataBase (F-DB)*, which enables one to share state information across modules. For example, if an instance wants to monitor the number of transferred packets by an OpenFlow enabled switch, it can simply request the F-DB for this information. In addition, this database can be used to temporarily store an instance.

Event management: The event manager notifies an instance about the occurrence of predefined events. It checks whether the registered events are triggered, and if so delivers these events to an instance. *FRESCO* supports many different kinds of events, including flow arrivals, denied connections, and session resets. In addition, the event manager exposes an API that enables event reporting from legacy DPI-based security applications, such as Snort [89] or BotHunter [34]. The security community has developed a rich set of network-based threat monitoring services, and the event manager's API enables one to tigger instances that incorporate flow rule response logic. *

Instance execution: This function loads the created instances into memory to be run over the *FRESCO* framework. During load time, *FRESCO* decrypts the application using the associated public key, and confirms that the ID embedded in the script corresponds to the appropriate public key. The application then operates with the authority granted to this application ID at registration time.

4.5.2 FRESCO Resource Controller

The *FRESCO* resource controller monitors OpenFlow network switches and keeps track of their status. A flow rule that is distributed from a *FRESCO* application is

*The example case for this scenario is shown in section 4.7.

inserted into a flow table in an OpenFlow switch. Because the flow table has a limit on the number of entries it can hold, it is possible that a flow rule from a *FRESCO* application cannot be inserted into the flow table. However, because flow rules from a *FRESCO* application deal with security policy enforcement, such flow rules require immediate installation into the flow table of an OpenFlow network switch. Thus, *FRESCO* may forcibly evict some old or stale flow rules, both *FRESCO* and non-*FRESCO*, from the switch flow table to make space for new flow rules. This operation is done by the resource controller. Garbage collecting inactive *FRESCO* rules does not compromise the network security policy: if a prohibited flow is re-attempted later, the *FRESCO* SEK will prevent other OF applications from performing the flow setup.

Variable	***Explanation***	***Possible Values***
instance name (#input)(#output)	denotes an instance name (should be unique)	(#input) and (#output) denote the number of inputs and outputs
type: [module]	denotes a module for this instance	[module] names an existing module
input: a_1,a_2,...	denotes input items for a module	a_n may be set of flows, packets or integer values
output: b_1, b_2,...	denotes output items for a module	b_n may be set of flows, packets or integer values
parameter: c_1,c_2,...	denotes configuration values of a module	c_n may be real numbers or strings
event: d_1,d_2,...	denotes events delivered to a module	d_n may be any predefined string
action : condition ? action,...	denotes set of conditions and actions performed in the module	condition follows the same syntax of *if condition* of python language; action may be one of the following strings (DROP, FORWARD, REDIRECT, MIRROR, QUARANTINE)
{ }	denotes the module start ({) and end (})	-

Table 4.1: Key variables in the *FRESCO* scripting language

The resource controller performs two main functions. The first function, which we call the *switch monitor*, periodically collects switch status information, such as

```
port_comparator (1)(1) {
   type:Comparator
   event:INCOMING_FLOW
   input:destination_port
   output:comparison_result
   parameter:80
/* no actions are defined */
   action: -
}
```

```
do_action (1)(0) {
   type:ActionHandler
   event:PUSH
   input:comparison_result
   output: - /* no outputs are defined */
   parameter: - /* no parameters are defined */
/* if input equals to 1, drop, otherwise, forward */
   action:comparator_result == 1 ? DROP : FORWARD
}
```

Figure 4.3: *FRESCO* script with two connecting modules used to build the naive port comparator

the number of empty flow entries, and stores the collected information in the switch status table. The second component, i.e., the *garbage collection*, checks the switch status table to monitor whether the flow table in an OpenFlow switch is nearing capacity. If the availability of a flow table becomes lower than a threshold value (θ), the garbage collector identifies and evicts the least active flow, using least frequently used (LFU) as *FRESCO*'s default policy.

4.5.3 FRESCO Script Language

To simplify development of security applications, *FRESCO* provides its own script language to assist developers in composing security functions from elementary modules. The textual language, modeled after the Click language [52], requires the definition of six different variables per instance of modular element: (*i*) `type`, (*ii*) `input`, (*iii*) `output`, (*iv*) `parameter`, (*v*) `action`, and (*vi*) `event`.

To configure modules through a *FRESCO* script, developers must first create an instance of a module, and this instance information is defined in *type* variable. For example, to use a function that performs a specific action, a developer can create an instance of the ActionHandler module (denoted as `type:ActionHandler` within a *FRESCO* script).

Developers can specify a script's input and output, and register events for it

to process by defining the script's *input*, *output*, *parameter*, and *event* variables. Multiple value sets for these variables (e.g., specifying two data inputs to `input`) may be defined by using a comma as the field separator.

Defining an instance is very similar to defining a function in C or C++. A module starts with the module name, two variables for representing the number of inputs and outputs, and left braces (i.e., {). The numbers of inputs and outputs are used to sanity check the script during module translation. Like C or C++ functions, a module definition ends with a right brace (i.e., }).

The `action` variable represents actions that a module will perform based on some conditions, where the conditions are determined by one of the *input* items. There may be multiple conditions in the `action`, which are separated by semicolons. We summarize these variables in Table 4.1, and Figure 4.3 shows example scripts of the port comparator application shown in Figure 4.2 (right) with two connecting modules.

***FRESCO* Script Execution:** We use a simple running example, shown in Figures 4.3 and 4.4, to illustrate the execution of a FRESCO script. First, an administrator composes a FRESCO script (shown in Figure 4.3) (1), and loads it into *FRESCO* (2). Next, when Host A sends a packet to port 80 of Host B through an OpenFlow switch, as illustrated in Figure 4.4 (3), this packet delivery event is reported to the FRESCO DE (4). The *FRESCO* DE creates instances from modules defined in the *FRESCO* script (i.e., port_comparator instance from comparator module and do_action instance from ActionHandler module) and dynamically loads them. The *FRESCO* DE runs each instance (5, 6), and when it receives an action from the do_action module (i.e., drop) (7), it translates this action into flow rules, which can be understood by an OpenFlow switch. Finally, these flow rules are installed into the switch through the *FRESCO* SEK (8).

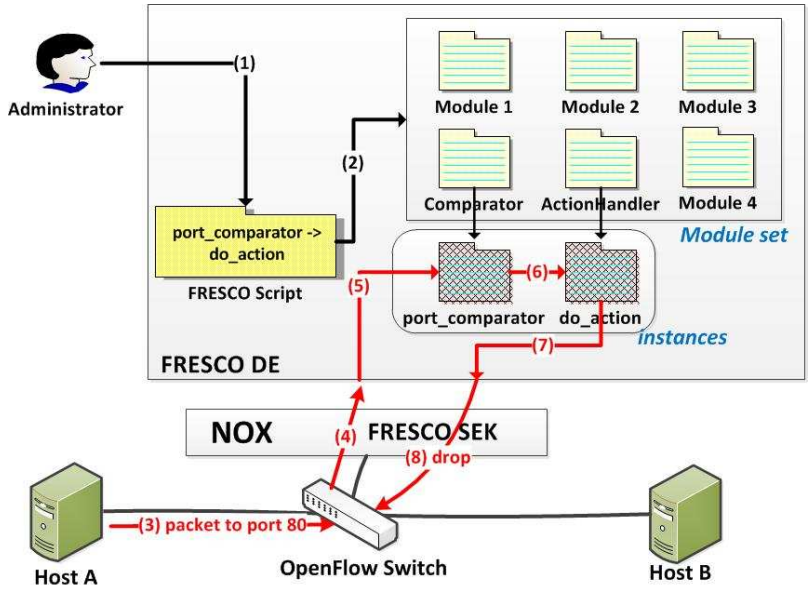


Figure 4.4: Operational illustration of running *FRESCO* script (case of the *FRESCO* script shown in Figure 4.3)

4.6 FRESCO Security Enforcement Kernel

Security applications developed in *FRESCO* scripts can enforce diverse security policies, such as DROP, REDIRECT, QUARANTINE, to react to network threats by simply setting an *action* variable, as listed in Table 4.1. These high-level security policies can help developers focus on implementing security applications, and these policies will be automatically translated into flow rules for OpenFlow enabled switches by *FRESCO* DE (e.g., the REDIRECT action will be translated into three flow rules). Thus, developers do not need to care about network-level flow rules.

However, when *FRESCO* DE enforces translated flow rules to switches, it will face a new challenge, which stems from the fact that OpenFlow provides no inherent mechanisms to reconcile rule conflicts as competing OpenFlow applications assert new rules into a switch. For example, a set of rules designed to quarantine an internal computing server (i.e., the QUARANTINE action in a *FRESCO* script) might

subsequently be overridden by a load-balancing application that may determine that the quarantined host is now the least-loaded server. One needs a mechanism to ensure that flow rules produced by a security application will take precedence over those produced from non-security-aware applications. OpenFlow also incorporates a packet alteration functions (i.e., the `set` action), specifiable within its flow rule format. This feature enables virtual tunnels between hosts, such that a virtual tunnel can be used to circumvent a flow rule that was inserted to prevent two hosts from establishing a connection.

To address this issue, *FRESCO* incorporates a security enforcement kernel (SEK), which is integrated directly into the OpenFlow controller upon which FRESCO operates. A more complete discussion of *FRESCO* SEK is provided in a published workshop paper [74]. *FRESCO* SEK offers several important features upon which FRESCO relies to ensure that flow rules derived from security services are prioritized and enforced over competing flow rules produced by non-security-critical applications:

- Rule source identification: The SEK introduces a trust model that allows FRESCO applications to digitally sign each candidate flow rule, thus enabling the SEK to determine if a candidate flow rule was produced by a FRESCO security module, by an OpenFlow application, or by a network administrator.

- Rule conflict detection: To detect conflicts between a candidate rule set and the set of rules currently active in the switch, the SEK incorporates an inline rule conflict analysis algorithm called *alias set rule reduction*, which detects flow rule conflicts, including those that arise through set actions that are used to produce virtual tunnels. Since this is not the main focus of this chapter, we include a relatively more detailed description of our rule conflict detection

algorithm in the following subsections.

- Conflict resolution: When a conflict arises, the SEK applies a hierarchical authority model that enables a candidate rule to override (replace) an existing flow rule when the digital signature of the rule source is deemed to possess more authority than the source whose rule is in conflict.

4.6.1 FRESCO Security Enforcement Kernel Implementation

It is possible that the flow rules created by non-security-related SDN applications conflict with the flow constraints distributed by *FRESCO* applications. A conflict arises when one or more flow rules would allow a flow from one end point to another that is specifically prohibited by a flow constraint rule produced by a *FRESCO* application. To manage *FRESCO* flow constraints and perform conflict evaluation, we introduce the *FRESCO* SEK as an embedded NOX extension. Since we use NOX for our SEK implementation, we use OpenFlow protocol and OpenFlow applications for testing its implementation.

Two main components of OpenFlow rules are match conditions and actions. The former specifies packet header fields that must match for the rule's associated actions to trigger. The *FRESCO* SEK maintains the set of active constraint rules produced by registered *FRESCO* modules. Constraint rules inserted into the *FRESCO* SEK *security constraints table* are considered active, and must be explicitly deactivated by a registered *FRESCO* module. Because non-*FRESCO* applications can publish flow rules that potentially violate *FRESCO*'s network security policy, The *FRESCO* SEK employs two protection mechanisms to prevent such violations. The first mechanism is *rule prioritization*, in which flow rules produced by *FRESCO* applications are published to the switch using the highest rule priority. This immediately overrides any active flow rules in the switch's flow table that may contradict *FRESCO*'s security

policy. Second, the *FRESCO* SEK applies a conflict detection algorithm between each new flow rule and the security constraints table, rejecting the new flow rule if a conflict is detected. Conflict detection is performed in two passes: alias set rule reduction, and then rule set conflict evaluation.

A conflict can also happen between security constraints enforced by different *FRESCO* applications. In this case, the *FRESCO* SEK can still detect conflicts but it needs to determine which constraint should be enforced. By default, *FRESCO* SEK keeps the first enforced constraint (i.e., ignore following conflicted constraint), but it is easy to be configured by the administrators to apply other approaches (e.g., keep the last enforced constraint, or based on some priority settings).

4.6.1.1 Alias Set Rule Reduction

To detect conflicts between a candidate rule set and *FRESCO*'s constraint rule sets, the source and destination IP addresses, their ports, and wild card members † for each rule in a rule set are used to derive rules with alias sets representing IP addresses and ports. The initial alias sets contain the first rule's IP addresses, network masks, and ports (where 0 [zero] represents any port). If the rule's action causes a field substitution via a *set action*, the resultant value is added to the appropriate alias set. These sets are then compared to the next rule's alias sets. If there is an intersection between both the source and address sets, the union of the respective sets is used as the subsequent rule's alias sets. For example, given the *FRESCO* rule,

$$a \rightarrow \text{b drop packet} \tag{4.1}$$

its source alias set is (a), while its destination alias set is (b). The derived rule is

†For OpenFlow 1.1, the examined members include the source and destination network mask fields (for OpenFlow 1.0 these are implicitly defined by the wildcard field).

$$(a) \rightarrow \text{(b) drop packet} \tag{4.2}$$

For the candidate (evasion) rule set,

$$\begin{aligned} &1 \text{ a} \rightarrow \text{c set (a} \Rightarrow \text{a')} \\ &2 \text{ a'} \rightarrow \text{c set (c} \Rightarrow \text{b)} \\ &3 \text{ a'} \rightarrow \text{b forward packet} \end{aligned} \tag{4.3}$$

the intermediate alias sets are

$$\begin{aligned} &1 \text{ a} \rightarrow \text{c set (a} \Rightarrow \text{a')} \quad \text{(a, a') (c)} \\ &2 \text{ a'} \rightarrow \text{c set (c} \Rightarrow \text{b)} \quad \text{(a, a') (c, b)} \\ &3 \text{ a'} \rightarrow \text{b forward packet (a, a') (c, b) forward packet} \end{aligned} \tag{4.4}$$

and the derived rule is

$$\text{(a, a')} \Rightarrow \text{(c, b) forward packet} \tag{4.5}$$

4.6.1.2 Rule Set Conflict Evaluation

The *FRESCO* SEK first performs alias set rule reduction on the candidate rule set. These validity checks are then performed between each derived *FRESCO* constraint rule *cRule* and each derived flow rule *fRule*, as follows:

1. Skip any cRule/fRule pair with mismatched prototypes.
2. Skip any cRule/fRule pair whose actions are both either forward or drop packet.
3. If cRule's alias sets intersect those of fRule's, declare a conflict.

Thus, given the example security constraint table in Equation 4.2 and the candidate rule set in Equation 4.5, assuming that both rules are TCP protocol, the

first candidate rule passes the first two checks. However, for the third check, because the intersection of the source and destination alias sets results in (a) and (b), respectively, the candidate rule is declared to be in conflict.

As a practical consideration, because OpenFlow rules permit both wildcard field matches and IP address network masks, determining alias set intersection involves more than simple membership equality checks. To accommodate this, we define comparison operators that determine if a field specification is (*i*) more encompassing ("wider"), (*ii*) more specific ("narrower"), (*iii*) equal, or (*iv*) unequal. Thus, an intersection occurs when the pairwise comparisons between all fields of a candidate rule are wider than, equal to, or narrower than that of the corresponding fields of the constraint table rule.

For a formalization of the above, we first define some terms: (i) S_i is the i_{th} entry of security constraints, (ii) F_i is the i_{th} entry of flow rules, (iii) $SC_{i,j}$ is the j_{th} item of the i_{th} entry of the condition part of the security constraint, (iv) SA_i is the i_{th} entry of the action part of the security constraint, (v) $FC_{i,j}$ is the j_{th} item of the i_{th} condition part of a flow rule from non-*FRESCO* applications, and (vi) FA_i is the i_{th} action part of the flow rule. At this time, both $SC_{i,j}$ and $FC_{i,j}$ are sets whose elements are one of the specific value or some ranges and $j \in \{1, 2, ..., 14\}$. Rule contradiction is then formalized using the following notation:

$$\begin{aligned} &\text{if there is any } S_i, \text{ satisfying } SC_{i,j} \cap FC_{i,j} \neq \oslash \text{ and} \\ &SA_i \neq FA_i, \text{ for all } j, \text{then} F_i \text{ is conflicted with } S_i \end{aligned} \tag{4.6}$$

Finally, upon an update to the security constraints table, rule set conflict resolution is performed against all flow rules currently active within the switch. If a conflict is detected in which the switch rule is found to be wider than the *FRESCO* rule, SEK initiates a request to the switch to flush the resident rule.

4.6.2 Extending FRESCO Security Enforcement Kernel with Formal Method

We propose a provably correct and automatic method for verifying that a given *non-bypass* property holds with respect to a set of flow rules committed by an OpenFlow controller. Non-bypassability is a basic security property, which is enforced by most firewalls and switches. This property stipulates that packets or flows satisfying specified conditions must adhere to a predefined action, such as forward or drop. Since flow tables of switches in OpenFlow environments can include a large number of prioritized flow entries, manual verification of the non-bypass property on large flow tables across switches is challenging. Furthermore, given the dynamic nature of flow tables, the heterogeneity of vendor implementation in flow table ordering and management, and complex flow rule constructs such as *set* operations that can alter packet content, even automated security evaluation systems are challenged by OpenFlow. Here, we address this challenge of verifying the compliance of a *flow rule set* against an invariant security policy. We call this kernel *FRESCO* SEK-FM.

4.6.2.1 Non-Bypass Security Property Violations

FRESCO SEK-FM addresses the problem of verifying that the current state of flow rules inserted in a switch's flow table(s) remain consistent with the current network security policy. We decompose the network security policy into a set of assertions, which we refer to as *Non-bypass properties*. Intuitively, a *Non-bypass property* is commonly observable in modern networks as the flow deny and allow rule, which are statically defined to restrict or enable flows throughout and across the network. A *Non-bypass property* specifies whether a certain packet/flow matching a set of conditions should be dropped or forwarded to its destination (we formalize this notion in Section 4.6.2.3).

For the purpose of verifying a non-bypass property across an OF-network, it is

Flow Table	Condition: Field 1 Src IP	Field 2 Src Port	Field 3 Dst IP	Field 4 Dst port	**Action Set**
1	5	[0,19]	6	[0,19]	{ (drop) }
1	5	[0,19]	[7,8]	[0,19]	{ (set $field_1$ 10), (goto 2) }
1	6	[0,19]	[6,8]	[0,19]	{ (forward) }
2	[10,12]	[0,19]	[0,12]	[0,19]	{ (set $field_3$ 6), (forward) }

Table 4.2: Example OpenFlow rule set used to illustrate coverage and modify violations

necessary to verify all flow tables within the OF-network. Table 4.2 is a simple instance of our proposed flow rule set model with no overlaps. For simplicity, we denote IP addresses as non-negative integers and provide a formal definition of our OF flow rule set in Section 4.6.2.3. Each entry of the flow rule set consists of conditions over defined fields and a set of actions. We assume that if a given packet matches all conditions of multiple entries, any set of actions corresponding to the matching entry may be performed.

FRESCO SEK-FM addresses two types of violations of the non-bypass security property that may be present in an OF flow rule set instance. For the first type of violation, we assume that Table 4.2 is evaluated against the following non-bypass property: *every packet that goes from source IP [5,6] to destination IP 6 must be dropped*. However, an OF switch using Table 4.2 will forward any packet that has 6 for both the source and destination IP address because of the third entry in the first flow table. That is, the final action for every packet satisfying the conditions of a given non-bypass property is inconsistent with the action of the property (and thus some packets can bypass the constraints). We call this kind of misconfiguration a **coverage violation**.

The second type of violation arises due to the *set* command in an OF flow table.

In this example, we define another non-bypass property such that *every packet which goes from source IP address 5 to destination IP address 6 must be dropped.* However, an adversary may tunnel the packet through a series of one or more intermediate receivers such that the transmission chain originates from IP address 5 and ends at destination IP address 6, which is a violation of the non-bypass property. For example, when an adversary sends a malicious packet p whose source and destination IP addresses are 5 and 7 respectively, the packet p is changed into p' that goes from source IP 10 to destination IP 6. Then, the packet p' is forwarded to another switch or the host whose IP address is 6 by the first rule of flow table 2. Thus, packet p' which originates from source IP 5 finally arrives at destination IP 6, which is a clear violation of the specified property. We call this type of violation a **modify violation**.

4.6.2.2 SMT Solving in Yices

Yices is a Satisfiability Modulo Theories (SMT) solver, developed at SRI. The core of Yices implements an efficient SAT solver based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [23]. Yices is provided with an input file modeling given first order logic. If a given model is satisfiable, i.e., there exists at least one instance satisfying all model constraints, Yices outputs such a satisfying example. Otherwise, Yices reports the model to be unsatisfiable. We leverage the soundness of Yices and its ability to efficiently find satisfying examples, to verify flow rule sets.

4.6.2.3 Non-bypass Property Representation

A *non-bypass security* property asserts a feature within a given flow rule set. Formally, a *non-bypass security* property is a form of first order logic consisting of universal quantifier, conditions and an action. An action can be *forward* or *drop*. The conditions part of a non-bypass security property is a conjunction of boolean

expressions over flow rule set fields. The maximum number of fields is up to 15 [70]. The condition for each field is encoded with a boolean expression specifying a range of non-negative integers because every field consists of a number of bits whose length varies from 3 to 64.

To assert non-bypass properties within a flow rule set against *coverage* and *modify* violations, *FRESCO* SEK-FM uses two forms of non-bypass security properties, respectively. The formal representation of a non-bypass property denoting that *a flow rule set is free from coverage violations* is as follows:

$$\textit{Non-bypass property}_c = \forall p(\bigwedge_{j=1}^{n} C_j(p) \rightarrow a), a \in \{\textit{forward, drop}\}$$

$$C_j(p) = F_j(p) \in [iLow_j, iHigh_j],\ F_j(p) = j \text{ th field of p}$$

This property denotes that if an initial packet, before modification by an OF-switch, matches the conditions then its final result must be consistent with the action of the property.

The formal representation of a non-bypass property proving that *a flow rule set has no modify violation* is as follows:

$$\textit{Non-bypass property}_m = \forall (p, p')(\bigwedge_{j=1}^{2} C_j(p) \bigwedge_{k=3}^{n} C_k(p') \rightarrow a),$$

$$a \in \{\textit{forward, drop}\}$$

$$(p, p') = \text{ a pair of initial packet } p \text{ and its final packet } p'$$

$$F_1(p) = \text{source IP field of packet p}, F_2(p) = \text{source port field of packet p}$$

This property dictates that if the initial packet p before modification by an OF-switch matches the source IP and port conditions and its final packet p' matches the remaining conditions of the property then, the final result for packet p must be consistent with the action specified by the property.

We assume that the administrator of an OF network has *a priori* knowledge of what non-bypass properties must be enforced within the network. *FRESCO* SEK-FM checks whether the specified non-bypass properties hold for the rule evaluation sequence imposed by the switch.

4.7 Working Examples

We show two case studies by creating real working security applications written in *FRESCO* script.

4.7.1 Implementing Reflector Net

FRESCO's power stems from its ability to use OpenFlow to effectively reprogram the underlying network infrastructure to defend the network against an emerging threat. To illustrate this notion, consider a *FRESCO* application that allows OF network operators to redirect malicious scanners to a third-party remote honeypot. Using *FRESCO* script, we compose two modules that first detect an active malicious scanner, and then reprogram the switch data plane to redirect all the scanner's flow into a remote honeynet. We refer to our composed security service as a threat *reflector net*, which effectively turns the address space of any OpenFlow network that deploys this service into a contact surface for a remote high-interaction honeypot. The incentive for an operator to use such a service is that the forensic evidence collected by the honeypot can be shared back for the purpose of refining the operator's local *FRESCO*-based firewall.

First, we create and configure a simple threshold-based scan detector instance. Since *FRESCO* already provides a "ScanDetector" module, we can instantiate an instance from this module for selecting malicious external targets. For this example, let us assume that our scan analysis is triggered by an external entity producing large numbers of failed TCP connections. Thus, we establish TCP_CONNECTION_FAIL,

which is captured in FRESCO's native DB service, as an input trigger event for our scan detection, which outputs a scan detection event when a threshold number of failed connections is observed.

Our FRESCO script instantiates the scan detection module using four key script variables: (*i*) input, (*ii*) output, (*iii*) parameter, and (*iv*) action. The input for this instance is a source IP address for a flow that causes TCP_CONNECTION_FAIL event. The parameter will determine a threshold value for a scan detection algorithm, and here, we set this value as 5 (i.e., if a source IP generates five failed TCP connections, we regard it as a scan attacker). The output is a source IP address and a scan detection result (noted as *scan_result*), which are delivered to the second instance as input variables. The action variable is not defined here, as the logic required to formulate and insert flow rules to incorporate duplex redirection is modularized into a second flow redirection instance. The *FRESCO* script for our flow redirection instance is shown in Figure 4.5 (left).

We configure a redirector instance to redirect flows from the malicious scanner to a honeynet (or forward benign flows). This function is an instance of *FRESCO's* "ActionHandler" module. This instance uses a PUSH event, which triggers the instance each time "find_scan" is outputted from the scan detection instance Finally, we need to define an action to redirect flows produced by scan attackers. Thus, we set the action variable of this instance as "scan_result == 1 ? REDIRECT : FORWARD", which indicates that if the input variable of scan_result equals 1 (denoting the scanner) this instance redirects all flows related to the source IP address. The FRESCO script for this instance is shown in Figure 4.5 (right).

We test this script in an OpenFlow simulation environment with Mininet [64], which is commonly used to emulate OpenFlow networks, to show its real operation. In this test, we created three hosts (scanner, target host, and honeynet) and an

```
find_scan (1)(2){
   type:ScanDetector
   event:TCP_CONNECTION_FAIL
   input:source_IP
   output:source_IP, scan_result
   parameter:5
/* no actions are defined */
   action: -
}
```

```
do_redirect (2)(0){
   type:ActionHandler
   event:PUSH
   input:source_IP, scan_result
   output: -
   parameter: -
/* if scan_result equals 1, redirect,
   otherwise, forward */
   action: scan_result == 1 ?
                   REDIRECT : FORWARD
}
```

Figure 4.5: *FRESCO* script with two connecting modules used to build a reflector net

OpenFlow enabled switch. All three hosts are connected to the switch and able to initiate flows to each another.

As illustrated in Figure 4.6, the malicious scanner (10.0.0.2) tries to scan the host (10.0.0.4) using Nmap tool [68]. The scan packets are delivered through an OpenFlow switch **(1)**, where the switch then forwards the flow statistics to a *FRESCO* application (i.e., find_scan instance) through a controller. The find_scan instance determines that these packets are scan-related, and it sends the detection result to the do_redirect instance to instantiate flow rules to redirect these packets to our honeynet (10.0.0.3) **(2)**. At this time, the network configuration of the honeypot is different from the original scanned machine (10.0.0.4), which opens network port 445 while the honeypot opens network port 444. Then, the honeypot returns packets to the scanner as if it is the original target **(3)**. Finally, the scanner receives packet responses from the honeypot **(4)**, unaware that all of its flows are now redirected to and from the honeynet.

4.7.2 Cooperating with a Legacy Security Application

FRESCO provides an interface, which receives messages from legacy security

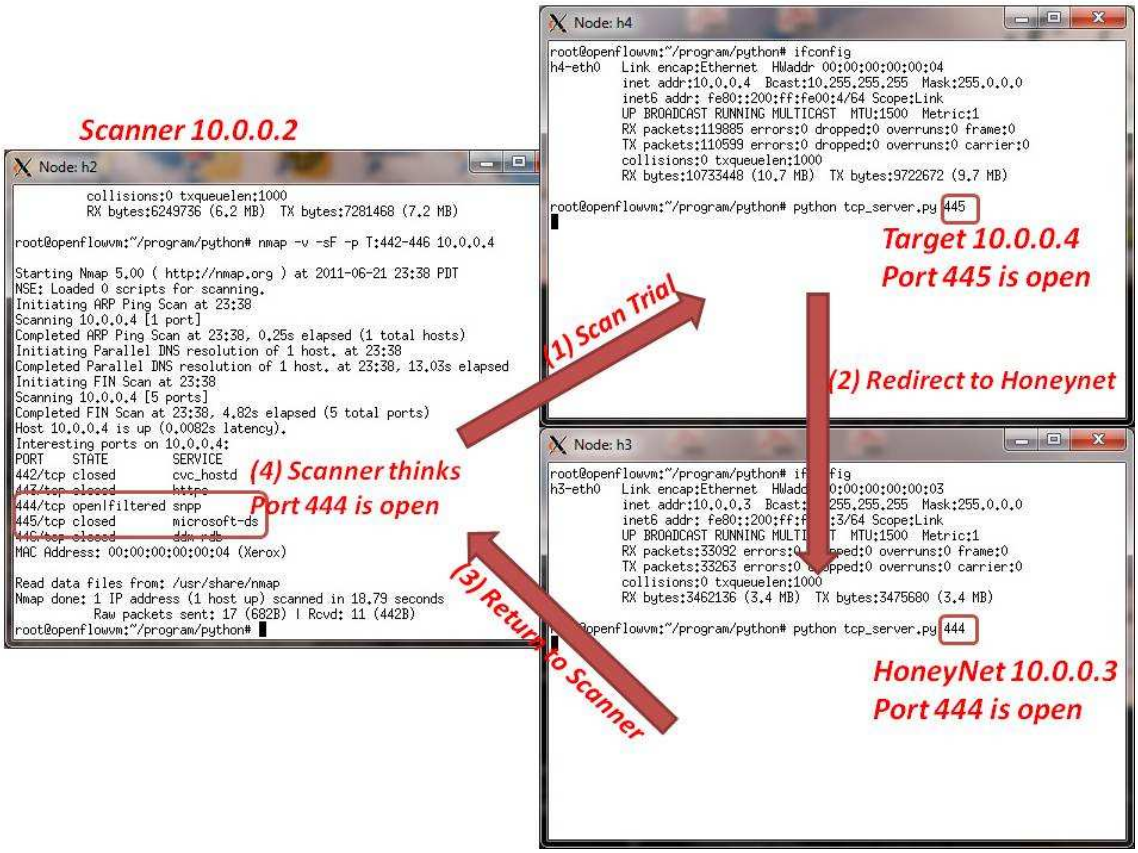


Figure 4.6: Operational illustration of a *FRESCO* reflector net application

applications, such as Snort [89] and BotHunter [34]. Usually, we use these network security applications to monitor our networks, often using DPI-based techniques to identify malicious traffic content or by simply monitoring flow patterns. Using *FRESCO*, alerts produced from such network security monitors can be integrated into the flow rule production logic of OF-enabled networks. To do this, we employ *FRESCO* actions (e.g., drop and redirect) as responses against network attacks.

One might consider reimplementing classic network security applications directly as OpenFlow applications. However, such efforts are both costly in engineering time and subject to limitations in the availability of data provided by the OF controller. Also, OpenFlow does not export full packet content over its APIs, so DPI-based security services must be implemented as external applications. To reduce the integration burden, *FRESCO* provides a function of receiving messages from third-party security applications, and we can simply design response strategies based on the messages through *FRESCO* script.

A message from a third-party security application will be delivered to a module as a type of event - MESSAGE_LEGACY, and the format of a message is of two kinds: (i) *FRESCO* type and (ii) other standardized formats such as the intrusion detection message exchange format (IDMEF) [78]. If we use FRESCO type, it is notified in the event as a keyword of FRESCO, and it can be represented as MESSAGE_LEGACY:FRESCO. If we use IDMEF, it can be shown as MESSAGE_LEGACY:IDMEF.

In the scenario, shown in Figure 4.7, an attacker sends a bot binary (1) to the host C, and BotHunter responds by producing an infection profile (2). Then, BotHunter reports this information (i.e., the Victim IP and forensic confidence score for the infection) to a security application written in *FRESCO* script (3). If the profile's forensic score achieves a threshold value, the application imposes a quarantine action

on the victim IP. The quarantine module uses the *FRESCO* SEK to enforce a series of flow rules that implement the quarantine action SEK (4, 5). Finally, if an infected host (the host C) sends another malicious data to other hosts, such as host A or host B (6), it is automatically blocked by the switch.

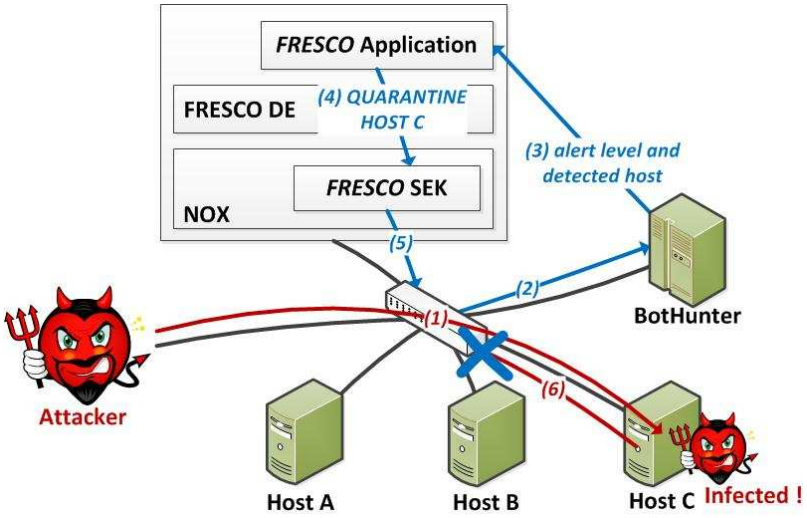


Figure 4.7: Operational illustration of a *FRESCO* actuator cooperating with BotHunter

To implement this function, we simply reconfigure the "do_quarantine" instance (or create another instance) used in the above example scenario for this case. This time, we instantiate the instance with four alternate variables: (*i*) event, (*ii*) input, (*iii*) parameter, and (*iv*) condition part of action. When BotHunter forwards its infection alarm using the *FRESCO* API, we set the event variable as MESSAGE_LEGACY:FRESCO. The input variables passed to this module include the victim_ip (reported as infected by BotHunter), and the infection confidence_score, which represents the degree of forensic evidence recorded within the infection profile. We also specify a parameter for the confidence_threshold, and trigger the QUAR-

ANTINE action when the confidence_score exceeds the confidence_threshold. The *FRESCO* script for this instance is shown in Figure 4.8.

```
do_quarantine (2)(0){
   type:ActionHandler
   event:MESSAGE_LEGACY:FRESCO
   input:victim_ip,confidence_score
   output: -
   parameter:confidence_threshold
/* redirect all flows from source IP */
   action:confidence_score > confidence_threshold
          ? QUARANTINE(victim_ip)
}
```

Figure 4.8: *FRESCO* script for invoking host quarantine for BotHunter

4.8 Implementation

We have developed a prototype implementation of the *FRESCO* architecture. The *FRESCO* Application Layer prototype is implemented in Python and runs as an OpenFlow application on NOX. The prototype operates on NOX version 0.5.0 using the OpenFlow 1.1.0 protocol, and is implemented in approximately 3,000 lines of Python. *FRESCO* modules are implemented as independent Python objects, and inputs and parameters of a module are input variables to the Python object. The return values of a Python object are considered as output values of a module.

A sample implementation of the *FRESCO* Comparator module, used in Figure 4.2 (right), is presented in Figure 4.9. All modules in *FRESCO* start with the function of module_start, and this function has two arguments: (*i*) `input_dic`, which is a dictionary data structure containing F-DB, event, and input values, and (*ii*) `param_list`, which is a list structure storing user-defined parameter values. All variables starting with "FR_" are *FRESCO* native variables. The developer fills in

additional specialized logic at the bottom of the module (lines 13-18).

```
1 def module_start(input_dic, param_list):
2    # initialize FRESCO native inputs
3    FR_FDB = input_dic['FR_FDB']
4    FR_event = in_dic['FR_event']
5    FR_input = input_dic['FR_input']
6
7    # initialize FRESCO variables
8    FR_ret_dic = {}
9    FR_ret_dic['output'] = []
10   FR_ret_dic['action'] = None
11
12   # start - user defined logic
13   if param_list[0] == FR_input[0]:
14      output = 1
15   else:
16      output = 0
17
18   FR_ret_dic['output'].append(output)
19   # end - user defined logic
20
21   return FR_ret_dic
```

Figure 4.9: *FRESCO* comparator module

The *FRESCO* SEK is implemented as a native C++ extension of the NOX source code in approximately 1160 lines of C++ code. We modified the `send_openflow_command` function, whose main operation is to send OpenFlow commands to network switches, to capture flow rules from all OpenFlow applications. *FRESCO* SEK intercepts flow rules in the function and stores them into the security constraints table if the rules are from *FRESCO* applications (i.e., flow rules produced through the *FRESCO* path are considered trusted flow rules and are preserved as active network security constraints). If a flow rule is from a non-*FRESCO* application, *FRESCO* SEK evaluates the rule to determine if a conflict exists within its security constraints table. The match algorithm is specifically optimized to perform the least-expensive comparisons

first. If there are conflicts, an error message is returned to the OF application. Otherwise, the rule is forwarded to the network switches. We implement and evaluate the security constraint table using the in-memory database opportunistic best-fit comparison algorithm, which reports an ability to execute queries in near-constant lookup time.

4.9 System Evaluation

We now evaluate the *FRESCO* framework with respect to its ease of use, flexibility, and security constraints preservation. To evaluate components in *FRESCO*, we employ *mininet* [64], which provides a rapid prototyping environment for the emulation of OpenFlow network switches. Using mininet, we have emulated one OpenFlow network switch, three hosts connected to the switch, and one host to operate our NOX controller. We perform flow generation by selecting one or two hosts to initiate TCP or UDP connections. The remaining host is employed as a medium interaction server, which responds to client application setup requests. We hosted our evaluation environment on an Intel i3 CPU with 4 GB of memory. In addition, we conduct live performance evaluations of the *FRESCO* SEK using an HP ProCurve 6600 OF-enabled switch in a test network laboratory.

4.9.1 Evaluating Modularity and Composability

For the evaluation, we begin with the basic problem of identifying entities performing flow patterns indicative of malicious network scanning, and compare schemes of implementing network scanning attacks with and without the use of *FRESCO*.

While network scanning is a well-studied problem in the network security realm, it offers an opportunity to examine the efficiency of entity tracking using *FRESCO*. Many well-established algorithms for scan detection exist [48, 47, 82]. However, under OpenFlow, the potential for *FRESCO* to dynamically manipulate the switch's

data path in reaction to malicious scans is a natural objective. This scenario also lets us examine how simple modules can be *composed* to perform data collection, evaluation, and response.

1. *FRESCO* Scan Deflector Service. Figure 4.10 illustrates how *FRESCO* modules and their connections can be linked together to implement a *malicious scan deflector* for OpenFlow environments. This scan detection function consists of the three modules described above. First, we have a module for looking up a blacklist. This module checks a blacklist table to learn whether or not an input source IP is listed. If the table contains the source IP, the module notifies its presence to the second module. Based on the input value, the second module performs threshold-based scan detection or it drops a packet. If it does not drop the packet, it notifies the detection result to the third module. In addition, this second module receives a parameter value that will be used to determine the threshold. Finally, the third module performs two actions based on input. If the input is 1, the module redirects a packet. If the input is 0, it forwards a packet. Implementing the three modules required 205 lines of Python code and 24 lines of *FRESCO* script (this script is shown in Figure 4.11).

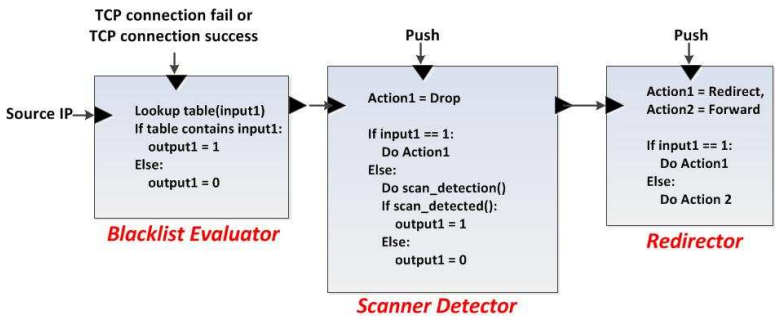


Figure 4.10: *FRESCO* composition of a scan deflector

```
1 blacklist_check (1)(1){
2   type:TableLookup
3   event:TCP_CONNECTION_FAIL,
          TCP_CONNECTION_SUCCESS
4   input:source_IP
5   output:blacklist_out
6   parameter:-
7   action:-
8 }
```

```
1 find_scan (1)(1){
2   type:ScanDetector
3   event:PUSH
4   input:blacklist_out
5   output:scan_out
6   parameter:-
7   action:blacklist_out == 1
          ?DROP
8 }
```

```
1 do_action (1)(0){
2   type:ActionHandler
3   event:PUSH
4   input:scan_out
5   output:-
6   parameter:-
7   action:scan_out == 1
          ?REDIRECT:FORWARD
8 }
```

Figure 4.11: *FRESCO* script for a scan detector

2. *FRESCO* BotMiner Service. To illustrate a more complex flow analysis module using *FRESCO*, we have implemented a *FRESCO* version of the *BotMiner* [33] application. Note that our goal here is not faithful, "bug-compatible" adherence to the full BotMiner protocol described in [33], but rather to demonstrate feasibility and to capture the essence of its implementation through *FRESCO*, in a manner that is slightly simplified for readability.

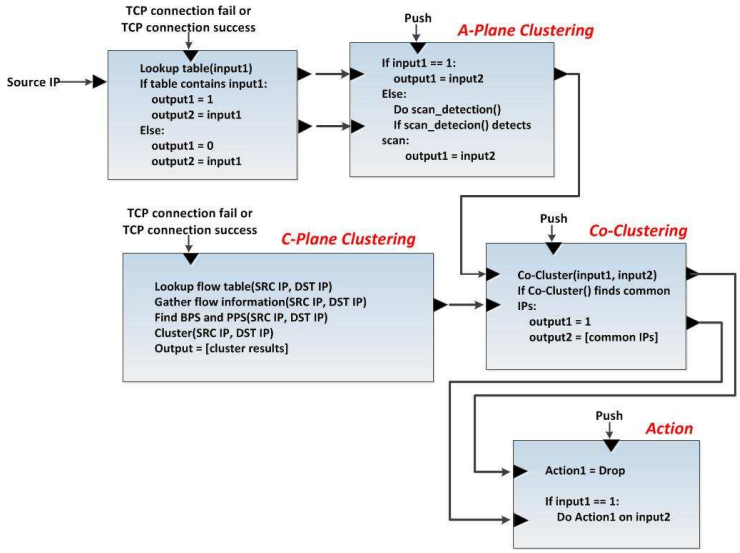


Figure 4.12: *FRESCO* composition of the BotMiner service

BotMiner detects bots through network-level flow analysis. We have implemented the essentials of its detection functionality using five modules as shown in Figure 4.12. BotMiner assumes that hosts infected with the same botnet exhibit similar patterns at the network level, and these patterns are different from benign hosts. To find similar patterns between bots, BotMiner clusters botnet activity in two dimensions (C-plane and A-plane). The C-plane clustering approach is used to detect hosts that resemble each other in terms of (packets per second) and bps (bytes per second). The A-plane clustering identifies hosts that produce similar network anomalies. In this implementation, we use the scan detector module to find network anomalies. Finally, if we find two clusters, we perform co-clustering to find common hosts that exist in both dimensions and label them as bots. BotMiner was implemented in 312 lines of python code and 40 lines of *FRESCO* script (the script for BotMiner is presented in Figure 4.13).

```
1 table_check (1)(2){
2   type:TableLookup
3   event:TCP_CONNECTION_FAIL,
          TCP_CONNECTION_SUCCESS
4   input:source_IP
5   output:table_out,source_IP
6   parameter:-
7   action:-
8 }
```

```
1 a_cluster (2)(1){
2   type:A-PlaneCluster
3   event:PUSH
4   input:table_out,source_IP
5   output:a_cls_out
6   parameter:-
7   action:-
8 }
-
```

```
1 c_cluster (0)(1){
2   type:C-PlaneCluster
3   event:TCP_CONNECTION_FAIL,
          TCP_CONNECTION_SUCCESS
4   input:-
5   output:c_cls_out
6   parameter:-
7   action:-
8 }
```

```
1 cr_cluster (2)(2){
2   type:CrossCluster
3   event:PUSH
4   input:a_cls_out,c_cls_out
5   output:cross_out,ip_list
6   parameter:-
7   action:-
8 }
-
```

```
1 do_action (2)(0){
2   type:ActionHandler
3   event:PUSH
4   input:cross_out,ip_list
5   output:-
6   parameter:-
7   action:cross_out == 1
          ?DROP(ip_list):FORWARD
8 }
```

Figure 4.13: *FRESCO* scripts illustrating composition of the BotMiner service

3. *FRESCO* P2P Plotter Service. We have implemented a *FRESCO*-based P2P malware detection service, similarly implemented to capture the concept of

the algorithm, but simplified for the purpose of readability. Motivated by Yen's work [109], we have implemented the P2P malware detection algorithm, referred to as *P2P Plotter*, using *FRESCO*. The P2P Plotter asserts that P2P malware has two interesting characteristics, which are quite different from normal P2P client programs. First, P2P malware usually operates at lower volumes of network flow interactions than what is typically observed in benign P2P protocols. Second, P2P malware typically interacts with a peer population that has a lower churn rate (i.e., the connection duration time of P2P plotters is longer than that of normal P2P clients). The algorithm operates by performing co-clustering, to find common hosts that exhibit both characteristics (i.e., low volume and low churn rate).

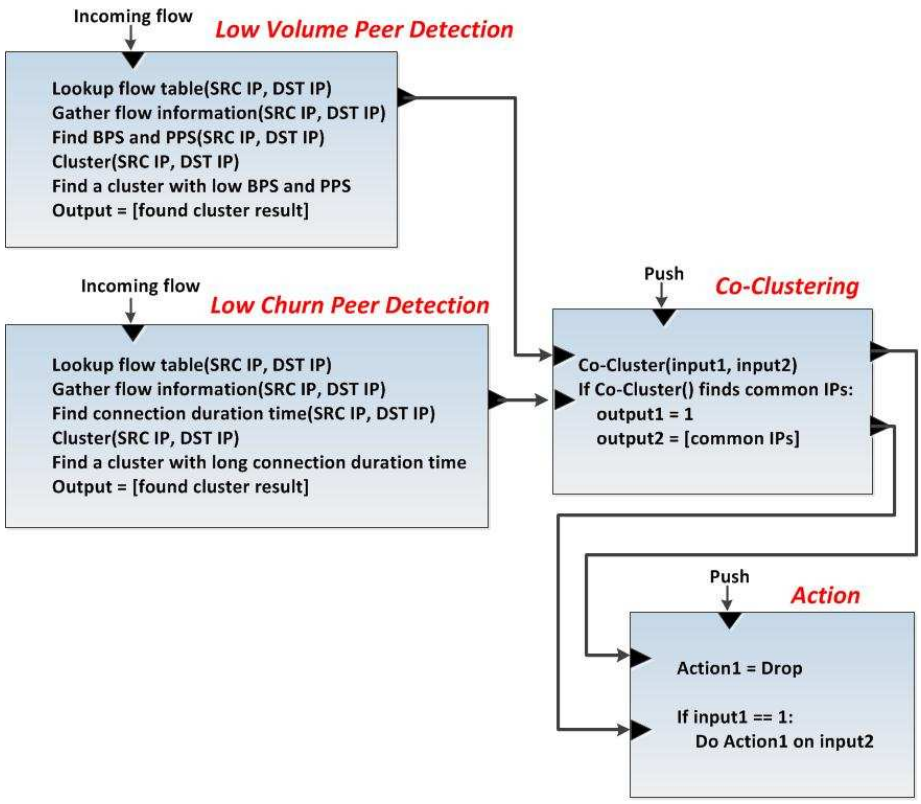


Figure 4.14: *FRESCO* composition of the P2P plotter

We have implemented this essential functionality of the P2P Plotter algorithm as a 4-module *FRESCO* script, which is shown in Figure 4.14. This involved 227 lines of Python code and 32 lines of *FRESCO* script. The script for the P2P Plotter is illustrated in Figure 4.15. The reuse of modules (i.e., `CrossCluster` and `ActionHandler`, from the BotMiner service implementation is noteworthy, highlighting the reuse potential of *FRESCO* modules.

```
1 low_volume_peer (0)(1){
2   type:VolumeDetector
3   event:INCOMING_FLOW
4   input:-
5   output:volume_out
6   parameter:-
7   action:-
8 }
```

```
1 low_churn_peer (0)(1){
2   type:ChurnDetector
3   event:INCOMING_FLOW
4   input:-
5   output:churn_out
6   parameter:-
7   action:-
8 }
```

```
1 cr_cluster (2)(2){
2   type:CrossCluster
3   event:PUSH
4   input:volume_out,churn_out
5   output:cross_out,ip_list
6   parameter:-
7   action:-
8 }
```

```
1 do_action (2)(0){
2   type:ActionHandler
3   event:PUSH
4   input:cross_out,ip_list
5   output:-
6   parameter:-
7   action:cross_out == 1 ? DROP(ip_list):FORWARD
8 }
```

Figure 4.15: *FRESCO* scripts illustrating composition of the P2P plotter

4.9.2 Comparing FRESCO Applications with Non-FRESCO Detectors

Network anomaly detection approaches, e.g., TRW [48], have been well-studied and are commonly used as a complement to signature-based detection systems in traditional networks. While these approaches may be instantiated as software programs or in hardware devices, the common practice is to implement them as stand-alone software programs. (We envision that the *FRESCO* development environment may be similarly used for rapid prototyping and evaluation of certain anomaly detection algorithms in OpenFlow networks.)

To highlight the advantages of *FRESCO*, we first choose an open-source network anomaly detection system and then replicate identical functionality using *FRESCO*. Specifically, we compare *FRESCO* with a recently published work [63], where the authors implemented popular network anomaly detection algorithms such as TRW-CB [79] and Rate Limit [98] as applications running on an OpenFlow network controller. We re-implement the same algorithms (i.e., TRW-CB and Rate Limit) using existing *FRESCO* modules and the *FRESCO* scripting language. We provide a comparison in Table 4.3, in terms of the number of lines of source code, to demonstrate the utility of the *FRESCO* development environment.

As summarized in Table 4.3, prior work [63] makes the case that its OpenFlow application implementation is slightly simpler than the standard implementation (i.e., the source code for the OpenFlow implementation is roughly 70% to 80% the length of the standard implementation). Using *FRESCO*, we are able to realize similar functionality with an order of magnitude fewer lines of code. That is, we have implemented the identical TRW-CB function with 66 lines of code (58 lines of Python and 8 lines of *FRESCO* script) and the rate limiting function with 69 lines of code (61 lines of Python and 8 lines of *FRESCO* script). These two examples represent 6% to 7% of the length of their standard implementations, and less than 9% of the recently published OpenFlow implementation.

	Implementation		
Algorithms	Standard	OpenFlow application	*FRESCO*
TRW-CB	1,060	741	66 (58 + 8)
Rate Limit	991	814	69 (61 + 8)

Table 4.3: Source code length for standard, OpenFlow and *FRESCO* implementations of the TRW-CB and Rate-Limit anomaly detection algorithms

4.9.3 Measuring and Evaluating FRESCO Overhead

***FRESCO* Application Layer Overhead.** We compare the flow setup time of NOX flow generation with five other *FRESCO* applications and summarize the results in Table 4.4. To measure this, we capture packets between NOX and the OpenFlow switch, and measure the round trip required to submit the flow and receive a corresponding flow constraint. We observe that *FRESCO* applications require additional setup time in the range of 0.5 milliseconds to 10.9 milliseconds.[‡]

	NOX	Simple Flow Tracker	Simple Scan Detector	Threshold Scan Detector	BotMiner Detector	P2P Plotter
Time (ms)	0.823	1.374	2.461	7.196	15.421	11.775

Table 4.4: Flow setup time comparison of NOX with five *FRESCO* applications

Resource Controller Overhead. The resource controller component monitors switch status frequently and removes old flow rules to reclaim space for new flow rules, which will be enforced by *FRESCO* applications. This job is performed by *FRESCO*'s garbage collector, a subcomponent of the resource controller, which we test under the following scenario. First, we let non-*FRESCO* applications enforce 4,000 flow rules to an OpenFlow network switch. In this case, we assume that the maximum size of the flow table in the switch is 4,000, and we set the threshold value(θ) for garbage collection as 0.75 (i.e., if the capacity of a flow table in a switch is $\leq$ 75%, we run the garbage collector). Our test results, shown in Figure 4.16, demonstrate that the garbage collector correctly implements its flow eviction policy.

[‡]These setup times were measured on mininet, which is an emulated environment running on a virtual machine. If we use a more powerful host for the controller, which is the common case in an OpenFlow network, this setup time will be reduced significantly.

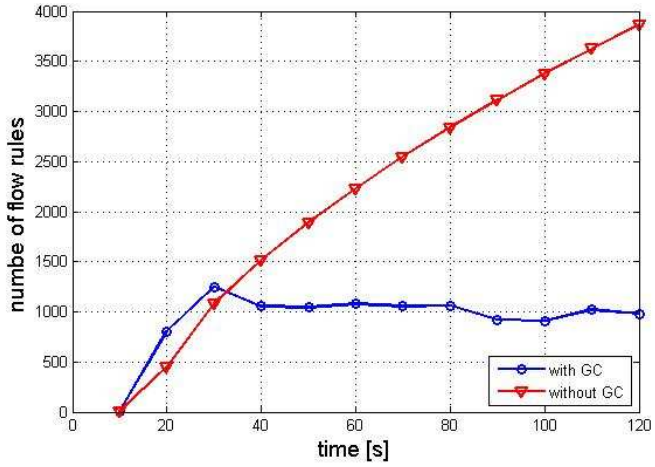


Figure 4.16: Operation of *FRESCO* garbage collector

4.10 Related Work

The OpenFlow standard has as its roots on a rich body of work on control-flow separation and clean-slate design of the Internet (e.g., [17], [31]). SANE [18] and Ethane [17] propose new architectures for securing enterprise networks. The SANE [18] protection layer proposes a fork-lift (clean-slate) approach for upgrading enterprise network security that introduces a centralized server, i.e., domain controller, to authenticate all elements in the network and grant access to services in the form of capabilities that are enforced at each switch. Ethane [17] is a more practical and backwards-compatible instantiation of SANE that requires no modification to end hosts. Both studies may be considered as catalysts for the emergence of OpenFlow and software-defined networking.

FRESCO is built over the foundations laid by these studies and shares a common objective with these systems in that it seeks to improve enterprise security using programmable network elements. However, *FRESCO* emphasizes composable security, and applications that it enables are much more sophisticated than simple access

control policies. In addition, the *FRESCO* SEK focuses on providing continued enforcement of potentially conflicting flow constraints imposed by FRESCO and other OF applications. Thus, we consider our work as greatly complementary to existing studies such as SANE and Ethane.

FRESCO's focus is on the development of a holistic platform for specifying and developing OF security applications and enforcement of security constraints generated by these applications. Prior work has addressed a part of this problem, i.e., development of new languages for specifying security policies. Nettle [100] is a new language for managing OF switches that is based on functional reactive programming. Frenetic [27] and Procera [101] provide declarative query language frameworks for managing distributed OF switches, describing high-level packet-forwarding and specifying network policies. The OpenSAFE system provides a language framework for enabling redirection of traffic to network monitoring devices [7]. In contrast to these languages, the *FRESCO* development environment is specialized to serve the needs of security applications. Specifically, *FRESCO* applications issue high-level security directives (e.g., REDIRECT, QUARANTINE, MIRROR), which are then translated into OF-specific commands by the script-to-module translator. In addition, *FRESCO* applications require aggregate session and flow state information as well as directives for asynchronous delivery of switch state information that is unavailable in standard OF environments. Applications such as Random Host Mutation [44] are additional motivating examples of candidate OF security applications whose development may be accelerated using *FRESCO*.

The *FRESCO* security enforcement kernel is informed by prior research focused on testing or verifying firewall and network device configuration [83, 25, 58, 59, 4, 105, 3], e.g., using Firewall Decision Diagrams (FDDs) [58] or test case generators [83, 25]. These studies do not deal with dynamic networks. More recently, *header space*

analysis was proposed, which is a generic framework to express various network misconfigurations and policy violations [50]. While HSA can in theory deal with dynamic networks, the *FRESCO* SEK differs in that it is specialized to deal with specific policy violations by OF applications, rule conflict detection, and dynamic flow tunneling. Veriflow proposes to slice the OF network into equivalence classes to efficiently check for invariant property violations [51]. The alias set rule reduction algorithm used by *FRESCO* SEK is complementary to this approach.

We build our system on NOX, which is an open-source OF controller [36]. However, our methodology could be extended to other architectures like Beacon [71], Maestro [13], and DevoFlow [65]. FlowVisor is a platform-independent OF controller that uses network slicing to separate logical network planes, allowing multiple researchers to run experiments safely and independently in the same production OpenFlow network [85]. Our work differs from FlowVisor in several ways. First, FlowVisor cares primarily about non-interference *across* different logical planes (slices) but does not instantiate network security constraints *within* a slice. It is possible that an OF application uses packet modification functions resulting in flow rules that are applied across multiple network switches within the same slice. In such cases, we need a security enforcement kernel to resolve conflicts as described in Section 4.6. Second, although FlowVisor improves security by separating the OF network into logical planes, it does not provide analogous capabilities to *FRESCO* for building additional security applications.

The need for better policy validation and enforcement mechanisms has been touched on by prior and concurrent research efforts. NICE provides a model-checking framework that uses symbolic execution for automating the testing of OpenFlow applications [16]. The Resonance architecture enables dynamic access control and monitoring in SDN environments [67]. The FlowChecker system encodes OpenFlow flow

tables into Binary Decision Diagrams (BDD) and uses model checking [3] to verify security properties. However, the evaluation of FlowChecker does not consider handling of *set* action commands, which we consider to be a significant distinguisher for OpenFlow networks. More recently, researchers have proposed developing language abstractions to guarantee consistency of flow updates in software-defined networks [77]. In contrast, our complementary work on the *FRESCO* security enforcement kernel is focused on detection of rule update conflicts and security policy violations. The Onix platform [54] provides a generalized API for managing a distributed control plane in Software Defined Networks. The techniques and the strategies developed in Onix for managing a distributed network information base are complementary and can be integrated into *FRESCO*.

4.11 Summary of this Chapter

Despite the recent success of SDN, developing and deploying complex SDN security services remains a significant challenge. We present *FRESCO*, a new application development framework specifically designed to address this problem. We introduce the *FRESCO* architecture and its integration with the NOX OpenFlow controller, and present several illustrative security applications written in the *FRESCO* scripting language. To empower *FRESCO* applications with the ability to produce enforceable flow constraints that can defend the network as threats are detected, we present the *FRESCO* security enforcement kernel. Our evaluations demonstrate that *FRESCO* introduces minimal overhead and that it enables rapid creation of popular security functions with significantly (over 90%) fewer lines of code. We believe that *FRESCO* offers a powerful new framework for prototyping and delivering innovative security applications into the rapidly evolving world of software-defined networks. We plan to release all developed code as open source software to the SDN community.

5. CONCLUSION AND FUTURE WORK

In this thesis, we present several techniques to protect our networks. First, we analyze the infection trends of recent bot malware samples, and we provide several new findings and insights based on the analysis results. For example, in this research, we first propose a way of comparing multiple different botnet malware, and it discovers many new things that have not shown before. In addition, we suggest an way of predicting malware infection just based on known infection trends. This approach is pretty simple, but it is promising in estimating which network is infected by bot malware. We believe that our findings, insights, and the prediction method help security researchers devise more intelligent malware defending systems.

Second, we design a bot malware detection system, and it can effectively detect bot malware without adding serious overhead to our systems. This system first sifts out benign processes, and it just focuses on some suspicious processes. After sifting out, it uses several techniques to investigate the suspicious processes more to know whether they are malicious or not. In our evaluation, we find that this system can detect 15 well-known bot malware clearly, and it has very low false-positive rates. Moreover, our test indicates that our system causes very little overhead (less than 2%) to each host, and it addresses the problem of existing host-based bot malware detection systems.

Above two techniques are useful to defend our network from bot malware. However, there are much more diverse network threats, and we want to defend our networks from these threats. To achieve this, we try to design a new framework to make our networks more secure. Since it is hard to change the architecture of existing networks, we have decided to use a new technology that can be used for a future

network. We employ the Software Defined Networking (SDN) technology that is considered as a key technology for future networks, and we realize our ideas with this technology. In this context, we create a new security frame work for, and it is FRESCO. With FRESCO, security researchers and network administrators can easily create intelligent security applications, and they can make their networks more secure.

We will continue our research to make our networks more secure and to protect them from advanced network threats. We will analyze more recent and diverse bot malware samples and their infection trends to investigate their characteristics. Since malware writers will keep developing new malware to attack our networks, we also need to keep monitoring our networks to find new malware. Finding new malware samples in the early stage is really important, because it will reduce the effect of malware infection.

We will extend the EFFORT system to make this system more efficient and effective. We will consider more network and host level features to make detection more effective. For example, EFFORT system currently investigates around 20 systems calls to check whether a suspicious process conducts malicious operations or not. We will add more systems calls (but not too much) for investigation, and they will help EFFORT more malicious operations.

We will build more intelligent security applications for FRESCO. For example, we will create some network anomaly detection applications running on FRESCO. In addition, we will improve the performance of FRESCO, and it will promote the usage of FRESCO applications in real networks.

REFERENCES

[1] Google Safe Browsing API. http://code.google.com/apis/safebrowsing/.

[2] AdSide. Adside, beyond standard ads. http://www.adside.com/home.html.

[3] E. Al-Shaer and S. Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of ACM workshop on Assurable and Usable Security Configuration (SafeConfig)*, pages 37–44, New York, NY, USA, 2010.

[4] E. Al-shaer, W. Marrero, A. El-atawy, and K. Elbadawi. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, pages 123–132, Princeton, NJ, USA, 2009.

[5] Heather Alderfer, Stephen Flynn, Bryan Birchmeier, and Emilie Schulz. Information Policy Country Report: Turkey. In *University of Michigan School of Information Report*, Ann Arbor, MI, USA, 2009. http://open.umich.edu/education/si/si507/fall2009/information-policy-country-report-turkey.

[6] Alexa. Alexa, web information company. http://www.alexa.com/.

[7] Jeffrey R. Ballard, Ian Rae, and Aditya Akella. Extensible and scalable network monitoring using opensafe. In *Proceedings of USENIX Internet Network Management workshop*, pages 8–18, San Jose, CA, USA, 2010.

[8] BOTLAB. A Study in Spam. http://botlab.org/.

[9] R. S. Braga, E Mota, and A Passito. Lightweight ddos flooding attack detection using nox/openflow. In *Proceedings of the 35th Annual IEEE Conference on Local Computer Networks (LCN)*, pages 408–415, Denver, CO, USA, 2010.

[10] Christopher J.C. Burges. A Tutorial on Support Vector Machines for Pattern Recognition. In *Journals of the Data Mining and Knowledge Discovery*, volume 2, pages 121–167, 1998.

[11] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of ACM Computer and Communications Security (CCS)*, pages 621–634, Chicago, IL, USA, 2009.

[12] Xue Cai and John Heidenmann. Understanding Address Usage in the Visible Internet. In *USC/ISI Technical Report ISI-TR-656*, LA, CA, USA, 2009.

[13] Zheng Cai, Alan L. Cox, and T.S. Eugene Ng. Maestro: A system for scalable openflow control. In *Rice University Technical Report*, Houston, TX, USA, 2010.

[14] CAIDA. Conficker/Conflicker/Downadup as seen from the UCSD Network Telescope. `http://www.caida.org/research/security/ms08-067/conficker.xml`.

[15] Marco Canini, Daniele Venzano, Peter Pereŝíni, Dejan Kostić, and Jennifer Rexford. A nice way to test openflow applications. In *Usenix Symposium on Networked Systems Design and Implementation*, pages 10–10, San Jose, CA, USA, April 2012.

[16] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, and Jennifer Rexford. A nice way to test openflow applications. In *Proceedings of USENIX*

Symposium on Networked systems design and implementation (NSDI), San Jose, CA, USA, 2012.

[17] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *Proceedings of ACM SIGCOMM*, pages 1–12, Kyoto, Japan, 2007.

[18] Martin Casado, Tal Garfinkel, Michael Freedman, Aditya Akella, Dan Boneh, Nick McKeowon, and Scott Shenker. Sane: A protection architecture for enterprise networks. In *Proceedings Usenix Symposium on Security*, Vancouver, B.C., Canada, 2006.

[19] Eric Chien. Downadup: Attempts at Smart Network Scanning. `http://www.symantec.com/connect/blogs/downadup-attempts-smart-network-scanning`.

[20] Chia Yuan Cho, Juan Caballeroy, Chris Grier, Vern Paxsonz, and Dawn Song. Insights from the Inside: A View of Botnet Management from Infiltration. In *Proceedings of the USENIX workshop on Large-Scale Exploits and Emergent Threats (LEET)*, San Jose, CA, USA, 2010.

[21] Corinna Cortes and V. Vapnik. Support-Vector Networks. In *Journals of the Machine Learning*, pages 273–297, 1995.

[22] Detours. Software packaged for detouring win32 and application apis. `http://research.microsoft.com/en-us/projects/detours/`.

[23] B. Dutetre and L. Moura. The YICES SMT solver. Technical report, SRI, Menlo Park, CA, USA, 2006.

[24] EasyHook. Easyhook - the reinvention of windows api hooking. `http://easyhook.codeplex.com/`.

[25] A. El-atawy, T. Samak, Z. Wali, E. Al-shaer, F. Lin, C. Pham, and S. Li. An automated framework for validating firewall policy enforcement. Technical report, Washington, DC, USA, 2007.

[26] FAQs.org. FAQ: How do spammers get people's email addresses ? `http://www.faqs.org/faqs/net-abuse-faq/harvest/`.

[27] Nate Foster, Rob Harrison, Michael Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM SIGPLAN Inernational Conference on Functional Programming*, pages 279–291, Tokyo, Japan, 2011.

[28] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of Annual Network and Distributed System Security Symposium (NDSS)*, pages 1–10, Chicago, IL, USA, February 2003.

[29] ghacks.net. Googleupdate.exe. `http://www.ghacks.net/2008/12/28/googleupdateexe/`.

[30] Google. Google web search engine. `http://www.google.com`.

[31] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Goeffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4d approach to network control and management. In *Proceedings of ACM Computer Communications Review*, volume 35, pages 41–54, 2005.

[32] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *Proceedings of USENIX Symposium on Security*, pages 139–154, San Jose, CA, USA, July 2008.

[33] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *Proceedings of USENIX Symposium on Security*, pages 139–154, 2008.

[34] Guofei Gu, Phillip Porras, Vinod Yegneswaran, Martin Fong, and Wenke Lee. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *Proceedings of USENIX Symposium on Security*, pages 121–126, Boston, MA, USA, August 2007.

[35] Guofei Gu, Junjie Zhang, and Wenke Lee. BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In *Proceedings of Network and Distributed System Security Symposium (NDSS'08)*, Sandiego, CA, USA, February 2008.

[36] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martn Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. In *Journals of ACM SIGCOMM Computer Communication Review*, volume 38, pages 105–110, July 2008.

[37] Ramakrishna Gummadi, Hari Balakrishnan, Petros Maniatis, and Sylvia Ratnasamy. Not-a-Bot (NAB): Improving Service Availability in the Face of Botnet Attacks. In *Proceedings of Symposium on Networked System Design and Implementation (NSDI)*, pages 307–320, Boston, MA, USA, April 2009.

[38] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 17–17, San Jose, California, USA, 2010.

[39] Thorsten Holz, Christian Gorecki, and Felix Freiling. Detection and Mitigation of Fast-Flux Service Networks. In *Proceedings of Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2008.

[40] IANA. IANA IPv4 Address Space Registry. `http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml`.

[41] Nicholas Ianelli and Aaron Hackworth. Botnets as a Vehicle for Online Crime. 2005.

[42] Nicholas Ianelli and Aaron Hackworth. Botnets as a Vehicle for Online Crime. In *Proceedings of Annual FIRST Conference*, Baltimore, MD, USA, 2006.

[43] IP2Location. IP2Location Internet IP Address 2009 Report. `http://www.ip2location.com/`.

[44] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: Transparent moving target defense using software-defined networking. In *Proceedings of ACM SIGCOMM HotSDN workshop*, pages 127–132, Helsinki, Finland, 2012.

[45] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of ACM conference on Computer and communications security (CCS)*, pages 121–128, Chicago, IL, USA, October 2007.

[46] John P. John, Alexander Moshchuk, Steven D. Gribble, and Arvind Krishnamurthy. Studying Spamming Botnets Using Botlab. In *Proceedings of USENIX Symposium on Networked systems design and implementation (NSDI)*, pages 291–306, Boston, MA, USA, 2009.

[47] J. Jung, R. Milito, and V. Paxson. On the Adaptive Real-time Detection of Fast Propagating Network Worms. In *Proceedings of Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, pages 175–192, Lucerne, Switzerland, 2007.

[48] Jaeyeon Jung, Vern Paxson, Arthur Berger, and Hari Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proceedings IEEE Symposium on Security and Privacy*, pages 211–225, Oakland, CA, USA, 2004.

[49] Anestis Karasaridis, Brian Rexroad, and David Hoeflin. Wide-scale botnet detection and characterization. In *Proceedings of the first conference on First workshop on Hot Topics in Understanding Botnets*, Cambridge, MA, USA, April 2007.

[50] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Prceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 9–9, San Jose, CA, USA, 2012.

[51] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of ACM SIGCOMM HotSDN workshop*, pages 15–28, Lombard, IL, USA, 2012.

[52] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, August 2000.

[53] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and Xiaofeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of USENIX Symposium on Security*, pages 351–366, Montreal, Canada, 2009.

[54] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–6, Vancouver, BC, Canada, 2010.

[55] Srinivasan Krishnan and Yongdae Kim. Passive identification of Conficker nodes on the Internet. In *University of Minnesota - Technical Document*, Minneapolis, MN, USA, 2009.

[56] Taku Kudo. Tinysvm: Support vector machines. `http://chasen.org/~taku/software/TinySVM/`.

[57] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. AccessMiner: using system-centric models for malware protection. In *Proceedings of ACM conference on Computer and Communications Security (CCS)*, pages 399–412, Chicago, IL, USA, 2010.

[58] A. Liu. Formal verification of firewall policies. In *Proceedings of IEEE International Conference on Communications (ICC)*, pages 1494–1498, Beijing,China, May 2008.

[59] A. Liu and M. Gouda. Diverse Firewall Design. *Journals of IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 19(8), 2008.

[60] MalwareDomains. Dns-bh malware domain blacklists. `http://www.malwaredomains.com/wordpress/?p=1411`.

[61] McAfee. Srizbi Infection. `http://www.mcafee.com/threat-intelligence/malware/default.aspx?id=142902`.

[62] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. In *Journals of ACM SIGCOMM Computer Communication Review*, volume 38, pages 69–74, April 2008.

[63] Syed Akbar Mehdi, Junaid Khalid, and Syed Ali Khayam. Revisiting traffic anomaly detection using software defined networking. In *Proceedings of Recent Advances in Intrusion Detection*, pages 161–180, Menlo Park, CA, USA, 2011.

[64] Mininet. Rapid Prototyping for Software Defined Networks. `http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet/`.

[65] Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, Andrew R. Curtis, and Sujata Banerjee. Devoflow: Cost-effective flow management for high performance enterprise networks. In *Proceedings of the Ninth ACM SIGCOMM workshop on Hot Topics in Networks*, pages 1:1–1:6, Monterey, California, USA, 2010.

[66] MicroSoft MSDN. Windows hook functions. `http://msdn.microsoft.com/en-us/library/ff468842(v=VS.85).aspx`.

[67] Ankur Nayak, Alex Reimers, Nick Feamster, and Russ Clark. Resonance: Dynamic access control for enterprise networks. In *Proceedings of Workshop on Research Enterprise Network*, pages 11–18, Barcelona, Spain, 2009.

[68] NMAP.ORG. Nmap: Open Source Network Discovery and Auditing Tool. `http://nmap.org`.

[69] OpenFlow. OpenFlow 1.1.0 Specification. `http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf`.

[70] OpenFlow. OpenFlow Swtch Specification version 1.1.0. Technical report, 2011. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[71] OpenFlowHub. BEACON. http://www.openflowhub.org/display/Beacon.

[72] Darren Pauli. Srizbi botnet sets new records for spam. PC World. Retrieved 2008-07-20.

[73] Lucian Popa, Minlan Yu, Steven Y. Ko, Ion Stoica, and Sylvia Ratnasamy. Cloudpolice: Taking access control out of the network. In *Proceedings of the 9th ACM workshop on Hot Topics in Networks*, HotNets, pages 7:1–7:6, Monterey, CA, USA, 2010.

[74] Phillip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A security enforcement kernel for openflow networks. In *Proceedings ACM SIGCOMM HotSDN workshops*, pages 121–126, Helsinki, Finland, 2012.

[75] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. My botnet is bigger than yours (maybe, better than yours): why size estimates remain challenging. In *Proceedings of the first conference on First workshop on Hot Topics in Understanding Botnets*, Cambridge, MA, 2007.

[76] Uri Raz. How do spammers harvest email addresses ? http://www.private.org.il/harvest.html.

[77] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent update for software-defined networks: Change you can believe in! In *Under Submission*, pages 7:1–7:6, Cambridge, MA, USA, 2011.

[78] RFC4765. The intrusion detection message exchange format (idmef). http://www.ietf.org/rfc/rfc4765.txt.

[79] S.E. Schechter, J. Jung, and A.W. Berger. Accuracy improving guidelines for network anomaly detection systems. In *Proceedings of International Symposium on Recent Advances Intrusion Detection*.

[80] B. Scholkopf, J.C. Platt, J.Shawe-Taylor, A.J. Smola, and R.C. Williamson. Estimating the support of a high-dimensional distribution. In *Technical report, Microsoft Research, MSR-TR-99-87*, pages 1443–1471, Cambridge, MA, USA, 1999.

[81] SecureWorks. Ozdok/Mega-D Trojan Analysis. `http://www.secureworks.com/research/threats/ozdok/?threat=ozdok`.

[82] Vyas Sekar, Yinglian Xie, Mike Reiter, and Hui Zhang. A Multi-Resolution Approach for Worm Detection and Containment. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 189–198, Philadelphia, PA, USA, 2006.

[83] D. Senn, D. Basin, and G. Caronni. Firewall conformance testing. In *Proceedings of IFIP International Conference on Testing of Communicating Systems (TestCom)*, pages 226–241, Montreal, Canada, 2005.

[84] Shadowserver. Botnet Measurement and Study. `http://shadowserver.org/wiki/`.

[85] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed. In *Proceedings of Operating System Design and Implementation (OSDI)*, pages 1–6, Vancouver, BC, Canada, 2010.

[86] Seungwon Shin and Guofei Gu. Conficker and Beyond: A Large-Scale Empirical Study. In *Proceedings of Annual Computer Security Applications Confer-*

ence (ACSAC), Austin, TX, USA, 2010.

[87] Seungwon Shin and Guofei Gu. Cloudwatcher: Network security monitoring using openflow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?). In *Proceedings of Workshop on Secure Network Protocols (NPSec)*, pages 1–6, Austin, TX, USA, October 2012.

[88] Seungwon Shin, Phil Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. Fresco: Modular composable security services for software-defined networks. In *Proceedings of Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2013.

[89] Snort. `http://snort.org`.

[90] SPAMHAUS. The SPAMHAUS Project. `http://www.spamhaus.org/`.

[91] SRI-International. An analysis of Conficker C. `http://mtc.sri.com/Conficker/addendumC/`.

[92] Elizabeth Stinson and John C. Mitchell. Characterizing the Remote Control Behavior of Bots. In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 89–108, Lucerne, Switzerland, 2007.

[93] Elizabeth Stinson and John C. Mitchell. Towards systematic evaluation of the evadability of bot/botnet detection methods. In *Proceedings of USENIX workshop on offensive technologies (WOOT)*, San Jose, CA, USA, 2008.

[94] Ben Stock, Markus Engelberth Jan Goebel, Felix C. Freiling, and Thorsten Holz. Walowdac Analysis of a Peer-to-Peer Botnet. In *Proceedings of European Conference on Computer Network Defense (EC2ND)*, pages 13–20, Milan, Italy, 2009.

[95] Symantec. Trojan.Srizbi. http://www.symantec.com/security_response/writeup.jsp?docid=2007-062007-0946-99.

[96] Microsoft Security Techcenter. Conficker worm. http://technet.microsoft.com/en-us/security/dd452420.aspx.

[97] I. Trestian, S. Ranjan, A. Kuzmanovic, and A. Nucci. Unconstrained Endpoint Profiling (Googling the Internet). In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 279–290, Seattle, WA, USA, 2007.

[98] Jamie Twycross and Matthew M. Williamson. Implementing and testing a virus throttle. In *Proceedings of the USENIX Security Symposium*, pages 20–20, Washington, DC, USA, 2003.

[99] VmWare. Vmware virtualization software for desktops servers. http://www.vmware.com/.

[100] Andreas Voellmy and Paul Hudak. Nettle: Functional reactive programming of openflow networks. In *Yale University Technical Report*, New Haven, CT, USA, 2010.

[101] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A language for high-level reactive network control. In *Proceedings of ACM SIGCOMM HotSDN workshop*, pages 43–48, Helsinki, Finland, 2012.

[102] David Watson. Know Your Enemy: Containing Conficker. http://www.honeynet.org/papers/conficker.

[103] Rhiannon Weaver. A Probabilistic Population Study of the Conficker-C Botnet. In *Proceedings of the Passive and Active Measurement Conference (PAM)*, pages 181–190, Zurich, Switzerland, 2010.

[104] WinPcap. The industry-standard windows packet capture library. `http://www.winpcap.org/`.

[105] G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, pages 2170–2183, Miami, FL, USA, 2005.

[106] Yinglian Xie, Fang Yu, Kannan Achan, Eliot Gillum, Moises Goldzmidt, and Ted Wobber. How Dynamic are IP Addresses? In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 301–312, Kyoto, Japan, 2007.

[107] Yinglian Xie, Fang Yu, Kannan Achan, Rina Panigraphy, Geoff Hulte, and Ivan Osipkov. Spamming Botnets: Signatures and Characteristics. In *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 171–182, Seattle, WA, USA, 2008.

[108] T.-F. Yen and M. K. Reiter. Traffic aggregation for malware detection. In *Proceedings of International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA2008)*, pages 207–227, Paris, France, 2008.

[109] T.-F. Yen and M. K. Reiter. Are your hosts trading or plotting? telling p2p file-sharing and bots apart. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 241–252, Genoa, Italy, 2010.

[110] Yuanyuan Zeng, Xin Hu, and Kang G. Shin. Detection of Botnets Using Combined Host- and Network-Level Information. In *Proceedings of Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 291–300, Chicago, IL, USA, June 2010.