SECURING COMPUTER NETWORKS: ACCESS CONTROL MANAGEMENT AND ATTACK SOURCE IDENTIFICATION

By

MYUNGKEUN YOON

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2008

© 2008 MyungKeun Yoon

Dedicated to my family

ACKNOWLEDGMENTS

First of all, I would like to express my deepest gratitude to Prof. Shigang Chen, my advisor, for his tireless guidance and encouragement throughout my graduate studies.

My special thanks also go to Prof. Sartaj Sahni, Prof. Jose Fortes, Prof. Ye Xia and Prof. Dapeng Wu for their instructive comments and support during my years at the University of Florida (UF).

I would like to thank all my colleagues in Prof. Chen's research group, including Yong Tang, Zhan Zhang, Liang Zhang, Ying Jian, Ming Zhang, Tao Li and Parbati Kumar Manna, for providing a high level of research support.

Last but not least, I want to thank my family for their support, love, understanding and many sacrifices they had to make throughout my graduate studies. I would like to say that I love Joon-Sup, Joon-Ho and especially Hye-Jung.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS . . . 4

LIST OF TABLES . . . 8

LIST OF FIGURES . . . 9

ABSTRACT . . . 12

CHAPTER

1 INTRODUCTION . . . 14

2 MINIMIZING THE MAXIMUM FIREWALL RULE SET IN A NETWORK WITH MULTIPLE FIREWALLS . . . 18

2.1 Motivation . . . 18
2.2 Related Works . . . 21
2.3 Problem Definition . . . 22
2.3.1 Network Model . . . 22
2.3.2 Notations . . . 22
2.3.3 Problems . . . 24
2.3.4 Rule Graph and Topology Graph . . . 25
2.3.5 Robustness . . . 27
2.4 NP-Completeness . . . 27
2.4.1 k-Firewall Decision Problem $\in$ NP . . . 28
2.4.2 NP-Hardness . . . 28
2.5 HAF: A Heuristic Algorithm for FPP, Partial FPP, FRP, Partial FRP, and Weighted FPP/FRP . . . 30
2.5.1 Overview . . . 30
2.5.2 Augmented Graph $G_t^{\langle x,y\rangle}$ and MinMax Path . . . 31
2.5.3 Find the MinMax Path in $G_t^{\langle x,y\rangle}$ . . . 32
2.5.4 Insert the MinMax Path to G_t . . . 34
2.5.5 Ensuring Connectivity . . . 34
2.5.6 Complexity Analysis . . . 37
2.5.7 Modifying HAF for FRP, partial FRP, and Weighted FPP/FRP . . . 37
2.6 Simulation . . . 37

3 A NOVEL INCREMENTALLY-DEPLOYABLE PATH ADDRESS SCHEME FOR THE INTERNET . . . 49

3.1 Motivation . . . 49
3.2 Related Work . . . 52
3.3 Path Address Scheme . . . 55
3.3.1 Objectives . . . 55
3.3.2 Definition of Path Address . . . 56

3.3.3 Extending Routing Protocol for Path Address 57
3.3.4 New Fields in Packet Header and Path Address Verification 59
3.3.5 Alternative Version of Path Address against Router Compromise . . 62
3.3.6 Self-Completeness of PAS for Incremental Deployment 64
3.4 Evaluation 66
3.4.1 Analysis 66
3.4.1.1 Analytical model 66
3.4.1.2 False-positive probability and false-negative probability of PAS 67
3.4.1.3 False-positive probability and false-negative probability of Pi 68
3.4.2 Simulations 69
3.4.2.1 Simulation setup 69
3.4.2.2 Performance evaluation with respect to attacker ratio . . . 71
3.4.2.3 Performance evaluation with respect to network topology . 71
3.4.2.4 Performance comparison with respect to r 72
3.4.2.5 Performance evaluation under incremental deployment . . 72

4 FIT A SPREAD ESTIMATOR IN A SMALL MEMORY 78

4.1 Motivation 78
4.2 Existing Spread Estimators 81
4.3 Design of Compact Spread Estimator (CSE) 83
4.3.1 Motivation for Virtual Vectors 83
4.3.2 CSE: Storing Contacts in Virtual Vectors 84
4.3.3 CSE: Spread Estimation 85
4.3.4 System Architecture 88
4.4 Analysis 89
4.4.1 Mean and Variance of $\hat{k}_1$ and $\hat{k}_2$ 89
4.4.2 Estimation Bias and Standard Deviation 92
4.5 Experiments 93
4.5.1 Experiment Setup 94
4.5.2 Accuracy of Spread Estimation 95
4.5.3 Impact of Different s Values on Performance of CSE 96
4.5.4 Impact of Different Column Sizes on Performance of OSM 97
4.5.5 An Application: Detecting Address Scan 97

5 REAL-TIME DETECTION OF INVISIBLE SPREADERS 103

5.1 Motivation 103
5.2 Invisible-Spreader Detection 105
5.2.1 Invisible-Spreader Detection Filter (ISD) 106
5.2.2 Parameter Configuration 107
5.3 Experiment 109
5.3.1 Traffic Trace and Implementation Details 110
5.3.2 Experimental Results 111

6 CONCLUSION 115

REFERENCES 116

BIOGRAPHICAL SKETCH 121

LIST OF TABLES

Table page

2-1 Frequently-used notations 41

2-2 Default simulation parameters 41

4-1 Bias with respect to s and k 99

4-2 False positive ratio and false negative ratio with respect to memory size. 102

4-3 With $\varepsilon = 10\%$, false positive ratio and false negative ratio with respect to memory size. 102

4-4 With $\varepsilon = 20\%$, false positive ratio and false negative ratio with respect to memory size. 102

5-1 Parameter configuration examples ($c = 10$) 113

LIST OF FIGURES

Figure | page

2-1 Two topologies that connect domains, x, u, v and y, via firewalls, f_1, f_2 and f_3, whose numbers of interfaces are 2, 3 and 2, respectively. 41

2-2 Rule matrix, rule graph, and topology graph 42

2-3 High-availability solutions 42

2-4 Pseudo code of HAF 43

2-5 Pseudo code of HAF_Dijkstra 44

2-6 (a) Augmented graph $G_t^{\langle x,y\rangle}$, where f_2 has one free interface and two virtual links; (b) Shortest path returned by Shortest_Path($G_t^{\langle x,y\rangle}, x, y$), where the relaxation is performed from y along the path to x; (c) Shortest path returned by Shortest_Path($G_t^{\langle x,y\rangle}, y, x$), where the relaxation is performed from x along the path to y. The best path is (x, f_1, v_1, f_2, y). 45

2-7 Shortest path when f_2 has two free interfaces. 45

2-8 Shortest path when f_2 and f_3 each have one free interface. 45

2-9 Pseudo code of Insert_Optimal_Path 46

2-10 Size of maximum rule set with respect to number n of domains. $10 \leq n \leq 120$, $m = 40$, $\overline{e(f)} = 4$, $\overline{r(i,j)} = 10$, $p = 0.7$. 47

2-11 Size of maximum rule set with respect to number m of firewalls. $n = 100$, $35 \leq m \leq 59$, $\overline{e(f)} = 4$, $\overline{r(i,j)} = 10$, $p = 0.7$. 47

2-12 Size of max rule set with respect to avg number $\overline{e(f)}$ of network interfaces per firewall. $n = 100$, $m = 40$, $3.5 \leq \overline{e(f)} \leq 6$, $\overline{r(i,j)} = 10$, $p = 0.7$. 47

2-13 Size of maximum rule set with respect to avg number $\overline{r(i,j)}$ of rules per domain pair. $n = 100$, $m = 40$, $\overline{e(f)} = 4$, $10 \leq \overline{r(i,j)} \leq 50$, $p = 0.7$. 47

2-14 Size of maximum rule set with respect to probability p. $n = 100$, $m = 40$, $\overline{e(f)} = 4$, $\overline{r(i,j)} = 10$, $0.3 \leq p \leq 1.0$. 48

2-15 Size of maximum rule set in sparse network. $10 \leq n \leq 120$, $m = (n-1)/(\overline{e(f)} - 1)$, $\overline{e(f)} = 4$, $\overline{r(i,j)} = 50$, $p = 1.0$. 48

3-1 Pi cannot be used for path address. 73

3-2 Local numbers of the interdomain routers. 74

3-3 Addresses for the routing paths from the routers to $AS1$. For example, $R8.paddr(AS1) =$ 01001010. It is the XOR of all local numbers on the routing path $R8 \rightarrow R7 \rightarrow R6 \rightarrow R5 \rightarrow R4 \rightarrow R3 \rightarrow R2 \rightarrow R1$. Alternatively it can be viewed as the XOR of $R8$'s local number and $R7.paddr(AS1)$. 74

3-4 Received values of the paddr and verification fields are shown beside each router. The two fields are set to zeros by the sender. The first interdomain router sets these fields with appropriate values. The path address field stays unchanged at the subsequent hops, but the verification field is XORed by the local number at each hop. The verification field should be zero when the packet reaches its receiver. 74

3-5 Malicious host in $AS4$ sets the paddr/verification fields arbitrarily with the P flag being one. As long as it does not know $R6.paddr(AS1)$, the attack packets to $AS1$ will be classified as abnormal, which is indicated by a cross below V in the figure. 75

3-6 Path address between $AS3$ and $AS1$ should be artificially made different from the address between $AS6$ and $AS1$. 75

3-7 *Left:* false positive ratios with respect to attacker ratio. *Right:* false negative ratios with respect to attacker ratio. 76

3-8 *Left:* false positive ratios with respect to fraction of degree-one nodes. *Right:* false negative ratios with respect to fraction of degree-one nodes. 76

3-9 *Left:* false positive ratios with respect to r. *Right:* false negative ratios with respect to r. 76

3-10 *Left:* false positive ratios with respect to deployment ratio. *Right:* false negative ratios with respect to deployment ratio. 77

4-1 The approximation error is very small when s is reasonably large. 99

4-2 Traffic distribution: each point shows the number of sources having a certain spread value. 99

4-3 $m = 0.5$MB. Each point in the first plot (CSE) or the second plot (OSM) represents a source, whose x coordinate is the true spread k and y coordinate is the estimated spread $\hat{k}$. The third plot shows the bias of CSE and OSM, which is the measured $E(\hat{k}-k)$ with respect to k. The fourth plot shows the standard deviation, which is the *measured* $\frac{\sqrt{Var(\hat{k})}}{k}$ for CSE and OSM, together with the *numerically-calculated* standard deviation for CSE based on (4–26) and (4–24). 100

4-4 $m = 1$M. See the caption of Fig. 4-3 for explanation. 100

4-5 $m =$2MB. See the caption of Fig. 4-3 for explanation. 100

4-6 $m =$4M. See the caption of Fig. 4-3 for explanation. 100

4-7 Left plot shows the bias of CSE, which is the measured $E(\hat{k} - k)$ with respect to k. Right plot shows the standard deviation of CSE, which is the *measured* $\frac{\sqrt{Var(\hat{k})}}{k}$. . . . 101

4-8 Left plot shows the bias of OSM, which is the measured $E(\hat{k} - k)$ with respect to k. Right plot shows the standard deviation of OSM, which is the *measured* $\frac{\sqrt{Var(\hat{k})}}{k}$. . . . 101

4-9 Left plot shows the distribution of $(k, \hat{k})$ for all sources under OSM when $r = 64$, where k and $\hat{k}$ are the true spread and the estimated spread, respectively. Right plot shows the distribution of $(k, \hat{k})$ under OSM when $r = 256$. . . . 101

5-1 Cumulative ratios of the numbers of distinct sources and distinct source/destination tuples with respect to source spread . . . 113

5-2 Cumulative ratios of the numbers of distinct destinations and distinct source/destination tuples with respect to destination spread . . . 113

5-3 Number of false negatives when M=256KB . . . 114

5-4 Number of false positives when M=256KB . . . 114

5-5 Number of false negatives when M=1MB . . . 114

5-6 Number of false positives when M=1MB . . . 114

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

SECURING COMPUTER NETWORKS: ACCESS CONTROL MANAGEMENT AND ATTACK SOURCE IDENTIFICATION

By

MyungKeun Yoon

December 2008

Chair: Shigang Chen
Major: Computer Engineering

We study the problem of securing computer networks. We mainly focus on two issues: managing access control lists of multiple firewalls and identifying attack sources. As the number of firewalls increases in computer networks, it is crucial to deploy the firewalls and to build an efficient access control list on each of them. Multiple firewalls cooperate to implement the access control by filtering out unwanted packets. The source address of a packet is a decisive parameter when the filtering is carried out. For example, edge firewalls between the intranet and the Internet may use dynamic filters, which can block packets of suspicious source addresses in order to defeat denial of service attacks. However, wily attackers may play tricks to give false information about their source addresses. Therefore, attack sources should be exactly identified before the filtering is applied. In this dissertation, we propose three novel techniques.

First, we study the problem of placing multiple firewalls in an enterprise network. A firewall's complexity is known to increase with the size of its access control list, i.e. rule set. When designing a security-sensitive network, it is critical to construct the network topology and its routing structure carefully in order to reduce the firewall rule sets, which helps lower the chance of security loopholes and prevent performance bottleneck. We study the problems of how to place the firewalls in a topology during network design and how to construct the routing tables during operation, such that the maximum firewall rule set can be minimized.

Second, we study the problem of identifying attack sources on the Internet. It is crucial to find out attacker's unique address before the corresponding filtering rule is activated at the edge firewalls. On the current Internet, not only is a host free to send packets to any destination address, but also it is free to forge any source address that it does not own. This freedom creates a huge security problem. The victims under attack do not know where the malicious packets are actually from and which sources should be blocked because, with forged source addresses, the malicious packets may appear to come from all over the Internet. We propose a path address scheme to identify attackers even when they use spoofed source addresses. Under this scheme, each path on the Internet is assigned a path address. IP addresses are owned by the end hosts; path addresses are owned by the network, which is beyond the reach of the hosts.

Third, we study the problems of spread estimation and spreader detection. The spread of a source host is the number of distinct destinations that it has sent packets to during a measurement period. A spread estimator is a software/hardware module on a router that inspects the arrival packets and estimates the spread of each source. It has important applications in detecting port scans and DDoS attacks, measuring the infection rate of a worm, assisting resource allocation in a server farm, determining popular web contents for caching, to name a few. We design a new spread estimator that delivers good performance in tight memory space where all existing estimators no longer work.

We also study the problem of detecting spreaders. We call an external source address a *spreader* if it connects to more than a threshold number of distinct internal destination addresses during a period of time (such as a day). We note that none of the current intrusion detection systems can identify spreaders in real-time if the attacker slows down in sending attack packets. We call such an attacker an invisible spreader. We observe that normal traffic has strong skewness especially in an enterprise (or university campus) network. We propose a new scheme to detect invisible spreaders by exploiting the traffic skewness.

CHAPTER 1
INTRODUCTION

As computer networks play vital roles in companies or institutions, securing them is crucial. Once an enterprise network is compromised, it causes a severe financial loss and a decline of public trust. On the other hand, enterprise networks are good targets of wily attackers who willingly spare no pains in breaking into the networks. Therefore, enterprise networks require a high level of security. It means that people are willing to trade off efficiency for enhanced security unless the efficiency degradation is significant.

An enterprise network consists of domains (subnets) that are connected with each other through firewalls. Each firewall has an interdomain access control list, i.e. rule set, which prevents unwanted packets from traversing different domains. At least one of the firewalls connects to the Internet for providing Internet services. As enterprise networks evolve, the number of domains and firewalls increases. A firewall's complexity is known to increase with the size of its rule set. Empirical studies show that, as the rule set grows larger, the number of configuration errors on a firewall increases sharply, while the performance of the firewall degrades. When designing a security-sensitive network, it is critical to construct the network topology and its routing structure carefully in order to reduce the firewall rule sets, which helps lower the chance of security loopholes and prevent performance bottleneck. In Chapter 2, we study the problems of how to place the firewalls in a topology during network design and how to construct the routing tables during operation, such that the maximum firewall rule set can be minimized. These problems have not been studied adequately despite their importance. We have two major contributions. First, we prove that the problems are NP-complete. Second, we propose a heuristic solution and demonstrate the effectiveness of the algorithm by simulations. The results show that the proposed algorithm reduces the maximum firewall rule set by $2 \sim 5$ times when comparing with other algorithms.

A firewall filters out unwanted packets after comparing their five-tuple values or subset values against its rule set. The five-tuple value consists of source address, destination address, source port number, destination port number and protocol type. However, wily attackers may play tricks to give false information about their source addresses. Therefore, identifying attack sources is required before the filtering is applied. Otherwise, the firewall may block legal users or may not prevent attackers.

On the current Internet, not only is a host free to send packets to any destination address, but also it is free to forge any source address that it does not own. This freedom creates a huge security problem. The victims under attack do not know where the malicious packets are actually from and which sources should be blocked because, with forged source addresses, the malicious packets may appear to come from all over the Internet. One important question is, if the source addresses from the attack packets are not reliable, what other kind of information is necessary. In Chapter 3, we propose a path address scheme to identify attackers even when they use spoofed source addresses. Under this scheme, each path on the Internet is assigned a path address. IP addresses are owned by the end hosts; path addresses are owned by the network, which is beyond the reach of the hosts. The path address carried in a packet is set by the network and reliably points out where the packet is from. Blocking a path address filters out the packets from an attack source. The victims may even require path-address based filters to be pushed into the network and all the way to the attack source. The path address scheme has desirable features that the previous works do not have. First, it can simultaneously keep false-positive (normal hosts misclassified as attackers) and false-negative (attackers misclassified as normal hosts) ratios to almost zero. Second, malicious packets can be blocked near the attackers. Third, attackers cannot gain any advantage by residing near the victims. By analysis and simulations, we show that the path address scheme is very effective in filtering out malicious packets.

In some cases, changing the source address, IP spoofing, is of no benefit to attackers. When they want to receive any feedback from the victim, the attackers should use the same source address. Otherwise, the reply packet will be sent to the spoofed address on the Internet and never come back. Using the same source address increases the risk of detection. However, wily attackers can evade the intrusion detection systems without changing their source addresses. Detecting such attackers is a challenging problem. We study the problems of spread estimation and spreader detection, which helps detect such attackers.

The spread of a source host is the number of distinct destinations that it has sent packets to during a measurement period. A spread estimator is a software/hardware module on a router that inspects the arrival packets and estimates the spread of each source. It has important applications in detecting port scans and DDoS attacks, measuring the infection rate of a worm, assisting resource allocation in a server farm, determining popular web contents for caching, to name a few. The main technical challenge is to fit a spread estimator in a fast but small memory (such as SRAM) in order to operate it at the line speed in a high-speed network. In Chapter 4, we design a new spread estimator that delivers good performance in tight memory space where all existing estimators no longer work. The new estimator not only achieves space compactness but operates more efficiently than the existing ones. Its accuracy and efficiency come from a new method for data storage, called virtual vectors, which allow us to measure and remove the errors in spread estimation. We perform experiments on real Internet traces to verify the effectiveness of the new estimator.

We call an external source address a *spreader* if it connects to more than a threshold number of distinct internal destination addresses during a period of time (such as a day). Detecting spreaders helps intrusion detection systems identify potential attackers. The existing work can only detect aggressive spreaders that scan a large number of distinct addresses in a short period of time. However, stealthy spreaders may perform

scanning deliberately at a low rate. We observe that these spreaders can easily evade the detection because their small traffic footprint will be covered by the large amount of background normal traffic that frequently flushes any spreader information out of the intrusion detection system's memory. In Chapter 5, we propose a new streaming scheme to detect stealthy spreaders that are invisible to the current systems. The new scheme stores information about normal traffic within a limited portion of the allocated memory, so that it will not interfere with spreaders' information stored elsewhere in the memory. The proposed scheme is light weight; it can detect invisible spreaders in high-speed networks while residing in SRAM. Through experiments using real Internet traffic traces, we demonstrate that our new scheme detects invisible spreaders efficiently while keeping both false-positives (normal sources misclassified as spreaders) and false-negatives (spreaders misclassified as normal sources) to low level.

CHAPTER 2
MINIMIZING THE MAXIMUM FIREWALL RULE SET IN A NETWORK WITH MULTIPLE FIREWALLS

A firewall's complexity is known to increase with the size of its rule set. Empirical studies show that, as the rule set grows larger, the number of configuration errors on a firewall increases sharply, while the performance of the firewall degrades. When designing a security-sensitive network, it is critical to construct the network topology and its routing structure carefully in order to reduce the firewall rule sets, which helps lower the chance of security loopholes and prevent performance bottleneck. This chapter studies the problems of how to place the firewalls in a topology during network design and how to construct the routing tables during operation, such that the maximum firewall rule set can be minimized. These problems have not been studied adequately despite their importance. We have two major contributions. First, we prove that the problems are NP-complete. Second, we propose a heuristic solution and demonstrate the effectiveness of the algorithm by simulations. The results show that the proposed algorithm reduces the maximum firewall rule set by $2 \sim 5$ times when comparing with other algorithms.

2.1 Motivation

Firewalls are the cornerstones of corporate network security. Once a company acquires firewalls, the most crucial management task is to correctly configure the firewalls with security rules [1]. A firewall's configuration contains a large set of access control rules, each specifying source addresses, destination addresses, source ports, destination ports, one or multiple protocol ids, and an appropriate action. The action is typically "accept" or "deny." Some firewalls can support other types of actions such as sending a log message, applying a proxy, and passing the matched packets into a VPN tunnel [2]. For most firewalls, the rule set is order-sensitive [3]. An incoming packet will be checked against the ordered list of rules. The rule that matches first decides how to process the packet. Other firewalls (such as early versions of Cisco's PIX) use the best-matching rule instead.

Due to the multi-dimensional nature of the rules (including source/destination addresses and ports), the performance of a firewall degrades as the number of rules increases. Commercially deployed firewalls often carry tens of thousands of rules, creating performance bottlenecks in the network. More importantly, the empirical fact shows that the number of configuration errors on a firewall increases sharply in the size of the rule set [4]. A complex rule set can easily lead to mistakes and mal-configuration. After analyzing the firewall rule sets from many organizations including telecommunication companies and financial institutes, Wool [4] quantified the complexity of a rule set as $R + O + \frac{I \times (I-1)}{2}$, where R is the number of rules in the set, O is the number of network objects referenced by the rules, and I is the number of network interfaces on the firewall. The number of network objects and the number of interfaces are normally much smaller than the number of rules. Therefore, it is very important to keep a firewall's rule set as small as possible in order to lower the chance of security loopholes [4]. In a network with multiple firewalls, reducing the number of rules requires not only local optimization at individual firewalls but also global optimization across all firewalls. This chapter studies how to minimize the maximum rule set among all firewalls in the network, which has not been adequately studied despite its importance in practice.

We investigate a family of related problems. The first one is about how to place the firewalls in a topology during network design. The so-called *firewall placement problem* (FPP) is to find the optimal placement of firewalls that connects a set of domains in such a way that minimizes the maximum number of rules on any firewall. The second problem, called *partial FPP*, is to expand an existing topology with new firewalls and domains such that the maximum rule set remains minimized. This problem arises during incremental deployment or in case that a partial network topology has been determined based on more important performance criteria before firewall rule sets are considered. FPP is a special case of partial FPP (with an empty existing topology).

We now move to operational networks whose topologies have already been fully established. Our third problem, called *firewall routing problem* (FRP), is to establish the optimal routing paths on an *existing* network topology such that the maximum number of rules on any firewall is minimized. The fourth problem is called *partial FRP*. It assumes that the routing tables in the network have been *partially* populated based on other performance criteria (such as reliability and bandwidth utilization). For example, if bandwidth is the most important criterion, some routing entries should be selected to optimize the use of bottleneck links, but the choice of other entries may be flexible if alternative paths after bottleneck links are allowed (since end-to-end bandwidth is solely decided by the bottleneck). In this case, we can determine those routing entries by using the secondary criterion of minimizing the maximum firewall rule set.

Our fourth and fifth problems are called *weighted FPP/FRP*. We assign each rule a weight (possibly representing the volume of traffic covered by this rule), and assign each firewall a weight (possibly representing the capacity of the firewall). The goal is to find the optimal network topology and/or routing paths that minimize the maximum weighted number of rules at any firewall. The solutions to the weighted problems take not only the number of rules but also traffic distribution, firewall performance and possibly other factors into consideration.

We have two major contributions. First, by reducing the well-known set-partition problem to the above problems, we prove that they are NP-complete. Second, we propose a heuristic algorithm to solve the FPP problem approximately. Not only does it construct a network topology among domains and firewalls, but also identify routing paths that minimize the maximum firewall rule set. The algorithm can be easily modified to solve partial FPP, FRP, partial FRP, weighted FPP, and weighted FRP. Hence, the algorithm can be used to construct a new topology, complete a topology that has been partially constructed (based on other performance criteria), expand an existing topology, or work on an established topology to build a new routing structure or complete an existing

routing structure that has been partially populated (based on other performance criteria). We demonstrate its effectiveness by simulations, which show that the proposed algorithm achieves far better results than two other solutions. The maximum size of all firewall rule sets produced by our algorithm is $2 \sim 5$ times smaller than those produced by others.

The rest of the chapter is organized as follows. Section 3.2 defines the network model and the problems to be solved. Section 2.4 proves that the problems are NP-complete. Section 5.2 proposes a heuristic algorithm. Section 3.4.2 presents the simulation results. Section 2.2 surveys the related work.

2.2 Related Works

Gouda and Liu developed a sequence of five algorithms that can be applied to generate a compact rule set while maintaining the consistency and completeness of the original rule set [5]. They also proposed a method for diverse firewall design and presented algorithms to detect discrepancies between two rule sets [6]. Recently Liu et al. proposed a novel algorithm for minimizing security policies of a firewall [7].

Wool investigated the direction-based filtering in firewalls [8]. Fulp studied the problem of reducing the average number of rules that must be examined for each packet [9]. Al-Shaer and Hamed identified anomalies that exist in a single- or multi-firewall environment, and presented a set of techniques to discover configuration anomalies in centralized and distributed legacy firewalls [10]. Smith et al. studied the problem of how to place a set of firewalls in a complex network to minimize cost and delay [11] and the problem of how to increase comprehensiveness and level of confidence in protection [12]. El-Atawy et al. proposed to optimize packet filtering performance by traffic statistical matching [13]. Hamed et al. designed algorithms that maximize early rejection of unwanted packets and utilize traffic characteristics to minimize the average packet matching time [14].

Packet filtering can be viewed as a special case of packet classification [15], which is to determine the first matching rule for each incoming packet at a router. Much work has

focused on solving the problem of how to find matching rules as quickly as possible by using sophisticated data structures or hardware-driven approaches [16–19]. Other work proposed algorithms for removing redundancy in packet classifiers [20, 21].

2.3 Problem Definition

2.3.1 Network Model

We consider a security-sensitive enterprise network consisting of domains (subnets) that are connected with each other through firewalls. We assume that intradomain security is appropriately enforced. This chapter focuses on interdomain access control. We further assume dynamic routing is turned off on firewalls while static routes are used to direct interdomain traffic, which is today's common practice in banks or other institutions that have high-level security requirements. In fact, some popular firewalls (such as many Cisco PIX models) do not support dynamic routing protocols. With static routes, robustness is achieved by using dual firewalls, which will be discussed shortly. Using static routes on firewalls is a direct consequence of the high complexity in managing the security of a mesh network. It has a number of practical advantages. First, it ensures that traffic flows are going through their designated firewalls where appropriate security policies are enforced. Second, predictable routing paths simplify the security analysis in a complex network environment and consequently reduce the chance of error in firewall configuration. Third, most existing dynamic routing protocols are not secure. Counterfeit routing advertisement can divert traffic through insecure paths where the packets may be copied or tampered. Note that dynamic routing is still used inside each domain as long as it does not cross an inter-domain firewall.

2.3.2 Notations

Let N be a set of n domains and M a set of m firewalls. Each firewall has two or more network interfaces. Different firewalls may have different numbers of interfaces. A network interface can be connected to any domain, forming a physical link between the firewall and the domain. In our model, two firewalls do not directly connect with each

other because otherwise we would treat them as one firewall with combined interfaces; two domains do not directly connect with each other because otherwise we would treat them as one domain. Let e be the total number of network interfaces available on all firewalls. The maximum number of links in the topology is bounded by e. A network interface that has not been used to connect a domain is called a *free interface*.

Each domain has one address prefix. Static routes are defined to route interdomain traffic, which ensures that each traffic flow has a specific path going through certain firewall(s) where the security policy governing this flow will be enforced. In order to support stateful inspection, routing symmetry is assumed. It means that the routing path from domain x to domain y is the same as the path from y to x, $\forall x, y \in N$. This assumption is made to comply with Cisco's CBAC (context-based access control) and other firewalls' stateful inspection mechanisms, which allow the system administrator to only specify the rules for traffic from clients to servers, while the firewall automatically inserts the rules for the return traffic on the fly. CBAC requires that a connection uses the same (interdomain) path for two-way communication. We want to stress that this assumption is made only for practical reasons. Our analysis and algorithm design can be easily modified to work for asymmetric routing.

For each pair of domains $x, y \in N$, there is a set $R(x, y)$ of access control rules, defining the traffic flows that are permitted from domain x to domain y. The optimization of the rule set is beyond the scope of this dissertation. Let $r(x, y) = |R(x, y)|$. Similarly the number of rules from y to x is denoted as $r(y, x)$. The total number of rules between the two domains is $r(x, y) + r(y, x)$. Once the routing path between x and y is determined, these rules will be enforced on the firewalls along the path. Each firewall may sit in the routing paths between many pairs of domains, and its rule set will be the aggregate of all rules between those domains. We want to construct the network topology and/or lay out the routing paths to avoid creating large firewall rule sets in the network. We assume that wild-card rules are processed separately. For example, if a domain requires to deny all

external packets from reaching an internal subnet, a wild-card deny rule for that subnet will be installed at all firewalls adjacent to the domain. Since wild-card rules typically account for a small portion of a large rule set, for simplicity, when we compute the size of a firewall rule set, we ignore the contribution of wild-card rules in this dissertation.

For any firewall $f \in M$, let $W(f)$ be the set of access control rules to be enforced by f. Let $w(f) = |W(f)|$. If f sits in the routing path from domain x to domain y, then it enforces all rules between them and thus $R(x,y) \subseteq W(f)$; otherwise, it does not enforce those rules and thus $R(x,y) \bigcap W(f) = \emptyset$. Let $\Pi(f)$ be the set of domain pairs, $\langle x, y\rangle$, with the routing path from x to y passing through f. We have

$$\begin{aligned} W(f) &= \bigcup_{\langle x,y\rangle \in \Pi(f)} R(x,y) \\ w(f) &= \sum_{\langle x,y\rangle \in \Pi(f)} r(x,y) \end{aligned} \tag{2–1}$$

Some frequently-used notations are listed in Table 2-1 for quick reference.

2.3.3 Problems

As the example in Fig. 2-1 shows, there are many ways to connect a set of domains via a set of firewalls. For any network topology, there are different ways to lay out the routing paths. In general, the rule sets to be enforced on the firewalls will be different when we change the network topology or the routing paths.

Definition 1. *The firewall placement problem (FPP) is to (a) optimally connect a set of domains via a set of firewalls to form a network topology and (b) establish optimal inter-domain routing tables on this topology, such that the maximum number of access control rules on any firewall, i.e.,* $max_{f \in M}\{w(f)\}$*, is minimized.*

Definition 2. *The partial FPP is the same as FPP except that it works on a given partially-constructed topology that allows limited freedom in the way that the topology can be expanded.*

In practice, the network topology is often fixed and cannot be changed. By optimizing routing paths, we can still reduce the maximum rule set.

Definition 3. *The firewall routing problem (FRP) is to construct optimal inter-domain routing tables on an existing topology to minimize the maximum number of access control rules on any firewall.*

Definition 4. *The partial FRP problem is to complete the partially-populated routing tables to minimize the size of the maximum firewall rule set.*

Moreover, we can introduce weights into the problem definition. Suppose each rule is assigned a weight, representing the expected traffic volume covered by this rule. The higher the traffic volume, the larger the weight. Suppose each firewall is also assigned a weight, representing the firewall's capacity (such as processing speed). The higher the capacity, the larger the weight. We define the *weight of a firewall rule set* to be the total weight of all rules in the set multiplied by the firewall's weight. The weighted version of the above problems is to minimize the maximum weight of any firewall rule set in the network. This allows us to model firewall performance more accurately. For example, in the solutions of the weighted FPP/FRP problems, a firewall with a larger capacity are likely to take more rules or those rules with heavier traffic.

We will prove that all the above problems are NP-complete, and we will design a heuristic algorithm for them. *Instead of enumerating over all problems, our presentation will focus on FPP for analysis and algorithm design. We will show that the results can be trivially extended to other problems.* Focusing on FPP is only a presentation choice because it is easier to extend the solution for FPP to other problems. This presentation choice does not mean that our solution is only designed for topology construction in the network design phase. The solution can also be used for topology expansion and routing optimization in the operation phase, which is probably the more common scenario of application.

2.3.4 Rule Graph and Topology Graph

We use Fig. 2-2 to illustrate a few concepts. There are eight domains with ids from 1 to 8. The rule matrix, $(r(x, y), x, y \in N)$, is shown in Fig. 2-2 (a). We construct a

rule graph (denoted as G_r) in Fig. 2-2 (b), where each node is a domain and there is an undirected *edge* $\langle x, y \rangle$ if $r(x, y) + r(y, x) > 0$. The number of access control rules to be enforced between the two domains, i.e., $r(x, y) + r(y, x)$, is shown beside the link. G_r is a graphical representation of the rule matrix, specifying the security requirement. It will be the input to the algorithm that solves FPP and other problems (approximately because they are NP-complete).

For the output of the algorithm, we define a *topology graph* (denoted as G_t), which consists of a network topology and a routing structure. A node in G_t is either a domain or a firewall. An undirected *link* (x, f) represents a physical connection between a domain x and a firewall f. Note that we use the term "link (x, f)" in G_t, in contrast to the term "edge $\langle x, y \rangle$" in G_r. Each node has a routing table consisting of routing entries, each specifying the next hop for a destination domain.

Suppose there are five firewalls, each having three network interfaces. Fig. 2-2 (c) shows the topology graph returned by the algorithm to be proposed in this dissertation. The number of access control rules enforced on a firewall is shown inside the box that represents the firewall. The routing tables are interpreted as follows. "$rt(1, 2) = f_3$" means the routing table at domain 1 has an entry for destination domain 2 with the next hop being firewall f_3. In reality, the gateway in domain 1 which connects to firewall f_3 must advertise within the domain that it can reach domain 2. Consequently, the routing tables at the internal routers will each have an entry for domain 2, pointing towards that gateway. "$rt(f_1, 1) = 1$" means that the routing table at firewall f_1 has an entry for domain 1 with the next hop being domain 1. It implies that f_1 is directly connected to a gateway in domain 1. Of course, the actual routing entry uses that gateway as the next hop. "$rt(f_1, 2) = 3$" means that the routing table at firewall f_1 has an entry for domain 2 with the next hop being domain 3. It implies that f_1 is directly connected to a gateway in domain 3. The actual routing entry uses that gateway as the next hop and the address

prefix of domain 2 as the destination. The other routing entries in the figure should be interpreted similarly.

Given a rule graph G_r and a set M of firewalls, for each feasible topology graph G_t, we can calculate $w(f), \forall f \in M$. The topology graph that minimizes $\max_{f \in M}\{w(f)\}$ is the solution. FPP has the largest set of feasible topology graphs; partial FPP has a smaller set due to the restriction of a given partial topology. FRP has only one feasible topology with many possible routing structures, while partial FRP gives less freedom in constructing a routing structure.

2.3.5 Robustness

Robustness against node failure is an important issue in network design. While dynamic routing is used inside each domain, we must guard against firewall failure. The most common way to achieve high availability is to use dual firewalls. The state-synchronization solution and the load-balancing solution [22–24] are prevalent in practice. Fig. 2-3 shows one example for each approach. In both cases, firewalls in parallel disposition have the same rule set so that one of them can continue the service when the other fails. Identical co-located firewalls can be logically treated as one in our solution. Therefore, we will not explicitly discuss the use of dual firewalls in the sequel.

2.4 NP-Completeness

In this section, we prove that FPP is NP-complete. The same process can be used to prove the NP-completeness of partial FPP, FRP, partial FRP, and weighted FPP/FRP, which is omitted to avoid excessive repetition.

FPP is an optimization problem. We define the corresponding decision problem as follows: Given a rule graph and a set of firewalls, the *k-firewall decision problem* is to decide whether there exists a topology graph such that $w(f) \leq k$, $\forall f \in M$, where k is an arbitrary, positive integer. To prove the NP-completeness of FPP, it is sufficient to prove its decision problem is NP-complete.

The proof consists of two steps. First, we show the k-firewall decision problem $\in$ NP. Second, we show it is NP-hard by reducing the set-partition problem (known to be NP-complete [25]) to the k-firewall decision problem in polynomial time.

2.4.1 k-Firewall Decision Problem $\in$ NP

To show the decision problem belongs to NP, we need to give a verification algorithm that can verify a solution G_t of the problem in polynomial time. G_t is a topology graph, specifying the network topology and the routing paths between domains. The verification algorithm is described as follows. Initially, $w(f) = 0$, $\forall f \in M$. For each edge $\langle x, y \rangle$ in G_r, we traverse the routing path between domain x and domain y in G_t. For each firewall f on the path, $w(f) := w(f) + r(x, y) + r(y, x)$. There are $O(n^2)$ edges in G_r and the length of a routing path is $O(n + m)$. Therefore, it takes $O(n^2(n + m))$ time to calculate $w(f), \forall f \in M$. After that, it takes $O(m)$ time to verify $w(f) \leq k, f \in M$.

2.4.2 NP-Hardness

We show that the set-partition problem can be reduced to the k-firewall decision problem in polynomial time. In that case, because the set-partition problem is NP-hard [25], the k-firewall decision problem is also NP-hard.

Given a finite set A of positive integers, the set-partition problem is to determine whether there exists a subset $A' \subseteq A$ such that $\sum_{a \in A'} a = \sum_{a \in A - A'} a$. We reduce it to the k-firewall decision problem as follows.

First, for each member $a \in A$, we associate it with a pair of two domains $\langle x_a, y_a \rangle$, and let the number of access control rules from x_a to y_a be a. In total, there are $2|A|$ domains. $N = \{x_a, y_a \mid a \in A\}$. For domain pairs $\langle x_a, y_a \rangle$, $\forall a \in A$, $r(x_a, y_a) = a$, and for all other domain pairs $\langle x, y \rangle$, $r(x, y) = 0$.

Second, we use two firewalls, denoted as f_1 and f_2. The number of network interfaces of each firewall is $2 \times |A|$. k is set to be $\frac{\sum_{a \in A} a}{2}$.

The reduction from the set-partition problem to the above k-firewall decision problem can be done in polynomial time since we only need to convert $|A|$ integers into $|A|$ domain pairs with the rule matrix $(r(x, y), x, y \in N)$ appropriately set.

Next, we prove that the set-partition problem is satisfiable if and only if the corresponding k-firewall decision problem is satisfiable.

First, suppose the set-partition problem is satisfiable, i.e., there exists a subset $A' \subseteq A$ such that $\sum_{a \in A'} a = \sum_{a \in A - A'} a = \frac{\sum_{a \in A} a}{2}$. We construct a topology graph as follows. For each member a in A', we connect both x_a and y_a to f_1, insert a routing path $x_a \rightarrow f_1 \rightarrow y_a$, and add $r(x_a, y_a)$, which equals a, to $w(f_1)$. For each member $a \in A - A'$, we connect both x_a and y_a to f_2, insert a routing path $x_a \rightarrow f_2 \rightarrow y_a$, and add $r(x_a, y_a)$, which equals a, to $w(f_2)$. Finally, we use the remaining free interfaces to make the graph connected. The constructed topology graph has the following property.

$$\begin{aligned} w(f_1) &= \sum_{a \in A'} r(x_a, y_a) = \sum_{a \in A'} a = \frac{\sum_{a \in A} a}{2} = k \\ w(f_2) &= \sum_{a \in A - A'} r(x_a, y_a) = \sum_{a \in A - A'} a = \frac{\sum_{a \in A} a}{2} = k \end{aligned} \tag{2–2}$$

Therefore, the k-firewall decision problem is also satisfiable.

Second, suppose the k-firewall decision problem is satisfiable, i.e., there exists a topology graph such that $w(f_1) \leq k$ and $w(f_2) \leq k$. Recall that $k = \frac{\sum_{a \in A} a}{2}$. We have

$$w(f_1) + w(f_2) \leq 2k = \sum_{a \in A} a \tag{2–3}$$

Each rule has to be enforced by f_1, f_2, or both. Therefore,

$$w(f_1) + w(f_2) \geq \sum_{a \in A} r(x_a, y_a) = \sum_{a \in A} a \tag{2–4}$$

By (4–11) and (2–4), we have

$$w(f_1) + w(f_2) = \sum_{a \in A} a \tag{2–5}$$

Consider an arbitrary member $a \in A$. The routing path from x_a to y_a must only pass either f_1 or f_2 but not both because otherwise the above equation could not hold. Let $\Pi(f)$ be the set of domain pairs, $\langle x_a, y_a \rangle$, whose routing path passes a firewall f. $\Pi(f_1) \bigcap \Pi(f_2) = \emptyset$. Because $w(f_1) \leq k$ and $w(f_2) \leq k$, by (2–5), we have $w(f_1) = w(f_2) = \frac{\sum_{a \in A} a}{2}$. Consider $w(f_1)$ and we have the following equation.

$$\sum_{\langle x_a, y_a \rangle \in \Pi(f_1)} r(x_a, y_a) = \frac{\sum_{a \in A} a}{2} \tag{2–6}$$

Let $A' = \{a \mid \langle x_a, y_a \rangle \in \Pi(f_1)\}$. The above equation can be rewritten as follows.

$$\sum_{a \in A'} r(x_a, y_a) = \sum_{a \in A'} a = \frac{\sum_{a \in A} a}{2} \tag{2–7}$$

Therefore, the set-partition problem is also satisfiable.

2.5 HAF: A Heuristic Algorithm for FPP, Partial FPP, FRP, Partial FRP, and Weighted FPP/FRP

We propose HAF — a Heuristic Algorithm to approximately solve the Firewall problems defined in Section 2.3.3. Our description of the algorithm centers around FPP. In Section 2.5.7, we show that the algorithm can be used to solve other problems.

2.5.1 Overview

The input of HAF is a rule graph G_r and a set M of firewalls and an initial topology graph G_t, which has no link for FPP, but is a partial or full topology for other problems. The output of HAF is a completed topology graph G_t, which consists of domains and firewalls as nodes, links connecting domains and firewalls, and routing tables.

We have shown that the global optimization problem of FPP, which is to find the optimal topology and routing paths that minimize the maximum firewall rule set in the network, is NP-complete. However, constructing an optimal routing path between one pair of domains is a polynomial problem. The basic idea behind HAF is to process the domain pairs one at time and iteratively insert the optimal routing path for each domain pair into a topology graph G_t. After the paths for all domain pairs are inserted,

G_t is an approximate solution to the FPP problem. HAF is particularly useful when the physical network is gradually expanding. After the algorithm produces a topology G_t for the current domains, when a new domain is added to the network, the algorithm can be naturally invoked to process the new domain pairs on top of the existing topology.

The pseudo code of the HAF algorithm is given in Fig. 2-4. For FPP, G_t is initially a topology graph of n domain nodes and m firewall nodes with no link. For each edge $\langle x, y \rangle$ in G_r, the subroutine Insert_Optimal_Path(G_t, x, y) is called to perform the following three tasks.

1. Define the set of feasible routing paths between domain x and domain y.
2. Find the optimal routing path between x and y that minimizes the maximum rule set among all feasible routing paths.
3. Insert the optimal routing path to G_t.

The loop of Lines 2-3 processes the set of edges $\langle x, y \rangle$ in G_r in the descending order of $(r(x, y) + r(y, x))$, which is the total number of rules between domain x and domain y. The topology graph G_t keeps growing as the loop inserts one routing path to G_t in each iteration. In the following, we show how to implement the above three tasks of Insert_Optimal_Path.

2.5.2 Augmented Graph $G_t^{\langle x,y \rangle}$ and MinMax Path

To define the set of feasible paths between domain x and domain y, we first construct an augmented graph $G_t^{\langle x,y \rangle}$ from G_t as follows. The links already in G_t are called *physical links*. For each firewall f with one or more free interfaces, we add a new link between f and x if they are not already connected. Similarly, we add a new link between f and y if they are not already connected. These new links are called *virtual links*. A virtual link may be turned into a physical one if needed. G_t and the virtual links together form the augmented graph $G_t^{\langle x,y \rangle}$. A routing path between x and y in the augmented graph is *feasible* if the following three conditions are satisfied.

- *Routing Condition: The routing path must be consistent with the routing tables at nodes on the path.* For an arbitrary link (v, u) on the path, if v already has a routing entry for destination y but the next hop is not u, then the path is not feasible.
- *Interface Condition: When all virtual links on the path are turned into physical ones, no firewall uses more network interfaces than it has.* Suppose a firewall f has only one free interface and both (f, x) and (f, y) are virtual links in $G_t^{\langle x,y \rangle}$. A path (x, f, y) is not feasible because we cannot turn both (x, f) and (f, y) into physical links.
- *Connectivity Condition: After the path is turned physical, G_t should still have enough free interfaces to turn itself into a connected graph.* Initially, G_t is not a connected graph. In the end, it has to be a connected graph. During the execution of HAF, there should always be enough free interfaces to make new links that are able to connect all separated topological components in G_t. Therefore, if a routing path uses too many free interfaces that makes G_t no longer connectable, then the path is not feasible.

In other words, a path is feasible if we can turn it into a physical path without violating the current routing structure in G_t, exceeding the interface limitation of any firewall, or rendering G_t not connectable.

Definition 5. *The MinMax path in $G_t^{\langle x,y \rangle}$ is the optimal feasible path between x and y that minimizes the maximum rule set on the path.*

Based on the construction of $G_t^{\langle x,y \rangle}$, all virtual links either connect to x or connect to y. Therefore, only the first and last links on the MinMax path may be virtual links.

2.5.3 Find the MinMax Path in $G_t^{\langle x,y \rangle}$

We transform the problem of finding the MinMax path to a variant of the shortest-path problem. We define a *cost* metric on nodes. The cost of a firewall is the size of its rule set, i.e., $w(f)$ as defined in (2–1). The cost of a domain is zero. The cost of a path is the maximum cost (instead of the sum of the costs) of all nodes on the path. One path is

shorter than another path if the cost of the former is smaller or the costs of the two paths are the same but the former has a fewer number of hops. By this definition, the shortest path between x and y must also be the MinMax path.

We design an algorithm, called *HAF_Dijkstra*, to find the shortest path between x and y in $G_t^{\langle x,y \rangle}$. It is an all-source single-destination variant of Dijkstra's algorithm, designed for a graph with 1) *virtual links* (subject to the interface condition stated in the previous subsection), 2) *routing restrictions*, 3) *node costs* instead of link costs, and 4) path length defined as the *maximum node cost* instead of the sum of the node costs on the path. Satisfying the connectivity condition is a rather complex task, which will be ignored for now and addressed in the next subsection, where we will modify the construction of $G_t^{\langle x,y \rangle}$ to include only those virtual links that do not make G_t unconnectable.

Before giving the pseudo code of the algorithm, we define the following variables. $rt[v, d]$ is the routing table entry at node v for destination d. Its value is inherited from G_t. If G_t does not have such a routing entry, the value of $rt[v, d]$ is NIL. $c[v]$ is the cost of node v. $cost[v, d]$ is the *estimated* cost of the shortest path from v to d. $hops[v, d]$ is the *estimated* number of hops on the shortest path from v to d. These two variables are initialized to ∞ and then improved by the algorithm until reaching the optimal values. $next[v, d]$ stores the next hop after v on the shortest path to d. Q is the set of nodes whose shortest paths to d have been found. Extract_Min(Q) and Relax(v, u) are two standard subroutines in Dijkstra's algorithm. Extract_Min(Q) finds the node u in Q that has the smallest $cost[u, d]$ value and, when there is a tie, has the smallest $hops[u, d]$ value. After the shortest path from u to d is found, Relax(v, u) propagates this information to all adjacent nodes v. The pseudo code of the HAF_Dijkstra algorithm is given in Fig. 2-5. ":=" is the assignment sign. s is the source node, and d is the destination node.

Routing_Condition(v, u, d) and Interface_Condition(v, u, s, d) make sure that the Relax subroutine is performed on link (v, u) only when both the routing condition and the interface condition are satisfied. By the construction of Shortest_Path($G_t^{\langle x,y \rangle}, s, d$), the

Routing_Condition and Interface_Condition subroutines are executed iteratively for all links of the shortest path, and therefore, the returned shortest path must be feasible.

HAF_Dijkstra first uses x as the source node and y as the destination node to find the shortest path by calling Shortest_Path($G_t^{\langle x,y \rangle}, x, y$). Then it uses y as the source node and x as the destination node to find the shortest path by calling Shortest_Path($G_t^{\langle x,y \rangle}, y, x$). Finally it returns the shorter one between these two paths. The reason for calling the Shortest_Path subroutine twice is due to the asymmetry caused by virtual links, which is illustrated in Fig. 2-6, assuming each node only has routing entries for directly connected nodes. The clouds, blocks, solid lines, dashed lines, and bold lines represent domains, firewalls, physical links, virtual links, and the shortest paths, respectively. If f_2 has two free interfaces, the shortest path is shown in Fig. 2-7. Another more complicated example is shown in Fig. 2-8, where f_2 and f_3 each have one free interface.

2.5.4 Insert the MinMax Path to G_t

After finding the MinMax path for $\langle x, y \rangle$, we insert the path to G_t. The following operations are performed.

- Convert each virtual link on the MinMax path to a physical link.
- For each firewall f on the MinMax path, increase the size of its rule set by $r(x, y) + r(y, x)$.
- Let the path be $(v_1, v_2, ..., v_l)$, where $v_1 = x$ and $v_l = y$. For $1 \leq i < l$, add a routing entry at v_i for each destination, v_{i+1} , ..., v_l, with the next hop being v_{i+1}. For $1 < i \leq l$, add a routing entry at v_i for each destination, v_1, ..., v_{i-1}, with the next hop being v_{i-1}. This will keep the routing symmetry during the execution of the HAF algorithm.

2.5.5 Ensuring Connectivity

G_t may not be a connected graph. A *component* of G_t is a connected subgraph that is not contained by a larger connected subgraph. Let c be the number of components in G^t.

Let ϕ be the total number of free interfaces of all firewalls. Note that a physical link can be inserted into the graph for each free network interface.

Property 2: G_t can be turned into a connected graph if and only if $\phi \geq c - 1$.

Proof: We first prove that $\phi \geq c - 1$ is a necessary condition for G_t to be turned into a connected graph. G_t has c components. For G_t to be turned into a connected graph, we must reduce c to one by adding at least $c - 1$ new links, which means there must be at least $c - 1$ free interfaces.

Next we prove $\phi \geq c - 1$ is a sufficient condition. First, we consider a simple case where all domains belong to one component. The remaining components must be single firewalls, each having at least two free interfaces. To form a connected graph, we can simply connect these firewalls to any domains.

Second, consider the case where the domains belong to at least two components. $\phi \geq c - 1 \geq 1$. There must be a firewall with a free interface. The firewall belongs to a component. There must be another component that has a domain. Connect the firewall and the domain, which uses one free interface and reduces the number of components by one. Therefore, the condition $\phi \geq c - 1$ remains true. Repeat the above process until all domains belong to one component. For this case, we have already proved that the graph can be made connected.

Therefore, $\phi \geq c - 1$ is a necessary and sufficient condition for G_t to be turned into a connected graph. ∇

Only the first and last links on the MinMax path may be virtual links. By inserting the MinMax path to G_t, we consume at most two free network interfaces. However, if the number of free interfaces is limited, the MinMax may be restricted to consume less than two free interfaces in order to leave enough free interfaces to ensure the connectivity of the graph. Assume the condition $\phi \geq c - 1$ holds in G_t before the insertion of the MinMax path. We want to keep the condition true after the insertion. There are three cases.

- **Case 1: $\phi \geq c+1$ before insertion.** The MinMax path is allowed to consume two free interfaces.
- **Case 2: $\phi = c$ before insertion.** The MinMax path is allowed to consume one free interface or two free interfaces if the path connects two components into one.
- **Case 3: $\phi = c-1$ before insertion.** The MinMax path is allowed to consume one free interface if the path connects two components into one, or consume two free interfaces if the path connects three components into one.

In order to enforce the above restrictions, we have to carefully redesign the subroutine of Insert_Optimal_Path, which is Line 3 of the HAF algorithm. Let $Com(x)$ be the component in G_t that contains x. The pseudo code of Insert_Optimal_Path is given in Fig. 2-9. We give a brief explanation below.

Lines 1-5 implement Case 1. There are plenty of free interfaces. For each firewall f with a free interface, the algorithm adds a virtual link between f and x (or y) if they are not already connected. It then runs the HAF_Dijkstra algorithm on the augmented graph to find the shortest path.

Lines 6-21 implement Case 2. Lines 7-12 add virtual links that connect different components. More specifically, for each firewall f with a free interface, if f and x (or y) belong to different components in G_t, add a virtual link between f and x (or y). When any one of these links is turned into a physical one, it consumes one free interface and also reduces the number of components by one.

Because $\phi = c$ in G_t, we are allowed to consume one free interface without reducing the number of components. In other words, the MinMax path is allowed to use a virtual link within the component that contains x, or a virtual link within the component that contains y, but not both. Lines 13-16 find the shortest path that may use a virtual link within the component of x's. Lines 17-20 find the shortest path that may use a virtual link within the component of y's. Line 21 returns the better of the two paths.

Lines 22-29 implement Case 3. Because $\phi = c - 1$ in G_t, we can use a virtual link only when it reduces the number of components by one. It means that the augmented graph can only have virtual links that connect different components.

2.5.6 Complexity Analysis

The time complexity of the Shortest_Path subroutine is the same as the complexity of Dijkstra's algorithm, which is $O(e + (n + m)\log(n + m))$. The complexities of the HAF_Dijkstra and Insert_Optimal_Path subroutines are the same as that of Shortest_Path. HAF executes the Insert_Optimal_Path subroutine for at most $O(n^2)$ times. Therefore, the total time complexity is $O(n^2 e + n^2(n + m)\log(n + m))$.

2.5.7 Modifying HAF for FRP, partial FRP, and Weighted FPP/FRP

To solve partial FPP, we simply initialize G_t as the existing partial network topology. To solve FRP, we initialize G_t as the existing network topology and set the number of free interfaces to be zero for all firewalls. For partial FRP, we further initialize the route entries $rt[v, d]$ whose values are known. The rest of HAF remains the same. To solve weighted FPP/FRP, we only need to change the definition of $r(x, y)$ and $w(f)$, while leaving the algorithm intact. Instead of $r(x, y) = |R(x, y)|$ as defined in Section 2.3.2, $r(x, y)$ should now be the sum of the weights of all rules in $R(x, y)$. Instead of $w(f) = \sum_{\langle x,y\rangle \in \Pi(f)} r(x, y)$ as in (2–1), $w(f)$ should now be interpreted as the weight of the rule set at f and defined as

$$w(f) = \frac{\sum_{\langle x,y\rangle \in \Pi(f)} r(x, y)}{\text{the weight of firewall } f}$$

2.6 Simulation

In this section, we evaluate the performance of the HAF algorithm for FPP. The results for FRP and weighted FPP/FRP are omitted due to space limitation. To the best of our knowledge, this is the first work that studies the FPP problem. We do not have existing algorithms to compare with. In our simulations, we implement two simple algorithms, called the *tree topology* algorithm (TREE for brevity) and the *full topology* algorithm (FULL for brevity), respectively.

For a given FPP problem, the TREE algorithm first constructs a tree topology, which defines unique routing paths between any two domains. To construct a tree, the algorithm begins with one domain as the root. A number of firewalls are selected to be the children of the root at the second level of the tree. We select firewalls in the descending order of their numbers of interfaces. For each second-level firewall, a number of domains are selected to be the children at the third level. We repeat this until the tree includes all domains or firewalls. The even levels of the tree are firewalls while the odd levels are domains. The number of children of a firewall is limited by its number of network interfaces. We also restrict the average number of children per domain to be the same as the average number of children per firewall.

The FULL algorithm first constructs a tree topology in the same way as the TREE algorithm does. It then *fully* utilizes all remaining free interfaces on the firewalls by making a link from each free interface to an arbitrary domain. After that, we run a shortest-path algorithm to find the least-hops routing path between each pair of domains. The tree topology with cross-links are often seen in organizations with hierarchical administrative structures.

The default simulation parameters are shown in Table 2-2. The simulations will change the default values of the parameters one at a time. n is the number of domains. m is the number of firewalls. Let $e(f)$ be the number of network interfaces on firewall f. $\overline{e(f)}$ is the average number of network interfaces per firewall. The value of $e(f)$, $\forall f \in M$, is generated from $[2..2\overline{e(f)} - 2]$ uniformly at random. $\overline{r(x, y)}$ is the average value of $r(x, y)$ among domain pairs $\langle x, y \rangle$ with $r(x, y) > 0$. p is the probability of $r(x, y) + r(y, x) > 0$ for an arbitrary domain pair $\langle x, y \rangle$. When $r(x, y) > 0$, its actual value is generated from $[1..2\overline{r(x, y)} - 1]$ uniformly at random.

Fig. 2-10-2-15 show the simulation results. In all figures, the y axis is the size of the maximum rule set ($max_{f \in M}\{w(f)\}$) at any firewall. We abbreviate “the size of the maximum firewall rule set” as “the MFRS size”. The x axis is one of the parameters.

The figures compare the MFRS sizes achieved by the three algorithms under different parameter values.

In Fig. 2-10, we vary the number n of domains in the simulation. When n is very small, the numbers of firewalls and interfaces are relatively plentiful such that most domain pairs are one firewall away from each other and the rules are well spread on the firewalls. The MFRS size is small for all three algorithms. As n increases, HAF performs far better than others. When $n = 120$, the MFRS size achieved by HAF is just 35.06% of that achieved by FULL, and 24.78% of that achieved by TREE.

In Fig. 2-11, we vary the number m of firewalls in the simulation. TREE is insensitive to the value of m because the tree topology can not take full advantage of the increased number of firewalls. HAF performs much better than TREE and FULL. When $m = 35$, the MFRS size achieved by HAF is 35.31% of that achieved by FULL, and 24.90% of that achieved by TREE.

In Fig. 2-12, we vary the average number $\overline{e(f)}$ of network interfaces per firewall. HAF performs best among the three. When $\overline{e(f)} = 3.5$, the MFRS size achieved by HAF is 43.02% of that achieved by FULL, and 31.34% of that achieved by TREE.

In Fig. 2-13, we vary the average number $\overline{r(x, y)}$ of rules per domain pair. As $\overline{r(x, y)}$ increases, the MFRS size increases proportionally for all three algorithms. When $\overline{r(x, y)} = 100$, the MFRS size achieved by HAF is 37.05% of that achieved by FULL, and 18.74% of that achieved by TREE.

In Fig. 2-14, we vary the probability p for a domain pair $\langle x, y \rangle$ to have one or more rules. The value of p determines the density of the rule graph G_r. As p increases, the MFRS size increases for all three algorithms. HAF performs better than the other two algorithms for all p values used in the simulation. When $p = 1$, the MFRS size achieved by HAF is 37.37% of that achieved by FULL, and 18.19% of that achieved by TREE.

In Fig. 2-15, we study sparse network topologies with $m = (n - 1)/(\overline{e(f)} - 1)$, which means the number of firewalls is just enough to keep the topology connected. During the

simulation, we discard the runs that have too few firewalls to form a connected topology. The figure shows that HAF works far better than others as n increases. When $n = 120$, the MFRS size achieved by HAF is 34.98% of that achieved by FULL, and 24.57% of that achieved by TREE.

Table 2-1. Frequently-used notations

N	the set of domains
n	the number of domains, i.e., $n = \|N\|$
M	the set of firewalls
m	the number of firewalls, i.e., $m = \|M\|$
e	the total number of network interfaces of all firewalls
$r(x, y)$	the number of access control rules for flows from domain x to domain y
$w(f)$	the number of access control rules to be enforced on a firewall f

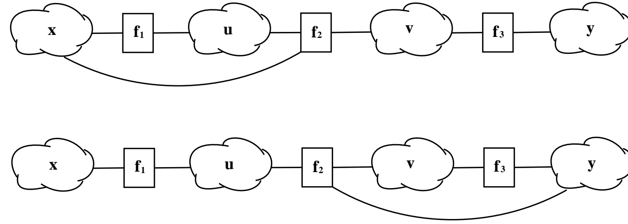


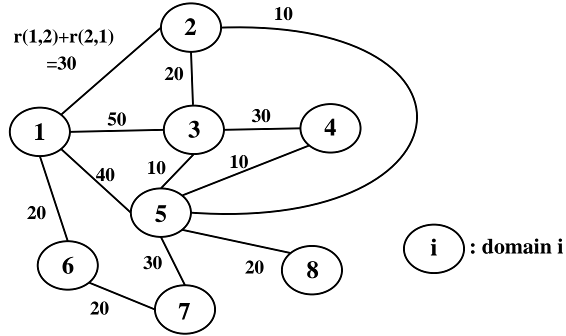
Figure 2-1. Two topologies that connect domains, x, u, v and y, via firewalls, f_1, f_2 and f_3, whose numbers of interfaces are 2, 3 and 2, respectively.

Table 2-2. Default simulation parameters

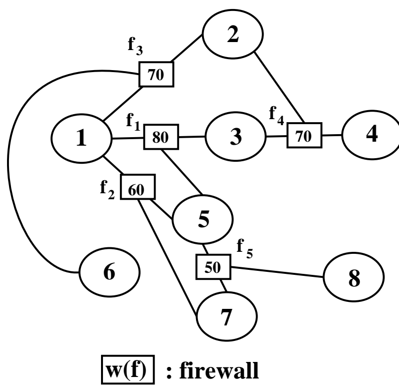
n	m	$\overline{e(f)}$	$\overline{r(x, y)}$	p
100	40	4	10	0.7

x \ y	1	2	3	4	5	6	7	8
1	0	11	40	0	11	10	0	0
2	19	0	17	0	8	0	0	0
3	10	3	0	16	5	0	0	0
4	0	0	14	0	8	0	0	0
5	29	2	5	2	0	0	12	6
6	10	0	0	0	0	0	1	0
7	0	0	0	0	18	19	0	0
8	0	0	0	0	14	0	0	0

(a) r(x,y)



(b) Rule graph G_r



routing tables of domains

rt(1,2)=f_3
rt(1,3)=f_1
rt(1,5)=f_2
rt(1,6)=f_3
rt(1,7)=f_2

rt(2,1)=f_3
rt(2,3)=f_4
rt(2,5)=f_4

rt(3,1)=f_1
rt(3,4)=f_4
rt(3,2)=f_4
rt(3,5)=f_1

rt(4,3)=f_4
rt(4,5)=f_4

rt(5,1)=f_2
rt(5,2)=f_1
rt(5,3)=f_1
rt(5,4)=f_1
rt(5,7)=f_5
rt(5,8)=f_5

rt(6,1)=f_3
rt(6,7)=f_3

rt(7,6)=f_2

rt(8,5)=f_5

routing tables of firewalls

rt(f_1,1)=1
rt(f_1,2)=3
rt(f_1,3)=3
rt(f_1,4)=3
rt(f_1,5)=5

rt(f_2,1)=1
rt(f_2,5)=5
rt(f_2,6)=1
rt(f_2,7)=7

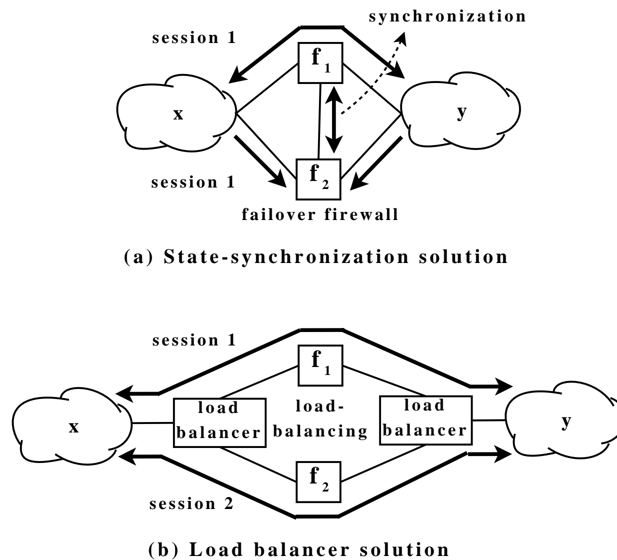
rt(f_3,1)=1
rt(f_3,2)=2
rt(f_3,6)=6
rt(f_3,7)=1

rt(f_4,2)=2
rt(f_4,3)=3
rt(f_4,4)=4
rt(f_4,5)=3

rt(f_5,5)=5
rt(f_5,7)=7
rt(f_5,8)=8

(c) Topology graph G_t

Figure 2-2. Rule matrix, rule graph, and topology graph



(a) State-synchronization solution

(b) Load balancer solution

Figure 2-3. High-availability solutions

```
HAF(G_r, G_t, M)
1.  for each ⟨x, y⟩ of G_r in descending order of (r(x, y) + r(y, x)) do
2.      Insert_Optimal_Path(G_t, x, y)
3.  return G_t
```

Figure 2-4. Pseudo code of HAF

Routing_Condition(v, u, d)

1. **if** $rt[v, d] = \textsc{nil}$ **or** $rt[v, d] = u$ **then**
2.   **return** true
3. **else**
4.   **return** false

Interface_Condition(v, u, s, d)

1. **if** $v = s \wedge next[u, d] = d$ **and** both (s, u) and (u, d) are virtual links but u has only one free interface **then**
2.   **return** false
3. **else**
4.   **return** true

Relax(v, u, d)

1. **if** $\max\{c[v], cost[u, d]\} < cost[v, d]$ **or** $\max\{c[v], cost[u, d]\} = cost[v, d] \wedge hops[u, d] + 1 < hops[v, d]$ **then**
2. $cost[v, d] := \max\{c[v], cost[u, d]\}$
3. $hops[v, d] := hops[u, d] + 1$
4. $next[v, d] := u$

Shortest_Path($G_t^{\langle x,y \rangle}, s, d$)

1. **for** each node $v \in N \bigcup M$ **do**
2. $cost[v, d] := \infty$, $hops[v, d] := \infty$, $next[v, d] := \textsc{nil}$
3. $cost[d, d] := 0$, $hops[d, d] = 0$
4. $Q := N \bigcup M$
5. **while** $Q \neq \emptyset$ **do**
6. $u :=$ Extract_Min(Q)
7.   **if** $u = s$ **then**
8.   break out of the while loop
9. $Q := Q - \{u\}$
10.   **for** every adjacent node v of u in $G_t^{\langle x,y \rangle}$ **do**
11.   **if** Routing_Condition(v, u, d) **and** Interface_Condition(v, u, s, d) **then**
12.   Relax(v, u,d)
13. **return** the shortest path from s to d stored in the *next* variable

HAF_Dijkstra($G_t^{\langle x,y \rangle}, x, y$)

1. $p_1 :=$ Shortest_Path($G_t^{\langle x,y \rangle}, x, y$)
2. $p_2 :=$ Shortest_Path($G_t^{\langle x,y \rangle}, y, x$)
3. **return** the better one between p_1 and p_2

Figure 2-5. Pseudo code of HAF_Dijkstra

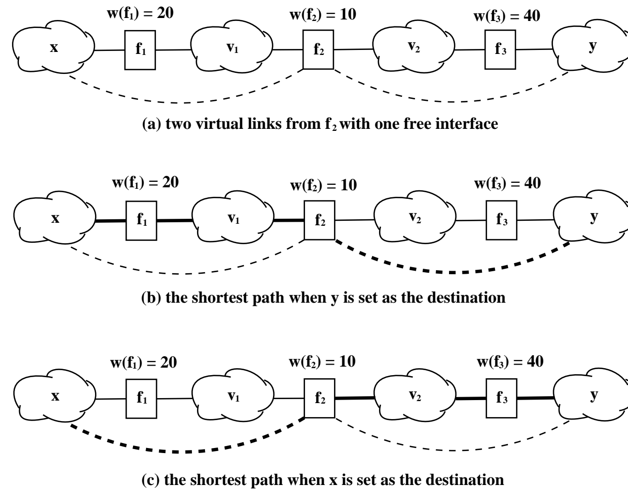


Figure 2-6. (a) Augmented graph $G_t^{\langle x,y\rangle}$, where f_2 has one free interface and two virtual links; (b) Shortest path returned by Shortest_Path($G_t^{\langle x,y\rangle}, x, y$), where the relaxation is performed from y along the path to x; (c) Shortest path returned by Shortest_Path($G_t^{\langle x,y\rangle}, y, x$), where the relaxation is performed from x along the path to y. The best path is (x, f_1, v_1, f_2, y).

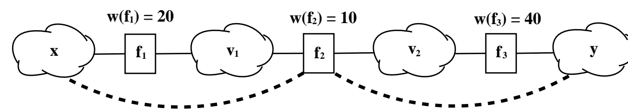


Figure 2-7. Shortest path when f_2 has two free interfaces.

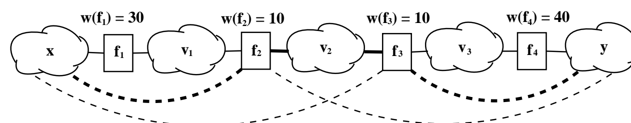


Figure 2-8. Shortest path when f_2 and f_3 each have one free interface.

Insert_Optimal_Path(G_t, x, y)

1. **if** $\phi \geq c + 1$ in G_t **then**
2.   initialize $G_t^{\langle x,y \rangle}$ to be G_t
3.   **for** each firewall f with a free interface **do**
4.   add a virtual link in $G_t^{\langle x,y \rangle}$ between f and x (or y) if they are not already connected
5. $p :=$ HAF_Dijkstra($G_t^{\langle x,y \rangle}, x, y$)
6. **else if** $\phi = c$ in G_t **then**
7.   initialize G'_t to be G_t
8.   **if** $Com(x) \neq Com(y)$ **then**
9.   **for** each firewall f with a free interface, $Com(f) \neq Com(x)$ **do**
10.   add a virtual link in G'_t between f and x
11.   **for** each firewall f with a free interface, $Com(f) \neq Com(y)$ **do**
12.   add a virtual link in G'_t between f and y
13.   initialize $G_t^{\langle x,y \rangle}$ to be G'_t
14.   **for** each firewall f with a free interface, $Com(f) = Com(x)$ **do**
15.   add a virtual link in $G_t^{\langle x,y \rangle}$ between f and x
16. $p_1 :=$ HAF_Dijkstra($G_t^{\langle x,y \rangle}, x, y$)
17.   initialize $G_t^{\langle x,y \rangle}$ to be G'_t
18.   **for** each firewall f with a free interface, $Com(f) = Com(y)$ **do**
19.   add a virtual link in $G_t^{\langle x,y \rangle}$ between f and y
20. $p_2 :=$ HAF_Dijkstra($G_t^{\langle x,y \rangle}, x, y$)
21. $p :=$ the better one between p_1 and p_2
22. **else if** $\phi = c - 1$ in G_t **then**
23.   initialize $G_t^{\langle x,y \rangle}$ to be G_t
24.   **if** $Com(x) \neq Com(y)$ **then**
25.   **for** each firewall f with a free interface, $Com(f) \neq Com(x)$ **do**
26.   add a virtual link in $G_t^{\langle x,y \rangle}$ between f and x
27.   **for** each firewall f with a free interface, $Com(f) \neq Com(y)$ **do**
28.   add a virtual link in $G_t^{\langle x,y \rangle}$ between f and y
29. $p :=$ HAF_Dijkstra($G_t^{\langle x,y \rangle}, x, y$)
30. Insert p to G_t

Figure 2-9. Pseudo code of Insert_Optimal_Path

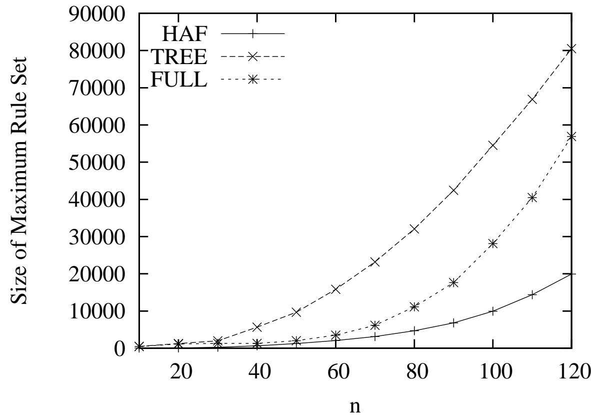


Figure 2-10. Size of maximum rule set with respect to number n of domains. $10 \leq n \leq 120$, $m = 40$, $\overline{e(f)} = 4$, $\overline{r(i,j)} = 10$, $p = 0.7$.

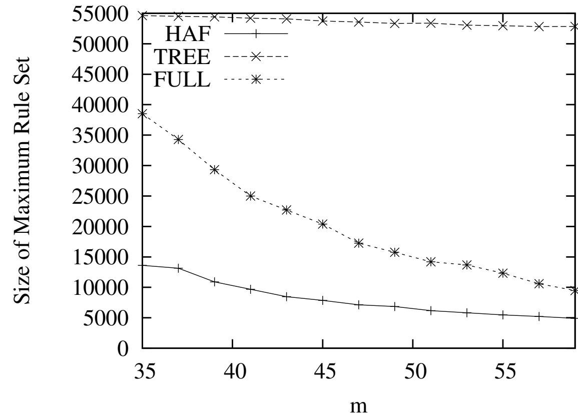


Figure 2-11. Size of maximum rule set with respect to number m of firewalls. $n = 100$, $35 \leq m \leq 59$, $\overline{e(f)} = 4$, $\overline{r(i,j)} = 10$, $p = 0.7$.

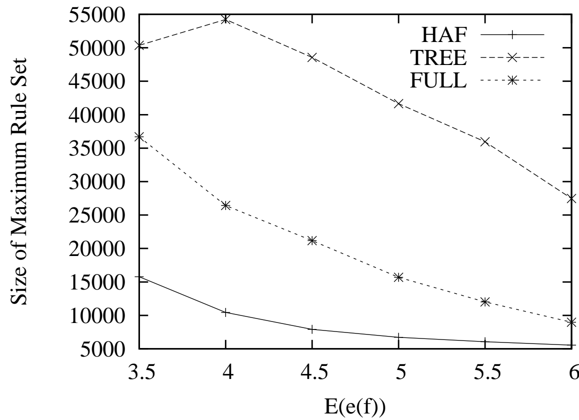


Figure 2-12. Size of max rule set with respect to avg number $\overline{e(f)}$ of network interfaces per firewall. $n = 100$, $m = 40$, $3.5 \leq \overline{e(f)} \leq 6$, $\overline{r(i,j)} = 10$, $p = 0.7$.

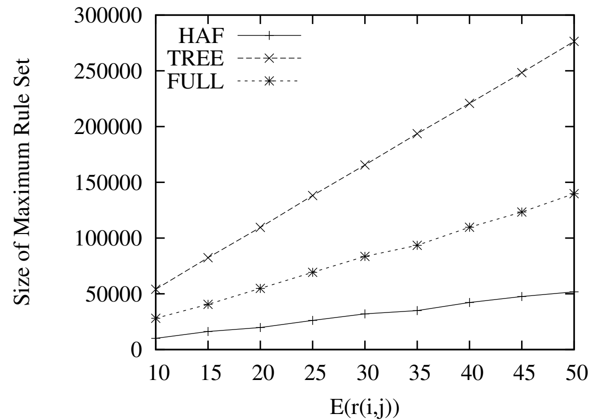


Figure 2-13. Size of maximum rule set with respect to avg number $\overline{r(i,j)}$ of rules per domain pair. $n = 100$, $m = 40$, $\overline{e(f)} = 4$, $10 \leq \overline{r(i,j)} \leq 50$, $p = 0.7$.

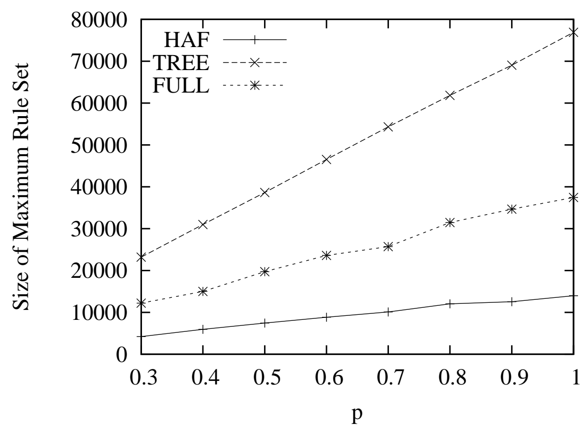


Figure 2-14. Size of maximum rule set with respect to probability p. $n = 100$, $m = 40$, $\overline{e(f)} = 4$, $\overline{r(i,j)} = 10$, $0.3 \leq p \leq 1.0$.

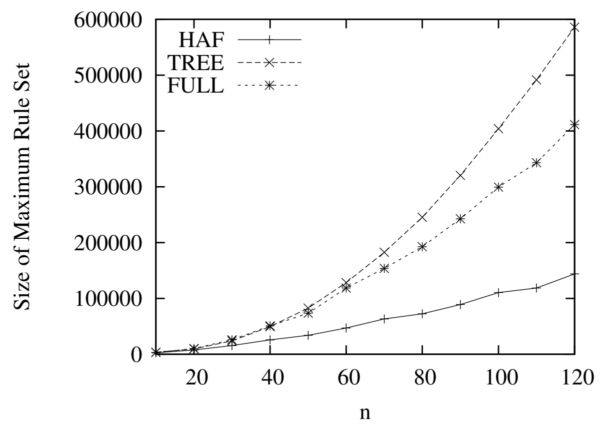


Figure 2-15. Size of maximum rule set in sparse network. $10 \leq n \leq 120$, $m = (n-1)/(\overline{e(f)} - 1)$, $\overline{e(f)} = 4$, $\overline{r(i,j)} = 50$, $p = 1.0$.

CHAPTER 3
A NOVEL INCREMENTALLY-DEPLOYABLE PATH ADDRESS SCHEME FOR THE INTERNET

The research community has proposed numerous network security solutions, each dealing with a specific problem such as address spoofing, DoS attacks, DoQ attacks, reflection attacks, viruses, or worms. However, due to the lack of fundamental support from the Internet, individual solutions often share little common ground in their design, which causes a practical problem: deploying all these vastly-different solutions will add an exceedingly high complexity to the Internet. In this chapter, we propose a simple, generic extension to the Internet, providing a new type of information, called *path addresses*, that simplify the design of security systems for packet filtering, fair resource allocation, packet classification, IP traceback, filter push-back, etc. IP addresses are owned by end hosts; path addresses are owned by the network core, which is beyond the reach of the hosts. We describe how to enhance the Internet protocols for path addresses that meet the uniqueness requirement, completeness requirement, safety requirement, and incrementally-deployable requirement. We evaluate the performance of our scheme both analytically and by simulations.

3.1 Motivation

After thirty years of accumulative development, the Internet is full of security challenges: address spoofing, denial-of-service attacks, denial-of-quality attacks, reflection attacks, viruses, worms, to name a few. The research community has proposed numerous detection/mitigation solutions [26–34], each dealing with a specific problem in its own unique way. The vast solution space, if viewed as a whole, seems able to handle many security problems, but deploying all these solutions can be practically infeasible. A major obstacle is that individual solutions often share little common ground in their design. Many solutions heuristically work around the limitations imposed by the legacy Internet protocols and demand orthogonal changes on routers. Their combined complexity added to the Internet will be exceedingly high. In this dissertation, we take a different angle to

study Internet security. We ask the following question: Can we identify a simple extension to the Internet protocols, which will provide certain new, critical information that is fundamental to solving many security problems? Such information, once being created inside the network, will be made accessible at the network edge, allowing various security applications to be developed. With most complexity remaining at the edge, the network core, which provides application-independent information, can be kept simple and stable.

What is the critical information that the network can provide to assist the development of security applications? There can be many. The one we propose here is called *path addresses*. A host on the Internet is identified by an IP address; a routing path on the Internet will be identified by a path address. The big question is, can path addresses help us in ways that IP addresses can not? Below we use a few examples to illustrate their differences.

In the first example, suppose a server under denial-of-service (DoS) attack attempts to identify the IP addresses of flooding sources and block the packets carrying those addresses. However, this approach will fail if malicious packets carry forged source addresses or a reflection attack is used to cover the true sources. In the second example, imagine a server under denial-of-quality (DoQ) attack tries to distribute its processing capacity fairly among the clients. It cannot perform such distribution based on IP addresses because there are too many of them. A certain kind of aggregation will be necessary. In the last example, suppose a victim has managed to capture an attack packet (say, containing a virus). Based on this single packet, how can the victim trace *across the Internet* back to the attacker, given that the source address in the packet may be a forged one? All above problems cannot be reliably solved based on IP addresses in the packet header, which are set by the sender and may not be genuine. We need address information that is beyond the reach of end hosts. This new address should be set and verified by the routers in the network. If each routing path is assigned a path address, which is carried in the header of packets routed on the path, then a server under DoS attack can block

packets based on path addresses that identify attack paths, a server under DoQ attack can distribute its capacity among packet groups classified based on path addresses, each representing an aggregate of client traffic, and a victim can use path address to find out the attack path and therefore the attack source at the end of the path.

We propose an incrementally-deployable path address scheme (PAS) that meets the following requirements: (1) Each routing path to a certain destination has a unique path address (with very high probability), which is called the *uniqueness requirement*. It ensures that path addresses accurately point out where packets are coming from. Blocking a path address filters out the packets from an attack source without causing significant collateral damage. (2) Each packet carries the address of the path it traverses; the packet has to carry that address from the first router all the way to the destination, which is called the *completeness requirement*. It gives the flexibility of classifying or blocking packets of a given path address anywhere along the path. (3) The path address in a packet's header can only be correctly set by the routers in the network; a host will not be able to forge the path address carried in its packets without being caught, which is called the *safety requirement*. (4) Any viable path address scheme must support incremental deployment on the existing Internet. It should bring benefit when only a portion of routers are upgraded for path addresses, which is called the *incrementally-deployable requirement*. This chapter describes in details how the Internet protocols may be enhanced to include path addresses based on the above requirements. We demonstrate that the proposed PAS scheme satisfies the self-completeness property for incremental deployment. We evaluate the performance of PAS both analytically and by simulations.

The addition of path addresses requires relatively small changes in Internet protocols. On the other hand, it may potentially have a large impact on how security systems will be designed. When a victim's intrusion detection system identifies malicious packets, it may extract the path addresses from the packets and pay special attention to future packets carrying the same path addresses, or even block such packets. If the victim has a mapping

table between path addresses and source IP address prefixes — which can learn from the normal packets received in the past, then it can trace back to the source domain based on the path address of a captured attack packet. Path addresses can also make the pushback mechanism [35] more powerful. After the victim identifies a set of path addresses from malicious packets, it may push these addresses into the network for blocking. When the first-hop router receives a packet from a neighbor router and finds that the packet carries a blocked path address, it drops the packet and then pushes the address to the neighbor. Eventually the addresses will be pushed all the way back to the edge of the domains where the attack hosts reside.

While the focus of this dissertation is on network security, the path address scheme can be used in other network functions, such as packet classification, resource reservation, and service differentiation. For example, instead of per-flow queueing, packet queues can be queued based on path addresses, allowing trunk resource reservation to be made for all flows sharing the same path between two ASes.

The rest of the chapter is organized as follows. Section 3.2 discusses the related work. Section 3.3 presents the design of our path address scheme. Section 4.4 and Section 3.4.2 evaluate PAS by analysis and simulations, respectively.

3.2 Related Work

The research community has proposed numerous solutions to prevent address spoofing, mitigate DoS attacks, filter malicious packets, control access to critical resources, and perform trackback. While tremendous progress has been achieved, most solutions have their own limitations. More importantly, these solutions are designed based on vastly different mechanisms and, as an aggregate, will impose enormous complexity on the Internet. Hence, to make them practically viable, we must seek ways to reduce such complexity with new, generic assistance from the network. We believe path address is a good candidate for achieving this goal. Below we survey the related work, many of which may benefit if path-address information becomes available.

Research on preventing address spoofing has brought a number of technical breakthroughs. Cryptographic cookies [36] allow a server to stay stateless until the address of a client is verified. Examples are SYN cookies [27] and http redirection cookies [37]. One problem is that it is more expensive for the server to generate/verify cookies than the attacker to forge request packets. The client-puzzle solutions [28, 38–40] require clients to solve cryptographic puzzles before their connections are established. However, significant computation overhead is placed not only on malicious hosts but also on legitimate clients. The route-based packet filtering scheme [29] requires each router to drop packets that are not supposed to pass a link. As the paper points out, it is very difficult to reliably determine the set of source-destination pairs whose packets will pass a link. The spoofing prevention method (SPM) [30] requires pairwise secure communication channels among ASes to synchronize their keys. Ingress filtering [31] requires the edge routers of stub networks to inspect outbound packets and discard those packets whose source addresses do not belong to the local networks. This approach requires the participation of all edge routers on the Internet; it does not work well for incremental deployment, which will be elaborated in Section 3.3.6.

Many systems have been proposed to mitigate DoS attacks. SOS [32] is a secure overlay service designed to protect emergency services from DoS attacks. Mayday [41] is a generalization of SOS. They both assume a closed group of trusted clients. WebSOS [42] applies the SOS architecture to the web service using graphic turing tests [43]. CenterTrack [44] is an IP overlay formed among a mesh of special tracking routers to trace the flooding sources.

In recent years, the *IP traceback* problem is intensively studied [26, 33, 45–47]. The goal is to find the origins of the packets with spoofed source addresses. Many traceback schemes incur considerable computation overhead [26], storage overhead [47], or communication overhead [46] in order to keep track of the routers that the packets

traverse. Moreover, they identify the attack paths but do not provide a means for the victim to block the attack packets.

The most related work is Pi [34, 45], which requires each router to insert an n-bit mark in the IP identification field, where n is typically 2. The marks inserted by Pi in the packet header is not suitable to serve as path address. Particularly, Pi does not satisfy the uniqueness, completeness and safety requirements (defined in Section 3.1). First, in Figure 3-1 (a), for packets coming from a long path, marks inserted by remote routers will be overwritten due to the limit size of the path identifier. Consequently, all packets arriving at R_8, even though they may come from different paths further upstream,[1] will have the same path identifier when they reach the receiver. This violates the uniqueness requirement. To block an attacker behind R_9 based on the path identifier, all normal users behind R_9 will also have to be blocked. Second, marks are inserted one at a time by the intermediate routers. Hence, a packet will not carry the same address information in the IP identification field along its route, which violates the completeness requirement. Third, in Figure 3-1 (b), if a router (such as R_8) has more than 2^n links, then there will not be enough mark values to uniquely distinguish where packets are from. On the other hand, if a router has less than 2^n links, it will leave some mark values unused. Fourth, in Figure 3-1 (c), if a zombie host is close to the receiver, only a few bits in the path identifier will be marked by routers, and the rest bits will carry arbitrary values set by the attacker, which violates the safety requirement. To block the attacker, the receiver has to block all path identifiers that carry the same value in those few bits, which means one sixteenth of all normal traffic will be mistakenly blocked in this example. If the zombie is one hop away from the receiver, then one fourth of all normal traffic will have to be

[1] One must imagine a whole tree of upstream routing paths rooted at $R8$, in which $R9$ is just one internal node.

mistakenly blocked. The problem is very serious because a *single* zombie close to the receiver can cause such significant collateral damage.

3.3 Path Address Scheme

This section presents the detailed design of the path address scheme.

3.3.1 Objectives

We are only concerned with the interdomain routing paths at the AS level.[2] Because the discussions are exclusively about interdomain subjects, we will sometimes refer to an "interdomain router" (e.g., a BGP router) simply as a "router" and an "interdomain routing protocol" (e.g., BGP) as a "routing protocol".[3] We will use "AS" and "domain" interchangeably.

We propose a path address scheme (PAS), which assigns each path an address. There is an inherent difference between IP addresses and path addresses. The IP addresses are owned by the hosts, which are given the full responsibility of setting the source addresses in their packets. The path addresses are owned by interdomain routers and kept secret to the hosts. Therefore, only routers are able to set path addresses appropriately in the packet header.

We will answer the following questions: How to define the address of a routing path? How to extend the routing protocols to keep track of the path addresses? What new fields should be introduced in the packet header for path addresses? How can the receiver verify the authenticity of the path address carried in a packet?

A packet carrying the authentic address of its routing path is called a *normal packet*; a packet carrying a false path address is called an *abnormal packet*. Our goal is to enable

[2] Technically, a similar scheme of path addresses may be introduced at the intra-domain level, especially for large ASes.

[3] The legacy names for interdomain router and interdomain routing protocol are exterior gateway and exterior gateway protocol, respectively.

the receiving host, as well as the intermediate routers, to classify the packets into these two categories. To fulfil this goal, the design of path addresses should meet the following objectives.

- **Objective 1:** All legitimate packets will carry the authentic path addresses and therefore be classified as normal packets.
- **Objective 2:** All malicious packets will either carry the authentic path addresses or otherwise be classified as abnormal packets.

The second objective needs more explanation. An attack host may inject malicious packets into a routing path. It has two choices, falsifying or not falsifying path addresses in the packet header. If the attack host sets false path addresses in its packets, the false addresses will be detected and the packets will be classified as abnormal ones. If the attack host lets the router set the authentic path address, all its packets will share a common characteristic: the same path address can be used for traceback or packet filtering.

3.3.2 Definition of Path Address

Each interdomain router generates a random number, called *local number*, which is l bits. The path address of a routing path is defined as the XOR of the local numbers of the routers on the path. An example is given in Figures 3-2 and 3-3, where only the first 8 bits of the local numbers are shown. Figure 3-2 shows the AS-level topology and the local numbers of nine interdomain routers. We denote the local number of a router Rx, $x \in [1..9]$, as $Rx.loc$. Figure 3-3 shows the addresses of the routing paths to $AS1$. We denote the address of the routing path from Rx to a domain y as $Rx.paddr(y)$, which is called the *path address from* Rx *to* y.

In Figure 3-3, the path from $R1$ to $AS1$ contains only one router, $R1$. Hence, $R1.paddr(AS1) = R1.loc = 10101101$. The path from $R2$ to $AS1$ contains two routers, $R2$ and $R1$. Hence, $R2.paddr(AS1) = R1.loc \oplus R2.loc = 10101101 \oplus 00010111 = 10111010$. The path addresses of all other routing paths to $AS1$ are similarly determined.

The local numbers are independent random numbers. Any two (undirected) paths have at least one different router.[4] Their addresses have at least one different local number in the XOR calculation. Therefore, the path addresses are also independent random numbers. Given a destination domain, we want the path addresses from all routers to be different with high probability, as illustrated in Figure 3-3, where $R1.paddr(AS1)$ through $R9.paddr(AS1)$ are all different.

Because the path address will be carried in the packet header, its length represents a performance/overhead tradeoff. Let p be the number of bits in a path address. The probability for two paths to have the same address is $\frac{1}{2^p}$, which diminishes rapidly as p increases. The current number of domains on the Internet is less than 2^{16} because the AS identifier is 16 bits long. Therefore, there are no more than 2^{16} interdomain routing paths to a given destination domain. Suppose a victim server decides to temporarily block a path address carried by identified malicious packets in a SYN-flood attack. The expected number of other domains whose routing paths to the victim happen to have the same address is bounded by 2^{16-p}, and the expected fraction of legitimate packets that are mistakenly blocked is $\frac{2^{16-p}}{2^{16}} = 2^{-p}$, which is only related to the value of p and not related to the number of ASes (or the size of AS identifer). The above two numbers are 2^{-16} and 2^{-32}, respectively, if $p = 32$. We expect the value of p to be set reasonably large.

3.3.3 Extending Routing Protocol for Path Address

The interdomain routing protocol (BGP) establishes a routing table at each router Rx, which has an entry for every reachable domain y. A path address field is added to the routing entry, storing $Rx.paddr(y)$, the address of the current routing path to y. The routing protocol can be easily extended to keep track of the path addresses as the routes

[4] There may be more than one communication link between two neighboring routers. If two paths have the same sequence of routers but use different links, we treat them as the same path.

change. First, the routers nearest to a destination y know the path addresses, which are simply their local numbers. Second, consider an arbitrary router Rx and let Rz be the next hop on its routing path to y. If Rz knows the correct value of $Rz.paddr(y)$, Rx will be able to calculate its path address to y by $Rx.paddr(y) = Rz.paddr(y) \oplus Rx.loc$. Therefore, by induction, the correct addresses for all paths will be found after the routes stabilize, assuming the neighboring routers exchange their knowledge about path addresses together with the classical routing information.[5] To guard against eavesdrop or injection attacks, this exchange may be protected by a secure inter-domain routing protocol such as secure BGPs [48, 49]. We also want to point out that the above distributed computation of path addresses does *not* assume symmetric routing.

The additional overhead (one integer for each routing entry) is very small, comparing with what today's BGP already stores: the whole interdomain path to a destination domain for each routing entry.

Incremental deployment can be achieved as follows: Define a new *optional transitive path attribute* in BGP for path address. For a BGP router that is upgraded to support PAS, when it advertises its routes to neighbors via UPDATE messages, it inserts the new transitive attribute in UPDATE to carry the path address of each route. When a BGP router that does not support PAS receives such UPDATE messages, according to the protocol of BGP [50], it will pass the received transitive attribute (i.e., path addresses) to its neighbors when the received routes are advertised. When a BGP router that supports PAS receives UPDATE messages with the new transitive attribute, it will extract the path addresses, update the routing table for new routes and new path addresses (by XORing the received path addresses with the local number). When it advertises the new routes, it inserts their path addresses as transitive attribute. Under incremental deployment, the

[5] Only the new information in the routing tables, including the path addresses, is exchanged periodically, while the full tables are exchanged in much longer time intervals.

address of a path calculated by the above protocol will be the XOR of the local numbers of the routers that have been upgraded to support PAS. The legacy routers are left out of the distributed calculation process.

A multi-homing domain has more than one path to each destination. Each of its BGP routers stores a different set of routes. The traffic from this domain may be split among those BGP routers, carrying different path addresses and following different routes to a destination. If an attack (such as DoS) is launched from this domain, the victim will identify more than one (most frequently appearing) path address associated with the attack.

BGP stability is of primary importance to the overall stability of the Internet [51]. Route flap damping techniques have been proposed and implemented to stabilize inter-domain routes [52]. Although route change is infrequent among BGP routers overall, it does happen occasionally due to link failure or other reasons. Before new routes are stabilized, some path addresses may be temporarily out-of-sync, causing a burst of packets to be classified as abnormal. Therefore, not all abnormal packets should be dropped automatically. Only when a victim is under attack and the need to immediately block out malicious packets outweights the collateral damage due to the *small possibility* of ongoing inter-domain route change, the victim may decide to block out all abnormal packets. Even when such misblocking happens, it is temporary. We want to point out that route change also poses similar challenge to Pi [45], IP trackback [26, 45–47], and other related work [29].

3.3.4 New Fields in Packet Header and Path Address Verification

PAS places two new fields and one new flag in the packet header: *paddr field*, *verification field*, and *P flag*. The paddr field carries the address of the routing path that the packet traverses. For example, in Figure 3-3, the packets from $AS5$ to $AS1$ will carry 01001010 in the paddr field if they are routed via $R8$. To routers, the P flag indicates whether the paddr field has been appropriately set or not. The purpose of the verification

field is to prevent a malicious host from falsifying the path addresses. In Figure 3-3, a malicious host in $AS4$ may forge packets with 01001010 in the paddr field and pretend that the packets are coming from $AS5$. When the forged packets arrive at $R6$, they are mixed with the legitimate packets from $AS5$, $R6$ must be able to classify the forged ones as abnormal packets. This is accomplished with the help of the verification field. We will explain the actual operations shortly. The problems of where to place these fields and how to operate under incremental deployment will be addressed at the end of this subsection.

The source host does not know the path address of its routing path. It sets the P flag to zero, and sends the packet, which arrives at the first interdomain router. When the router receives a packet, if the P flag is zero, the router knows that it is the first hop on the path and is responsible to assign the appropriate values for the paddr/verification fields. After that is done, the router will change the P flag to one so that the subsequent routers will not change the path address carried in the packet, which satisfies the completeness property.

An example is given in Figure 3-4, where the received packet is shown beside the router. When the first interdomain router, $R8$, receives the packet, it finds that the P flag is zero. $R8$ sets the paddr field to be $R8.paddr(AS1)$, which is the path address from itself to the destination. It sets the verification field to be $R8.paddr(AS1) \oplus R8.loc$, which gives the path address from the next hop router to the destination. Finally it sets the P flag to one before forwarding the packet. When the next-hop router, $R7$, receives the packet, it keeps the path address field intact but updates the verification field by XORing it with the local number. The new value of the verification field is the path address from the yet next hop ($R6$) to the destination. Consequently, each intermediate router Rx is able to verify the authenticity of the path address in the paddr field by matching the received value in the verification field against $Rx.paddr(AS1)$, which can be found in the routing table. If the two matches, then the packet is a normal one. Otherwise, it is classified as

an abnormal one. The verification process must be carried out at each hop because forged packets may be injected anywhere.

When a packet reaches the receiver, the verification field is zero if it is a normal packet and non-zero if it is an abnormal packet. Because the path address should be kept secret from the end host, the last hop router will disguise the path address by performing a keyed hash on the paddr field. All normal packets traversed the same routing path will have the same hash value in the paddr field when reaching the receiver.

So far we have described the normal behavior. Next, we study what a malicious host can do. In Figure 3-5, a malicious host resides in $AS4$. When producing attack packets, it has two choices, either setting the P flag to zero or to one. (1) If it sets the P flag to zero, $R6$ will insert the path address in the packet header. Because the packets carry the correct path address, they will be classified as normal (by definition) all the way to the receiver. When the receiver finds itself is under attack, it tries to mitigate the attack by filtering out the malicious packets. Recall that the last hop router will hash the paddr field. Without knowing the actual path address, the receiver performs filtering based on the hashed path address carried in those packets. (2) To hide itself, a malicious host may set the P flag to one and the paddr field to an arbitrary value. However, it does not know the correct value for the verification field, which must be the path address from $R6$ to the destination. If it sets this value wrong, all intermediate routers will classify the packets as abnormal. An example is given in Figure 3-5, where both paddr and verification fields of an attack packet are initially set to 01001010, representing a false source of $AS5$. The packet is classified as abnormal by all routers on the path.

The new fields can be easily incorporated into IPv6 by adding an extension header. The new fields may also be embedded in the IPv4 header for backward compatibility by creating a new IP option or using the 16 bits from the IP identification field, 1 bit from the flag field, and 13 bits from the offset field, as many other works [26, 45–47] do. The verification field can be made shorter than the paddr field, which provides flexibility of

making performance tradeoff under space constraint. In this case, only a subset of paddr bits are verified.

During incremental deployment, some routers are upgraded to support PAS, while others are not. The former will process the packets as described above. The latter will simply forward the packets without PAS-related operations. Because the path address is the XOR of the local number of the upgraded routers, the verification process will be performed successfully.

3.3.5 Alternative Version of Path Address against Router Compromise

A serious problem arises when the attacker compromises interdomain routers. It is unlikely that this problem will happen frequently because interdomain routers, as critical infrastructure, are closely watched. But if an attacker gains the control of an interdomain router, it can do a variety of harms, such as causing inconsistent routing tables, producing false routes, and injecting forged packets. Our focus is on path address. In Figure 3-5, if the malicious host compromises $R6$, it knows the key and learns the path addresses from $R6$ to all destinations. The malicious host can instruct $R6$ to forge packets with arbitrary values in the paddr field but correct values in the verification field, which allows the packets to pass the verification along the routing paths. Router compromise poses similar challenge to Pi [45], IP trackback [26, 45–47], and other related work [29], even though most did not consider this issue. There is no way one can save the legitimate packets arriving at the compromised $R6$ because $R6$ can corrupt or even drop them. But it is possible for us to enhance the design of path addresses so that packets from $R6$ can be separated from packets forwarded on other paths. The basic intuition is that, if the subsequent routers after $R6$ are not compromised, they should construct a portion of the path address that is beyond the control of $R6$. If necessary, this portion of the address can be used by the receiver to classify the packets from $R6$.

The new way of constructing a path address performs *shifted XOR*, instead of XOR, on the local numbers of the routers. Let the routing path from Rx to a destination domain

y be $Rx \rightarrow R[x-1] ... \rightarrow R1 \rightarrow y$. The distance from Rx to y is x. Let d be a small integer. To calculate $Rx.paddr(y)$, when we XOR the local numbers, we shift $R2.loc$ to the right by d bits, $R3.loc$ to the right by $2d$ bits, ..., and $Rx.loc$ to the right by $(x-1)d$ bits. Hence, the enhanced version of path address is defined as follows.

$$\begin{aligned} Rx.paddr(y) &= \bigoplus_{i \in [1..x]} (Ri.loc \gg (i-1)d) \\ &= R[x-1].paddr(y) \ \oplus \ (Rx.loc \gg (x-1)d) \end{aligned} \quad (3\text{–}1)$$

where $\gg$ is the right shift operator. It is easy for a routing protocol to keep track of the new path address if Rx knows its distance to a destination y and knows the path address from its next hop (denoted as $R[x-1]$) to y. Below we give a few examples based on the local numbers in Figure 3-2. Let $d = 2$.

$$\begin{aligned} R1.paddr(y) &= R1.loc = \mathit{10}101101... \\ R2.paddr(y) &= R1.paddr(y) \oplus (R2.loc \gg 2) \\ &= 10101101... \oplus \mathbf{00}00010111... \\ &= \mathit{1010}1000... \\ R3.paddr(y) &= R2.paddr(y) \oplus (R3.loc \gg 4) \\ &= 10101000... \oplus \mathbf{0000}11010011... \\ &= \mathit{101001}01... \\ &... \end{aligned}$$

where the bold zeros are inserted due to right-shift operations, and the italic bits in a path address will not change in the subsequent computations. The leftmost d bits of a path address are determined by the last hop ($R1$) on the routing path, the next d bits are determined by the last two hops, and so on. If Ri is compromised, it has no impact on the leftmost $(i-1)d$ bits in the path address.

The procedure for setting the values in the paddr/verification fields is similar to what has been described in Section 3.3.4, except that shifted XOR is used. The verification is

however different. When an intermediate router Ri receives the packet, it classifies the packet as normal only if the verification field matches $Ri.paddr(y)$ and the leftmost $(i \times d)$ bits of the paddr field match those in $Ri.paddr(y)$.

If a malicious host compromises Ri, it cannot set the leftmost $(i-1)d$ bits in the paddr field to arbitrary values because they have to match those in $R[i-1].paddr(y)$ in order to pass the verification of the next hop. Consequently, all attack packets from Ri will carry the same $(i-1)d$ bits in the paddr field. A defense system may be designed based on this property. What about the attacker compromises $R1$, the last hop router to the destination? Because the packets from all over the Internet are fully mixed there, it is no longer possible to separate the legitimate packets from the malicious ones if that router is compromised, unless the legitimate packets are protected by end-to-end cryptographic schemes.

Shifted XOR shares superficial similarity with the operation of Pi [45]. In Pi, each router can only set 2 bits in the IP identification field. A malicious host that is one hop away from the victim can arbitrarily set other bits in that field. In shifted XOR, each router has much more impact. For example, the last hop router will influence all bits in the path address. The malicious host one hop away cannot set any bit in the path address without being detected.

3.3.6 Self-Completeness of PAS for Incremental Deployment

During incremental deployment, let C be the set of ASes that have deployed a defense system and C' the set of ASes that have not. The system is said to be *self-complete* if it is fully functional among ASes in C even when attacks are launched from C'. A self-complete system must defeat both the "internal" attackers from C, which are within the defense coverage, and the "external" attackers from C', which are outside of the defense coverage.

Many existing defense systems are not self-complete. Take ingress filtering [31] as an example. Suppose all networks in C perform ingress filtering and those in C' do not.

The attackers from C' can forge any source addresses and pretend to be from C. A victim cannot distinguish such attack packets from legitimate packets from C, and has to drop both. Hence, ingress filtering is not self-complete. SYN-dog [53] is not self-complete by a similar analysis. It can be shown that the IP traceback systems [26, 45–47] are also not self-complete.

When an AS deploys a system that is not self-complete, it essentially takes a *good-citizen* strategy to help in a global effort for defeating a certain network threat. But the benefits for itself arrives only after other organizations on the Internet are also good citizens and, moreover, implementing the same defense. On the contrary, if an AS joins a self-complete system such as PAS, it immediately receives the full defense function for traffic between itself and other ASes that also deployed the system. This has a significant practical impact: Immediate benefit during incremental deployment gives incentive for ASes to deploy such a system.

An AS is *PAS-aware* if it deploys PAS on all its BGP border routers; otherwise, it is *PAS-unaware*. We are not concerned with the uniqueness for the address of a path whose source or destination is PAS-unaware. However, for our scheme to be self-complete, we must ensure that the path address from a PAS-aware source AS to a PAS-aware destination AS (denoted as x) must have a negligibly small probability to be the same as the address from another source AS to x, regardless of whether that source AS is PAS-aware or not. This is generally true, as illustrated in Fig. 3-6, where black circles are routers supporting PAS and white circles are routers not supporting PAS. The path address from $AS3$ to $AS1$ is $R3.paddr(AS1) = R3.loc \oplus R1.loc$. It is different from the path addresses from other domains to $AS1$, with one exception: the address from the PAS-*unaware* $AS6$ to $AS1$ is also $R3.loc \oplus R1.loc$. To solve this problem, when $R3$ receives a packet from an external interface (connecting AS6) with the P flag being zero, it will set the paddr field to be $R3.paddr(AS1) \oplus R3.loc \oplus r = R1.loc \oplus r$, where r is a random number associated with the external interface. It sets the verification field to be the paddr

field XORed with r, which ensures that the verification process will be successful down the path. When the destination $AS1$ receives packets from $AS6$, it will see path address $R1.loc \oplus r$, instead of $R3.loc \oplus R1.loc$.

There is one additional operation. Before a router forwards a packet destined to another AS, if the router's path address to the destination AS is equal to its local number, the router knows that it is the last AS-aware router on the path. In this case, it should disguise the paddr field by hashing before forwarding the packet.

3.4 Evaluation

We evaluate our path address scheme both analytically and by simulations, and compare our scheme with the most related work, Pi [45].

3.4.1 Analysis

Our analytical results reveal some interesting properties of PAS and Pi.

3.4.1.1 Analytical model

We consider an attack involving one malicious host. The case of multiple malicious hosts will be studied by simulations. Suppose the malicious host has already launched an attack, the intrusion detection system at the victim has identified the attack, and it has extracted a path identifier (if the Pi scheme is deployed) or a path address (if the PAS scheme is deployed) from the attack packet. Hoping to block the malicious host, the victim decides to filter all packets carrying the path identifier (or path address). With this scenario, we try to quantify the false-positive probability and the false-negative probability of PAS (and Pi).

When a normal host sends a legitimate packet to the victim, if the packet is mistakenly filtered, we call the event a *false positive*. The probability for that to happen is called *false-positive probability*. Clearly, it is equal to the percentage of all legitimate packets that are filtered, traditionally called *false-positive ratio*.

When the malicious host launches a new attack, if the attack packet is not filtered, we call the event a *false negative*. The probability for that to happen is called *false-negative probability*. It is equal to the percentage of all attack packets that are not filtered, traditionally called *false-negative ratio*.

Let h be the number of routers that have been upgraded to support PAS (or Pi) on the path from the malicious host to the victim. In the analysis, we ignore routers that do not support PAS (or Pi). Let m be the number of bits used to store the path identifier in Pi or the paddr/verification fields and the P flag in PAS. For Pi, let n be the number of bits in any mark inserted to the path identifier by a router. For PAS, let p be the number of bits in the paddr field and v the number of bits in the verification field. $p + v = m - 1$. If v is chosen smaller than p, then only v bits in the paddr field are verified.

3.4.1.2 False-positive probability and false-negative probability of PAS

Suppose the victim has identified a previous attack packet and it begins to block the path address extracted from the packet. The fact that the attack packet was not classified as an abnormal packet means that the path address to be blocked must be the authentic one.

Consider a legitimate packet from a normal host. The design of PAS is completely different from that of Pi. Each router contributes a full p-bit local number to the path address. As long as the routing path from the normal host to the victim has one router that is not in the path from the malicious host to the victim, the addresses of the two paths may differ in any of the p bits. Hence, the chance for these two path addresses to be the same, i.e., the false-positive probability, is

$$FP_{PAS}(p) = \frac{1}{2^p} \qquad (3\text{–}2)$$

Next, consider an attack packet from the malicious host. If the malicious host does not falsify the path address carried in the packet and lets the routers on the path set the address, then the address will be the authentic one, matching the blocked path

address. To produce a false negative, the malicious host has to falsify the values in the paddr/verification fields and set the P flag to be one. By the design of PAS, for any path, there exists only one value for the verification field that can pass the verification process. The chance for a random value in the verification field to pass the verification, i.e., the false-negative probability is

$$FN_{PAS}(v) = \frac{1}{2^v} \tag{3–3}$$

Because $p + v = m - 1$, if m is fixed, we can tune the values of p and v to make tradeoff between the false-positive probability and the false-negative probability. However, we are able to lower both probabilities if m can be increased. If m is large enough (e.g., 30), we can lower both to almost zero. Finally, by (3–2)-(3–3), neither false-positive nor false-negative probabilities depends on the distance h from the malicious host to the normal host.

3.4.1.3 False-positive probability and false-negative probability of Pi

Suppose the victim has identified a previous attack packet and it begins to block the path identifier extracted from the packet.

Consider a legitimate packet from a normal host. Suppose the routing path from the normal host to the victim shares the last c hops with the routing path from the malicious host. The path identifier carried in the legitimate packet must share $c \times n$ common bits with the blocked identifier. The packet will be mistakenly filtered if the other $(m - c \times n)$ bits happen to also have the same value as the blocked identifier. Hence, the false-positive probability is

$$FP_{Pi}(m, n, c) = \begin{cases} \frac{1}{2^{m-n\times c}} & \text{if } n \times c < m \\ 1 & \text{if } n \times c \geq m \end{cases} \tag{3–4}$$

Next, consider an attack packet from the malicious host. This new attack packet follows the same path as the previous one. Hence, the path identifier carried in the packet shares $h \times n$ common bits with the blocked identifier. The other $(m - h \times n)$ bits

are randomly set by the malicious host. The packet will not be filtered if any of those $(m - h \times n)$ bits is different from the one in the blocked identifier. Hence, the false-negative probability is

$$FN_{Pi}(m, n, h) = \begin{cases} 1 - \frac{1}{2^{m-n\times h}} & \text{if } n \times h < m \\ 0 & \text{if } n \times h \geq m \end{cases} \tag{3–5}$$

Pi has two design parameters, m and n. (3–4)-(3–5) reveal some interesting properties of Pi. First, we show a counter-intuitive result. It is not true that a larger size for path identifier always leads to better performance. From (3–4)-(3–5), increasing m reduces the false-positive probability, but increases the false-negative probability when $n \times h < m$. Decreasing m has the opposite effect. Second, increasing n reduces the false-negative probability, but increases the false-positive probability. Decreasing n has the opposite effect. Third, the false-negative probability increases exponentially as the distance h from the malicious host to the victim decreases.

We can tune the values of m and n to make tradeoff between the false-positive probability and the false-negative probability, but we cannot simultaneously drive them down. Another way to reduce the false-negative probability is to add a path identifier to the blocked list whenever the intrusion detection system identifies an attack packet that is not filtered. However, as the number of blocked path identifiers is increased, the chance for a normal packet to match one of them, i.e., the false-positive probability, will also be increased.

3.4.2 Simulations

We use simulations to evaluate PAS and compare it with Pi in terms of false-positive ratio and false-negative ratio.

3.4.2.1 Simulation setup

The simulation network has 10,000 nodes (ASes). The nodes are inter-connected through their gateways (inter-domain routers). A node is said to be *normal* if it does not

have an attack host; it is *malicious* if it does. A certain number of nodes are randomly selected to be malicious. The *attacker ratio* is defined as the number of malicious nodes divided by the total number of nodes (which is 10,000). The default attacker ratio is 0.1, but we will vary it in the simulation. A single victim is randomly selected from the network. The network topology is generated based on the Power-Law Internet model [54].

The attack model used in our simulations is similar to that in [45]. There are two phases. The first is called the *learning phase*, and the second is called the *attack phase*. In the learning phase, we assume that an intrusion detection system identifies the attack packets and extracts the path identifiers or path addresses (e.g., the most-frequently-received ones under a DoS attack) for blocking. How to design an intrusion detection system in general is beyond the scope of this chapter. Suppose, after this phase, the victim learns the path address from each malicious node to the victim, or if Pi is used, it learns up to r path identifiers for each malicious node. In Figure 3-1 (c), we have showed that there can be multiple path identifiers from an attacker if it resides near the victim. The default value for r is one, but we will vary it in the simulation.

In the attack phase, the victim filters all packets carrying the path identifers or path addresses learned in the previous phase. For Pi, the malicious nodes generate attack packets with random initial values in the path-identifier field. For PAS, the malicious nodes generate attack packets with random initial values in the paddr/verification fields and one for the P flag. We measure the *false-positive ratio*, which is the fraction of legitimate packets from all sources that are mistakenly filtered, and the *false-negative ratio*, which is the fraction of attack packets that are not filtered in this phase.

Since Pi uses no more than 30 bits in the packet header for its path identifier, including 16 bits from the IP identification field, 1 bit from the flag field, and 13 bits from the offset field. We allow the same number of bits for PAS in our simulations for fair comparison. PAS uses 16 bits for the paddr field, 1 bit for the P flag, and 13 for the verification field. Pi uses all 30 bits for the path identifier. We have learned from

Section 4.4 that, in Pi, the number n of bits for a mark represents a tradeoff between the false-positive ratio and the false negative ratio. We let n be 2, 3, 5, or 6 in the simulations. Pi with these n values are denoted as Pi(2), Pi(3), Pi(5) and Pi(6), respectively.

3.4.2.2 Performance evaluation with respect to attacker ratio

In the first simulation, we vary the attacker ratio from 0.05 to 0.3. That means the number of malicious nodes ranges from 500 to 3,000. Figure 3-7shows the false-positive ratios and the false-negative ratios, respectively. Both false-positive ratio and false-negative ratio of PAS are near zero, varying between 0 and 0.0002. The attacker ratio has hardly any impact on PAS. None of Pi(2)-Pi(6) has a low false-positive ratio and a low false-negative ratio at the same time. The false-positive ratios of Pi(2) and Pi(3) are close to zero, but their false-negative ratios are almost one. Note that the simulation in [45] did not include malicious nodes close to the victim, but the simulation in this dissertation does, which reveals the above serious performance problem. Figure 3-7 also confirms our analytical result in Section 4.4 that increasing n reduces the false-negative ratio of Pi but increases the false-positive ratio. For the same value of n, as the attacker ratio increases, the false-positive ratio increases while the false-negative ratio decreases. The reason is that, the more the attackers, the more the number of blocked path identifiers (identified in the learning phase), the higher the chance of misblocking normal packets, and the lower the chance of not blocking attack packets.

3.4.2.3 Performance evaluation with respect to network topology

In the second simulation, we study the impact of network topology on the performance of PAS and Pi. We change the density in network connectivity by varying the fraction of degree-one nodes from 0.1 to 0.5. Figure 3-8 shows the false-positive ratios and the false-negative ratios, respectively. PAS has very small false-positive and false-negative ratios, ranging between 0.0001 and 0.0002. Topology variation has little impact on its performance. For Pi, however, as the fraction of degree-one nodes increases, the false-positive ratios increase while the false-negative ratios decrease. The reason is that

increasing degree-one nodes reduces the number of links in the network, which has two consequences. First, it increases the lengths (denoted as c) of common subpaths shared by attack packets and normal packets, and hence increases the false-positive ratio due to (3–4) in Section 4.4. Second, it increases the path length (denoted as h) and thus increases the false-negative ratio due to (3–5).

3.4.2.4 Performance comparison with respect to r

In the third simulation, we vary r from 1 to 101. Recall that r is the maximum number of path identifiers per malicious node learned in the first phase. Figure 3-9 shows the false-positive ratios and the false-negative ratios, respectively. It is a parameter for Pi, and thus has no impact on PAS. The larger the value of r, the larger the number of blocked path identifiers in the second phase, the higher the chance of misblocking normal packets, and the lower the chance of not blocking attack packets. Therefore, as r increases, the false-positive ratio of Pi increases and the false-negative ratio decreases.

We also run the above simulations on shifted XOR. Its performance is very close but slight worse than PAS. We do not plot it in the figures because it almost completely overlaps with the curve of PAS except for the left plot of Figure 3-9, where the false-positive ratio of shifted XOR would be half a percentage higher than that of PAS.

3.4.2.5 Performance evaluation under incremental deployment

Let the deployment ratio be the fraction of all routers that support PAS (or Pi). In the last simulation, we vary the deployment ratio from 0.1 to 1.0, and randomly select ASes to be PAS/Pi-aware. Figure 3-10 shows the false-positive ratios and the false-negative ratios for packets from all PAS/Pi-aware ASes to a PAS/Pi-aware victim. The false-negative and positive ratios of PAS are near zero due to PAS's self-completeness (Section 3.3.6). To the contrary, the false negative ratio of Pi is very high when the deployment ratio is small.

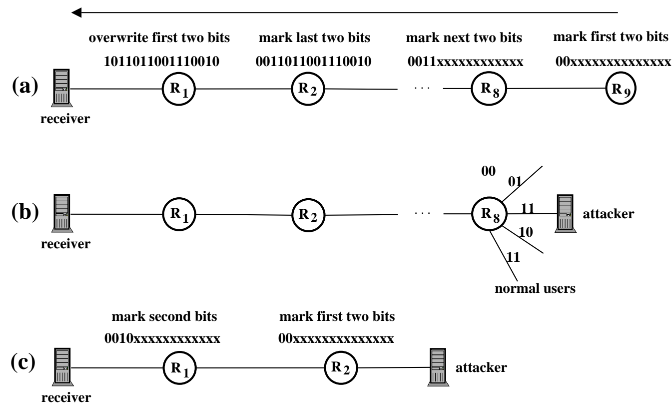


Figure 3-1. Pi cannot be used for path address.

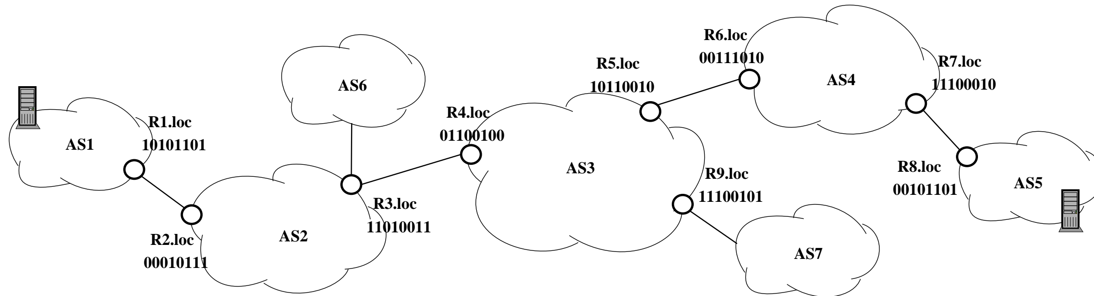


Figure 3-2. Local numbers of the interdomain routers.

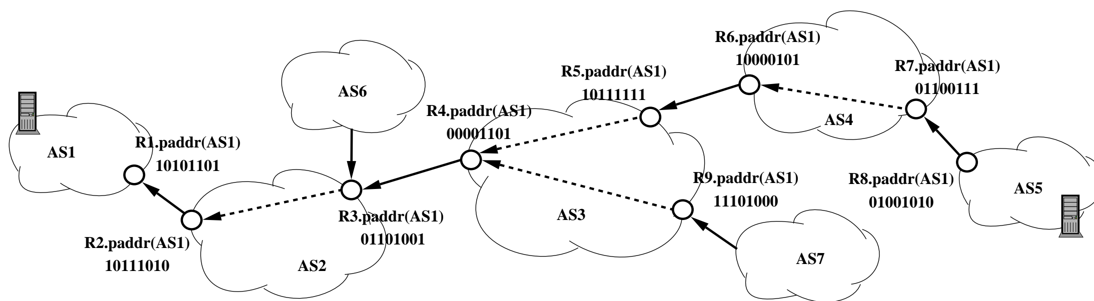


Figure 3-3. Addresses for the routing paths from the routers to $AS1$. For example, $R8.paddr(AS1) = 01001010$. It is the XOR of all local numbers on the routing path $R8 \rightarrow R7 \rightarrow R6 \rightarrow R5 \rightarrow R4 \rightarrow R3 \rightarrow R2 \rightarrow R1$. Alternatively it can be viewed as the XOR of $R8$'s local number and $R7.paddr(AS1)$.

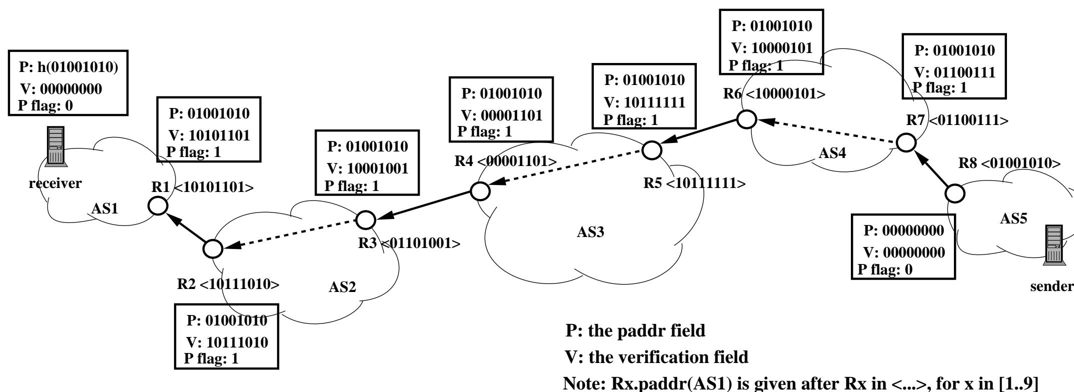


Figure 3-4. Received values of the paddr and verification fields are shown beside each router. The two fields are set to zeros by the sender. The first interdomain router sets these fields with appropriate values. The path address field stays unchanged at the subsequent hops, but the verification field is XORed by the local number at each hop. The verification field should be zero when the packet reaches its receiver.

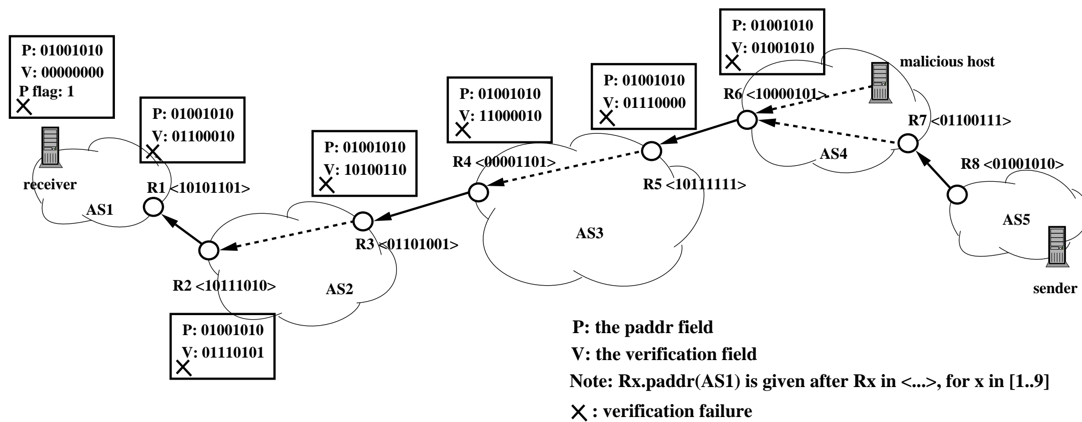


Figure 3-5. Malicious host in $AS4$ sets the paddr/verification fields arbitrarily with the P flag being one. As long as it does not know $R6.paddr(AS1)$, the attack packets to $AS1$ will be classified as abnormal, which is indicated by a cross below V in the figure.

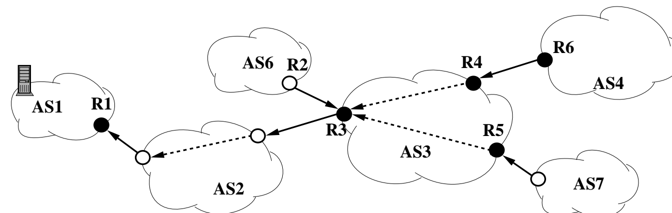


Figure 3-6. Path address between $AS3$ and $AS1$ should be artificially made different from the address between $AS6$ and $AS1$.

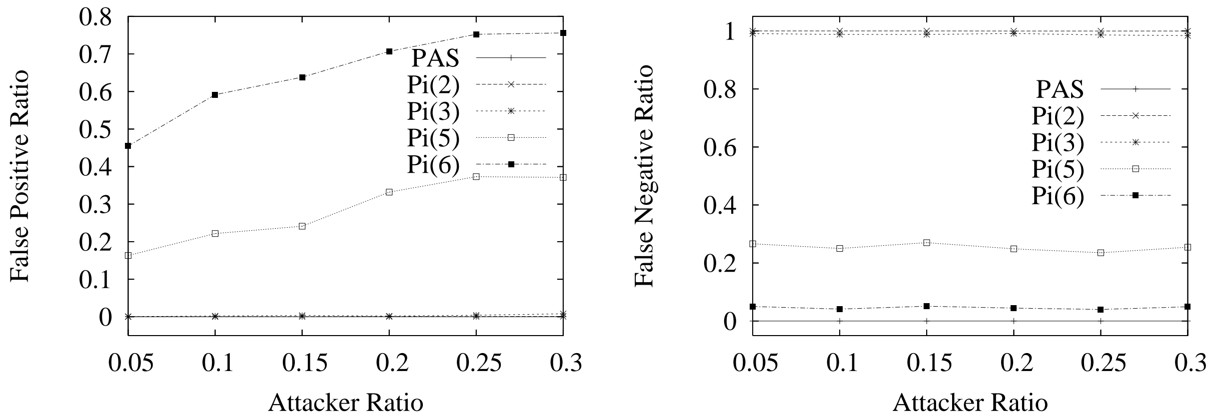


Figure 3-7. *Left:* false positive ratios with respect to attacker ratio. *Right:* false negative ratios with respect to attacker ratio.

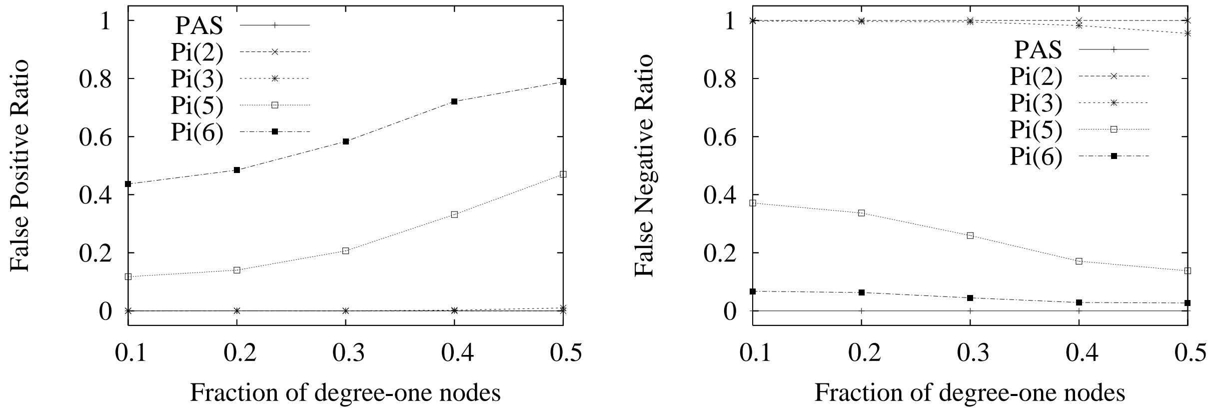


Figure 3-8. *Left:* false positive ratios with respect to fraction of degree-one nodes. *Right:* false negative ratios with respect to fraction of degree-one nodes.

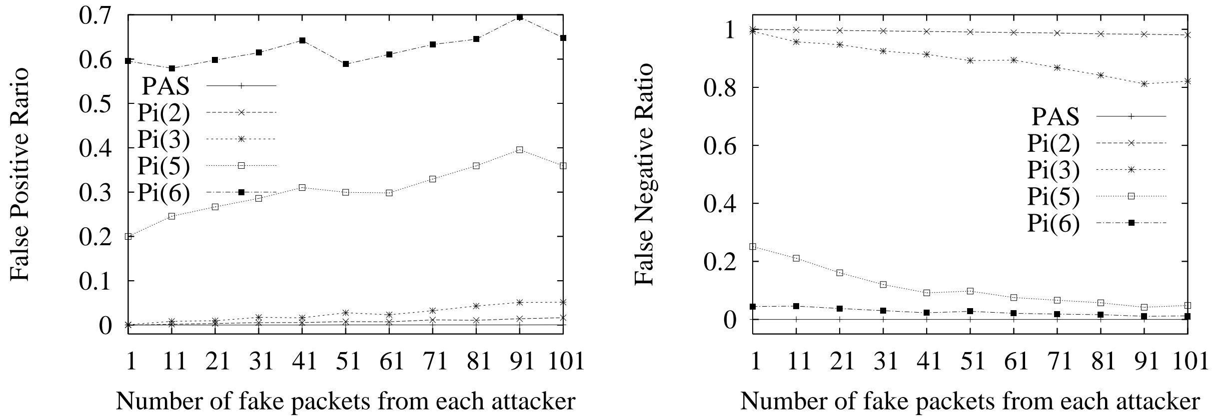


Figure 3-9. *Left:* false positive ratios with respect to r. *Right:* false negative ratios with respect to r.

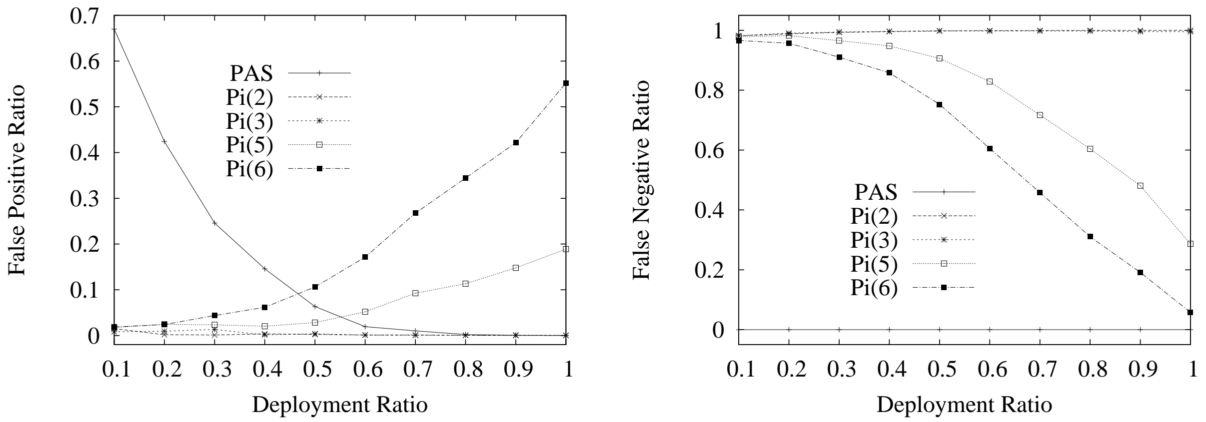


Figure 3-10. *Left:* false positive ratios with respect to deployment ratio. *Right:* false negative ratios with respect to deployment ratio.

CHAPTER 4
FIT A SPREAD ESTIMATOR IN A SMALL MEMORY

The spread of a source host is the number of distinct destinations that it has sent packets to during a measurement period. A spread estimator is a software/hardware module on a router that inspects the arrival packets and estimates the spread of each source. It has important applications in detecting port scans and DDoS attacks, measuring the infection rate of a worm, assisting resource allocation in a server farm, determining popular web contents for caching, to name a few. The main technical challenge is to fit a spread estimator in a fast but small memory (such as SRAM) in order to operate it at the line speed in a high-speed network. In this chapter, we design a new spread estimator that delivers good performance in tight memory space where all existing estimators no longer work. The new estimator not only achieves space compactness but operates more efficiently than the existing ones. Its accuracy and efficiency come from a new method for data storage, called virtual vectors, which allow us to measure and remove the errors in spread estimation. We perform experiments on real Internet traces to verify the effectiveness of the new estimator.

4.1 Motivation

Traffic measurement in high-speed networks has many challenging problems [55–59]. In this chapter, we study the problem of *spread estimation*, which is to estimate the number of distinct destinations to which each source has sent packets that are of all or certain specific types.

We define a *contact* as a pair of source and destination, for which the source has sent a packet to the destination. In the most general terms, the source or destination can be an IP address, a port number, or a combination of them together with other fields in the packet header. The *spread* of a source is the number of distinct destinations contacted by the source during a measurement period. A *spread estimator* is a software/hardware module on a router (or firewall) that inspects the arrival packets and estimates the spread

of each source. It must implement two functions. The first function is to store the contact information extracted from the arrival packets. The second function is to estimate the spread of each source based on the collected information. Although our discussion will focus on the source's spread, we may change the role of source and destination and use the same spread estimator to measure the *spread of a given destination*, which is the number of distinct sources that have contacted the destination.

A spread estimator has many important applications in practice. Intrusion detection systems can use it to detect port scans [60], in which an external host attempts to establish too many connections to different internal hosts or different ports of the same host. It can be used to detect DDoS attacks when too many hosts send traffic to a receiver [61], i.e., the spread of a destination is abnormally high. It can be used to estimate the infection rate of a worm by monitoring how many addresses the infected hosts will each contact over a period of time. A large server farm may use it to estimate the spread of each server (as a destination) in order to assess how popular the server's content is, which provides a guidance for resource allocation. An institutional gateway may use it to monitor outbound traffic and determine the spread of each external web server that has been accessed recently. This information can also be used as an indication of the server's popularity, which helps the local proxy to determine the cache priority of the web content.

The major technical challenge is how to fit a spread estimator in a small high-speed memory. Today's core routers forward most packets on the fast forwarding path between network interfaces that bypasses the CPU and main memory. To keep up with the line speed, it is desirable to operate the spread estimator in fast but expensive, size-limited SRAM [62]. Because many other essential routing/security/performance functions may also run from SRAM, it is expected that the amount of high-speed memory allocated for spread estimation will be small. Moreover, depending on the applications, the measurement period can be long, which requires the estimator to store an enormous number of contacts. For example, to measure the popularity of web servers, the

measurement period is likely to be hours or even days. Hence, it is critical to design the estimator's data structure as compact as possible.

The past research meets the above challenge with several spread estimators [62–64] that process a large number of contacts in an ever smaller space. This dissertation adds a new member that not only requires far less memory than the best known one but also operates much more efficiently. It is able to provide good estimation accuracy in a tight space where all existing estimators fail. Our major contribution is a new methodology for contact storage and spread estimation based on *virtual vectors*, which use the available memory more efficiently for tracking the contacts of different sources.

Do we need a new spread estimator when there are already several? Consider the following scenario. Collected from the main gateway at the University of Florida on a day in 2005, the Internet traffic trace that we used in our experiments has around 10 million distinct contacts from 3.5 million distinct external sources to internal destinations. We expect that today's traffic on the Internet core routers will exceed these figures by far. Now assume the router can only allocate 1MB SRAM for the spread estimator. On average there are only 2.3 bits allocated for tracking the contacts from each source. We classify the existing estimators into several categories based on how they store the contact information: 1) storing per-flow information, such as Snort [65] and FlowScan [61], 2) storing per-source information, such as Bitmap Algorithms [64] and One-level/Two-level Algorithms[1] [63], and 3) mapping sources to the columns of a bit matrix, where each column stores contacts from all sources that are mapped to it, such as On-line Streaming Module (OSM) [62]. Obviously the first two categories will not work here because 2.3 bits

[1] Venkatataman et al. [63] used a probabilistic sampling technique to reduce the number of contacts that are input to the estimator (at the expense of estimation accuracy). Naturally, it also reduces the number of sources appearing in input contacts. This technique can be equally applied to other estimators such as those in [64] and the one to be proposed in this dissertation. When we say per-source state, we refer to sources that appear in the contacts after sampling (if the sampling technique is used).

are not enough to store the contacts of a source. As we will discuss shortly, OSM is also ineffective in this scenario because mapping multiple sources to one column introduces significant unremovable errors in spread estimation. Our new estimator uses a simple one-dimension bit array. A virtual bit vector is constructed from the array for each source. The virtual vectors share bits uniformly at random and introduce uniform errors in spread estimation that can be measured and removed. Based on virtual vectors, a spread estimator is mathematically developed and analyzed. The new estimator requires much less computation and fewer memory accesses than OSM, yet it can work in a very small memory where OSM cannot. Its performance is evaluated by experiments using real Internet traffic traces.

4.2 Existing Spread Estimators

Snort [65] maintains a record for each active connection and a connection counter for each source IP. Keeping per-flow state is too memory-intensive for a high-speed router, particularly when the fast memory allocated to the function of spread estimation is small.

One-Level/Two-Level Algorithms [63] maintain two hash tables. One stores all distinct contacts occurred during the measurement period, including the source and destination addresses of each contact. The other hash table stores the source addresses and a contact counter for each source address. A probabilistic sampling technique is used to reduce the number of contacts to be stored. However, instead of storing the actual source/destination addresses in each sampled contact, one can use bitmaps [64] to save space. Each source is assigned a bitmap where a bit is set for each destination that the source contacts. One can estimate the number of contacts stored in a bitmap based on the number of bits set [64]. An index structure is needed to map a source to its bitmap. It is typically a hash table where each entry stores a source address and a pointer to the corresponding bitmap. However, such a spread estimator cannot fit in a tight space where only a few bits are available for each source — not enough for a bitmap to work appropriately.

If we make each bitmap sufficiently long, we will have to reduce the number of them and there will not be enough bitmaps for all sources. One solution is to share each bitmap among multiple sources. Consider a simple spread estimator that uses a bit matrix whose *columns* are *bitmaps*. Sources are assigned to columns through a hash function. For each contact, the source address is used to locate the column and, through another hash function, the destination address is used to determine a bit in the column to be set. One can estimate the number of contacts stored in a column based on the number of bits set. However, the estimation is for contacts made by *all sources that are assigned to the column*, not for the contacts of a specific source under query.

The information stored for one source in a column is the *noise* for others that are assigned to the same column. One must remove the noise in order to estimate the spread correctly. To solve this problem, OSM (Online Streaming Module) [62] assigns each source randomly to l (typically three) columns through l hash functions, and it sets one bit in each column when storing a contact. A source will share each of its columns with a different set of other sources. Consequently the noise (i.e., the bits set by other sources) in each column will be different. Based on such difference, a method was developed to remove the noise and estimate the spread of the source [62].

OSM also has problems. Not only it increases the overhead by performing $l + 1$ hash operations, making l memory accesses and using l bits for storing each contact, but the noise can be too much to be removed in a *compact* memory space where a significant fraction of all bits (e.g., above 50%) are set. The columns that high-spread sources are assigned to have mostly ones; they are called *dense columns*, which present a high level of noise for other sources.[2] The columns that *only* low-spread sources are assigned to are likely to have mostly zeros; they are called *sparse columns*. As we observe in the experiments, in a tight space, dense columns will account for a significant fraction of all

[2] Note that each high-spread source produces l dense columns.

columns. The probability for a low-spread source to be assigned to l dense columns is not negligible. Since these dense columns will have many bits set at common positions, the difference-based noise removal will not work, and hence the spread estimation will be wrong. We will confirm the above analysis by the experimental results in Section 4.5.

Also related is the detection of stealthy spreaders using online outdegree histograms in [66], which detects the event of collaborative address scan by a large number of sources, each scanning at a low rate. It is able to estimate the number of participating sources and the average scanning rate, but it cannot perform the task of estimating the spread of each individual source in the arrival packets.

4.3 Design of Compact Spread Estimator (CSE)

We first motivate the concept of virtual vectors that are used to store the contact information. We then design our compact spread estimator (CSE). Finally we discuss how to store source addresses.

4.3.1 Motivation for Virtual Vectors

Existing estimators divide the space into bitmaps and then allocate the bitmaps to sources. If we use per-source bitmaps and each bitmap has a sufficient number of bits, then the total memory requirement will be too big. If we share bitmaps, it is hard to remove the noise caused by sources that are assigned to the same bitmap. Resolving this dilemma requires us to look at space allocation from a new angle.

Our solution is to create a *virtual bit vector* for each source by taking bits uniformly at random from a common pool of available bits. In the previous estimators, *two bitmaps do not share any bit.* Two sources either do not cause noise to each other, or cause severe noise when they share a common bitmap — they share all bits in the bitmap. Each source experiences a different level of noise that cannot be predicted. In our estimator, two virtual vectors may share one or more (which is very unlikely) common bits. While each source has its own virtual vector to store its contacts, noise still occurs through the common bit between two vectors. However, there is a very nice property: Because the bits

in virtual vectors are randomly picked, there is an equal probability for any two bits from different vectors to be the same physical bit. The probability for the contacts of one source to cause noise to any other source is the same. When there are a large number of sources, the noise that they cause to each other will be roughly uniform. Such uniform noise can be measured and removed. This property enables us to design a spread estimator for a tight space where the previous estimators will fail. The new estimator is not only far more accurate in spread estimation but also much more efficient in its online operations.

4.3.2 CSE: Storing Contacts in Virtual Vectors

Our compact spread estimator (CSE) consists of two components: one for storing contacts in virtual vectors, and the other for estimating the spread of a source. The first component will be described below, and the second will be in the next subsection.

CSE uses a bit array B of size m, which is initialized to zeros at the beginning of each measurement period. The ith bit in the array is denoted as $B[i]$. We define a *virtual vector* $X(src)$ of size s for each source address src, where $s \ll m$. It consists of s bits pseudo-randomly selected from B.

$$X(src) = (B[H_0(src)], B[H_1(src)], ..., B[H_{s-1}(src)]), \tag{4–1}$$

where H_i, $0 \leq i \leq s-1$, are different hash functions whose range is $[0..m-1]$. They can be generated from a single master hash function H_M.

$$H_i(src) = H_M(src \oplus R[i]) \tag{4–2}$$

where R is an array of s different random numbers and $\oplus$ is the XOR operator.

When a contact (src, dst) is received, CSE sets one bit in B and the location of the bit is determined by both src and dst. More specifically, the source address src is used to identify a virtual vector $X(src)$, and the destination address dst is used to determine a bit location i^* in the virtual vector.

$$i^* = H_M(dst) \mod s \tag{4–3}$$

From equations 4–2 and 4–3, we know that the i^*th bit in vector $X(src)$ is at the following physical location in B:

$$H_{i^*}(src) = H_M(src \oplus R[i^*]) = H_M(src \oplus R[H_M(dst) \bmod s]).$$

Hence, to store the contact (src, dst), CSE performs the following assignment:

$$B[H_M(src \oplus R[H_M(dst) \bmod s])] := 1. \tag{4–4}$$

We stress that setting one bit by equation 4–4 is the only thing that CSE does when storing a contact. It takes two hash operations and one memory access. The source's virtual vector, as defined in equation 4–1, is never explicitly computed until the spread estimation is performed on an offline machine (to be described shortly). The bit, which is physically at location $H_M(src \oplus R[H_M(dst) \mod s])$ in B, is logically considered as a bit at location $(H_M(dst) \mod s)$ in the virtual vector $X(src)$. Note that duplicate contacts will be automatically filtered because they are setting the same bit and hence have no impact on the information stored in B. Multiple different contacts may set the same physical bit. This is embodied in the probabilistic analysis when we derive the spread estimation formula.

4.3.3 CSE: Spread Estimation

At the end of the measurement period, one may query for the spread of a source src (i.e., the number of distinct contacts that src makes in the period). Let k be the actual spread of src. The formula that CSE uses to compute the estimated spread $\hat{k}$ of src is

$$\hat{k} = s \cdot \ln(V_m) - s \cdot \ln(V_s) \tag{4–5}$$

where V_m is the fraction of bits in B whose values are zeros and V_s is the fraction of bits in $X(src)$ whose values are zeros. The value of V_m and V_s can be easily found by counting zeros in B and $X(src)$, respectively. The first item, $(-s \cdot \ln(V_m))$, captures the noise, which is uniformly distributed in B and thus does not change for different sources. The second

item, $(-s \cdot \ln(V_s))$, is the estimated number of contacts that are stored in $X(src)$, including the contacts made by src and the noise.

We expect that queries are performed after B is copied from the router's high-speed memory to an offline computer in order to avoid interfering with the online operations. Below we will derive equation 4–5 mathematically. Its accuracy and variance will be analyzed in the next section.

Some additional notations are given as follows. Let n be the number of distinct contacts from all sources during the measurement period, U_m be the random variable for the number of '0' bits in B, and U_s be the random variable for the number of '0' bits in the virtual vector $X(src)$. Clearly, $V_m = \frac{U_m}{m}$ and $V_s = \frac{U_s}{s}$.

Let A_j be the event that the jth bit in $X(src)$ remains '0' at the end of the measurement period and 1_{A_j} be the corresponding indicator random variable. We first derive the probability for A_j to occur and the expected value of U_s. For an arbitrary bit in $X(src)$, each of the k contacts made by src has a probability of $\frac{1}{s}$ to set the bit as one, and each of the contacts made by other sources has a probability of $\frac{1}{m}$ to set it as one. All contacts are independent of each other when setting bits in B. Hence,

$$Prob\{A_j\} = (1 - \frac{1}{m})^{n-k}(1 - \frac{1}{s})^k, \quad \forall j \in [0..s-1]$$

Since U_s is the number of '0' bits in the virtual vector, $U_s = \sum_{j=0}^{s-1} 1_{A_j}$. Hence,

$$\begin{aligned} E(V_s) &= \frac{1}{s}E(U_s) = \frac{1}{s}\sum_{j=0}^{s-1} E(1_{A_j}) = \frac{1}{s}\sum_{j=0}^{s-1} Prob\{A_j\} \\ &= (1 - \frac{1}{m})^{n-k}(1 - \frac{1}{s})^k \qquad (4\text{–}6) \\ &\simeq e^{-\frac{n-k}{m}} e^{-\frac{k}{s}}, \;\; as \;\; (n-k), m, k, s \to \infty \\ &\simeq e^{-\frac{n}{m} - \frac{k}{s}} \quad as \;\; k \ll m \qquad (4\text{–}7) \end{aligned}$$

The above equation can be rewritten as

$$k \simeq -s \cdot \frac{n}{m} - s \cdot \ln(E(V_s)). \tag{4–8}$$

Since the bits in any virtual vector are selected from B uniformly at random, the process of storing n contacts in the virtual vectors is to *set n bits randomly selected (with replacement) from a pool of m bits.* The mathematical relation between n and m has been given in [67] (in a database context) as follows.

$$n \simeq -m \cdot \ln(E(V_m)) \tag{4–9}$$

$$\text{where } E(V_m) = (1 - \frac{1}{m})^n \tag{4–10}$$

Hence, equation 4–8 can be written as

$$k \simeq s \cdot \ln(E(V_m)) - s \cdot \ln(E(V_s)). \tag{4–11}$$

We have a few approximation steps above. In practice, n and m are likely to be very large numbers, the spread values (k) that are of interest are likely to be large, and s will be chosen large. The approximation errors that are accumulated in equation 4–11 can be measured as

$$\frac{|s \cdot \ln(E(V_m)) - s \cdot \ln(E(V_s)) - k|}{k} = |s \cdot \ln(\frac{1 - \frac{1}{m}}{1 - \frac{1}{s}}) - 1|$$

which is independent of n and k. This error is very small when s is reasonably large. For example, when $m = 1MB$, as shown in Fig. 4-1, the error is only 0.25% when s is 200.

Let $k_1 = -s \cdot \ln(E(V_m))$ and $k_2 = -s \cdot \ln(E(V_s))$. Eq (4–11) is rewritten as

$$k \simeq -k_1 + k_2.$$

Replacing $E(V_m)$ and $E(V_s)$ by the instance values, V_m and V_s, that are obtained from B and $X(src)$ respectively, we have the following estimation for k_1, k_2 and k.

$$\hat{k_1} = -s \cdot \ln(V_m) \tag{4–12}$$

$$\hat{k_2} = -s \cdot \ln(V_s) \tag{4–13}$$

$$\hat{k} = -\hat{k_1} + \hat{k_2} \tag{4–14}$$

According to Theorem A4 in [67], $\hat{k_1}$ is the maximum likelihood estimator (MLE) of k_1. Following a similar analysis, it is straightforward to see that $\hat{k_2}$ and $\hat{k}$ are the maximum likelihood estimators of k_2 and k, respectively. $\hat{k_1}$ is the noise, the estimated number of contacts made by others but inserted in $X(src)$ due to bit sharing between virtual vectors, and $\hat{k_2}$ estimates the total number of contacts stored in $X(src)$, including the noise.

4.3.4 System Architecture

The spread estimation system consists of a sampling module, CSE, and a module for Storing distinct Source Addresses, denoted as SSA. CSE has two sub-modules: one for Storing Contacts, denoted as CSE-SC, and the other for Spread Estimation, denoted as CSE-SE, which have been described in the previous two subsections. CSE-SC is located in the high-speed memory (such as SRAM) of a router, and CSE-SE is located on an offline computer answering spread queries.

The sampling module is used to handle the mismatch between the line speed and the processing speed of CSE-SC. In case that CSE-SC cannot keep up with the line speed, the source/destination addresses of each arriving packet will be hashed into a number in a range $[0, N)$. Only if the number is greater than a threshold $T(< N)$, the contact is forwarded to CSE-SC. The threshold can be adjusted to match CSE-SC with the line speed. The final estimated spread of a source will become $\hat{k}\frac{N}{T}$. The focus of this dissertation is on CSE, assuming an incoming stream of contacts, regardless whether it comes from a sampling module or not.

Most applications, such as those we discuss in the previous section, are interested in high-spread sources. For them, we do not have to invoke SSA for each packet. When CSE-SC stores a contact at a bit in B, *only if the bit is set from '0' to '1'*, the source address is passed to the SSA module, which checks whether the address has already been

stored and, if not, keeps the address. Comparing with CSE-SC, SSA operates infrequently. First, numerous packets may be sent from a source to a destination in a TCP/UDP session, only the first packet may invoke SSA because the rest packets will set the same bit. Second, while a source may send thousands or even millions of packets through a router, the number of times its address is passed to SSA will be bounded by s (which is the number of bits in the source's virtual vector). Hence, SSA can be implemented in the main memory, thanks to its infrequent operation.

For CSE-SE to work, m and s should be chosen large enough such that the noise introduced by other sources does not set all (or most) bits in a virtual vector. Hence, it is unlikely that the address of a high-spread source will not be stored in SSA. For example, even when only 10% of the bits in a virtual vector are not set by noise, for a source making 100 distinct contacts, the probability for none of its contacts being mapped to those 10% bits is merely $(1 - 10\%)^{100} = 2.65 \times 10^{-5}$.

4.4 Analysis

We first study the mean and variance of $\hat{k_1}$ and $\hat{k_2}$, based on which we analyze the accuracy of the spread estimation $\hat{k}$.

4.4.1 Mean and Variance of $\hat{k_1}$ and $\hat{k_2}$

After setting n bits randomly selected from a pool of m bits, Whang [67] uses $\hat{n} = -m \ln V_m$ to estimate the value of n and gives the following results.

$$E(\hat{n}) = E(-m \ln V_m) \simeq n + \frac{e^{\frac{n}{m}} - \frac{n}{m} - 1}{2}$$
$$Var(\hat{n}) = Var(-m \ln V_m) \simeq m(e^{\frac{n}{m}} - \frac{n}{m} - 1)$$

Since $\hat{k_1} = -s \cdot \ln(V_m)$, we have

$$E(\hat{k_1}) \simeq \frac{s}{m}(n + \frac{e^{\frac{n}{m}} - \frac{n}{m} - 1}{2}) \tag{4–15}$$
$$Var(\hat{k_1}) \simeq \frac{s^2}{m}(e^{\frac{n}{m}} - \frac{n}{m} - 1). \tag{4–16}$$

If we choose an appropriate memory size m such that $m = O(n)$ and $e^{\frac{n}{m}} - \frac{n}{m} - 1$ is negligible when comparing with n, then $E(\hat{k_1}) \simeq s\frac{n}{m}$, which is indeed the average noise that a virtual vector of size s will receive when all n contacts are evenly distributed across the space of m bits. When m is large, the standard deviation, which is the square root of $Var(\hat{k_1})$, is insignificant when comparing with the mean.

Next we study $\hat{k_2}$. Let $\alpha = \frac{n}{m} + \frac{k}{s}$. Equation 4–7 can be rewritten as

$$E(V_s) \simeq e^{-\alpha}. \tag{4–17}$$

We derive $Var(V_s)$, and it is

$$Var(V_s) \simeq \frac{1}{s}(e^{-\alpha} - e^{-2\alpha} - \frac{k}{s} \cdot e^{-2\alpha}). \tag{4–18}$$

Proof. The probability for A_i and A_j, $\forall i, j \in [0..s-1], i \neq j$, to happen simultaneously is

$$Prob\{A_i \cap A_j\} = (1 - \frac{2}{m})^{n-k}(1 - \frac{2}{s})^k.$$

Since $V_s = \frac{U_s}{s}$ and $U_s = \sum_{j=1}^{s} 1_{A_j}$, we have

$$\begin{aligned}
E(V_s^2) &= \frac{1}{s^2}E((\sum_{j=1}^{s} 1_{A_j})^2) \\
&= \frac{1}{s^2}E(\sum_{j=1}^{s} 1_{A_j}^2) + \frac{2}{s^2}E(\sum_{1 \leq i < j \leq s} 1_{A_j}1_{A_j}) \\
&= \frac{1}{s}(1 - \frac{1}{m})^{n-k}(1 - \frac{1}{s})^k + \frac{s-1}{s}(1 - \frac{2}{m})^{n-k}(1 - \frac{2}{s})^k.
\end{aligned}$$

Based on equation refequ100 and the equation above, we have

$$\begin{aligned}
&Var(V_s) = E(V_s^2) - E(V_s)^2 \\
&= \frac{1}{s}(1-\frac{1}{m})^{n-k}(1-\frac{1}{s})^k + \frac{s-1}{s}(1-\frac{2}{m})^{n-k}(1-\frac{2}{s})^k \\
&\quad - (1-\frac{1}{m})^{2(n-k)}(1-\frac{1}{s})^{2k} \\
&= \frac{1}{s}((1-\frac{1}{m})^{n-k}(1-\frac{1}{s})^k - (1-\frac{2}{m})^{n-k}(1-\frac{2}{s})^k) \\
&\quad + (1-\frac{2}{m})^{n-k}(1-\frac{2}{s})^k - (1-\frac{1}{m})^{2(n-k)}(1-\frac{1}{s})^{2k} \\
&\simeq \frac{1}{s}(e^{-\alpha} - e^{-2\alpha}) + e^{-2\frac{n-k}{m}-2\frac{k}{s}}(\frac{-k}{s^2}) \\
&\simeq \frac{1}{s}(e^{-\alpha} - e^{-2\alpha} - \frac{k}{s}e^{-2\alpha}).
\end{aligned}$$

□

In equation 4–13, $\hat{k_2}$ is a function of V_s. We expand the right-hand side of equation 4–13 by its Taylor series about $q = E(V_s) \simeq e^{-\alpha}$.

$$\hat{k_2}(V_s) = s \cdot (\alpha - \frac{V_s - q}{q} + \frac{(V_s-q)^2}{2q^2} - \frac{(V_s-q)^3}{3q^3} + ...) \qquad (4\text{–}19)$$

Since $q = E(V_s)$, the mean of the second term in equation 4–19 is 0. Therefore, we keep the first three terms when computing the approximated value for $E(\hat{k_2})$.

$$E(\hat{k_2}) \simeq s \cdot (\alpha + \frac{1}{2q^2}E((V_s-q)^2))$$

$E((V_s-q)^2)) = Var(V_s)$ by definition. Applying equation 4–18, we have

$$E(\hat{k_2}) = s \cdot (\alpha + \frac{e^{\alpha} - 1 - \frac{k}{s}}{2s}) \qquad (4\text{–}20)$$

If s is large enough such that $\frac{e^{\alpha}-1-\frac{k}{s}}{2s}$ is negligible, then $E(\hat{k_2}) \simeq s\alpha = s\frac{n}{m} + k$. Recall that $E(\hat{k_1}) \simeq s\frac{n}{m}$. Hence, $E(\hat{k}) = -E(\hat{k_1}) + E(\hat{k_2}) \simeq k$. In the next subsection, we will characterize more precisely the mean of $\hat{k}$ and how much it deviates from the true value of k.

To derive the variance of $\hat{k_2}$, we keep the first two items on the right-hand side of equation 4–19.

$$Var(\hat{k_2}) \simeq s^2 \cdot Var(\alpha - \frac{V_s - q}{q}) \\ = \frac{s^2}{q^2} \cdot Var(V_s) \simeq s(e^{\alpha} - \frac{k}{s} - 1) \tag{4–21}$$

The combined impact of $V(\hat{k_1})$ and $V(\hat{k_2})$ on the variance of $\hat{k}$ will be studied next.

4.4.2 Estimation Bias and Standard Deviation

Based on the means of $\hat{k_1}$ and $\hat{k_2}$ derived previously, we obtain the mean of the spread estimation $\hat{k}$.

$$E(\hat{k}) = E(\hat{k_2}) - E(\hat{k_1}) \\ \simeq s(\alpha + \frac{e^{\alpha} - 1 - \frac{k}{s}}{2s}) - \frac{s}{m}(n + \frac{e^{\frac{n}{m}} - \frac{n}{m} - 1}{2}) \tag{4–22}$$

The *estimation bias* is

$$E(\hat{k} - k) \simeq \frac{m(e^{\alpha} - 1 - \frac{k}{s}) - s(e^{\frac{n}{m}} - \frac{n}{m} - 1)}{2m} \tag{4–23}$$

As an example, for $n = 10,000,000$, $m = 2$ MB, and $s = 400$, 600 or 800, the bias with respect to k is shown in Table 4-1. It is very small when comparing with the true spread k.

The variance of $\hat{k}$ is

$$Var(\hat{k}) = Var(\hat{k_1}) + Var(\hat{k_2}) - 2Cov(\hat{k_1}, \hat{k_2}) \\ = Var(\hat{k_1}) + Var(\hat{k_2}) + 2\left[E(\hat{k_1})E(\hat{k_2}) - E(\hat{k_1}\hat{k_2})\right]. \tag{4–24}$$

We have already obtained $Var(\hat{k_1}), Var(\hat{k_2}), E(\hat{k_1})$ and $E(\hat{k_2})$, and thus only need to derive $E(\hat{k_1}\hat{k_2})$. Recall that $\hat{k_1} = s \cdot (-\ln(V_m))$ and $\hat{k_2} = s \cdot (-\ln(V_s))$. We expand $-\ln(V_m)$

and $-\ln(V_s)$ by their Taylor series about $p = e^{-\frac{n}{m}}$ and $q = e^{-\alpha}$, respectively.

$$\begin{aligned}
E(\hat{k_1}\hat{k_2}) &= s^2 E((-\ln(V_m))(-\ln(V_s))) \\
&= s^2 E((\frac{n}{m} - \frac{V_m - p}{p} + \frac{(V_m - p)^2}{2p^2} - \ldots) \\
&\quad \cdot (\alpha - \frac{V_s - q}{q} + \frac{(V_s - q)^2}{2q^2} - \ldots)) \\
&\simeq s^2 \Big[\frac{n}{m} E(\alpha - \frac{V_s - q}{q} + \frac{(V_s - q)^2}{2q^2}) \\
&\quad + \alpha E(\frac{n}{m} - \frac{V_m - p}{p} + \frac{(V_m - p)^2}{2p^2}) - \frac{n}{m}\alpha\Big] \\
&= s^2 \Big[\frac{n}{m}(\alpha + \frac{e^{\alpha} - 1 - \frac{k}{s}}{2s}) + \frac{\alpha}{m}(n + \frac{e^{\frac{n}{m}} - \frac{n}{m} - 1}{2}) - \frac{n}{m}\alpha\Big] \\
&= s^2 \Big[\frac{n}{m}\alpha + \frac{\frac{n}{m}(e^{\alpha} - 1 - \frac{k}{s})}{2s} + \frac{\alpha(e^{\frac{n}{m}} - \frac{n}{m} - 1)}{2m}\Big]
\end{aligned} \tag{4–25}$$

From equations 4–15, 4–16, 4–20, 4–21, 4–24 and 4–25, we can obtain the closed-form approximation of $Var(\hat{k})$, which we omit. The standard deviation, divided by k to show the relative value, is

$$StdDev(\frac{\hat{k}}{k}) = \frac{\sqrt{Var(\hat{k})}}{k} \tag{4–26}$$

We have made a number of approximations, particularly, the truncation of less significant items in the Tayler series when deriving $Var(\hat{k_1}), Var(\hat{k_2}), E(\hat{k_1})$ and $E(\hat{k_2})$ and $E(\hat{k_1}\hat{k_2})$. The standard deviation embodies all those approximations. In Section 4.5, Fig. 4-3-4-6, we will show the numerical values of the standard deviation calculated from equation 4–26 and compare them with the values measured from the experiments. The result demonstrates that the analytical approximations only introduce minor error when the source spread is not too small.

4.5 Experiments

We evaluate CSE through experiments using real Internet traffic traces. Our main goal in this chapter is to provide a good spread estimator that can work in a small memory. In most of our experiments, the memory size, when averaging over all sources appearing in the input stream of contacts, ranges from 1.15 bits per source to 9.21 bits

per source. Existing estimators that keep per-flow or per-source state [63, 64] will not work here as we have explained in Section 4.2. The only related work that can still be implemented in such a small memory is OSM (Online Streaming Module) [62]; however, as the experimental results will demonstrate, it does not work well. Hence, CSE is valuable in the sense that it substantially extends the low end of memory requirement for the function of spread estimation in practice.

It should be noted that CSE makes two hash operations and one memory access for storing each contact, whereas OSM makes $l + 1$ hash operations and l memory accesses, where l is typically three. While CSE's efficient online operations are clearly advantageous for high-speed routers, our evaluation will focus on the area that is less quantified so far — the accuracy of spread estimation.

4.5.1 Experiment Setup

We obtained inbound packet header traces that were collected through Cisco's NetFlow from the main gateway at University of Florida for six days from April 1st to 6th, 2005. We implemented CSE and OSM, and executed them with the input of the six days' data. The experimental results are similar for those days. In this section, we will only present the results for the first day.

In our experiments, the source of a contact is the IP address of the packet sender, and the destination is the IP address of the receiver. The traffic trace on April 1 has 3,558,510 distinct source IP addresses, 56,234 distinct destination addresses, and 10,048,129 distinct contacts. The average spread per source is 2.84; namely, each source makes 2.84 distinct contacts on average. Fig. 4-2 shows the number of sources at each spread value in log scale. The number of sources decreases exponentially as the spread value increases from 1 to around 500. After that, there is zero, one or a few sources for each spread value.

We always allocate the same amount of memory to CSE and OSM for fair comparison. In each experiment, we feed the contacts extracted from the traffic trace to CSE or OSM, which stores the contact information in its data structure (located in SRAM or high-speed

cache memory when deployed in a real router). The source addresses will be recorded in a separate data structure (located in the main memory because the operations for recording source addresses are performed infrequently as explained in Section 4.3.4). After all contacts are processed, we use CSE or OSM to estimate the spread of each recorded source (which should be performed on an offline computer such as the network management center in practice).

4.5.2 Accuracy of Spread Estimation

The first set of experiments compare CSE and OSM in the accuracy of their spread estimations. CSE has two configurable parameters: the memory size m and the virtual vector size s. We perform four experiments with m = 0.5MB, 1MB, 2MB, and 4MB, respectively. In each experiment, we choose a value for s that minimizes the standard deviation as defined in equation 4–26 at $k = 250$, which is the middle point of the range (0..500) in which the spreads of most sources fall (Fig. 4-2).

OSM also has two configurable parameters: the memory size m and the column size (the number of rows in the bit matrix). The original paper does not provide a means to determine the best column size, but it suggests that 64 bits are typical. We tried many other sizes, and the performance of OSM under different column sizes will be presented shortly. After comparison, we choose the column size to be 128, which we believe is better than or comparable with other sizes for our experiments.

Figs. 4-3-4-6 present the experimental results when the memory allocated is 0.5MB, 1MB, 2MB and 4MB, respectively. Each figure has four plots from left to right. Each point in the first plot (CSE) or the second plot (OSM) represents a source, whose x coordinate is the true spread k and y coordinate is the estimated spread $\hat{k}$. The line of $\hat{k} = k$ is also shown. The closer a point is to the line, the more accurate the spread estimation is. To make the figure legible, when there are too many sources having a certain spread k, we randomly pick five to show in the first two plots. The third and fourth plots present the bias, $E(\hat{k} - k)$, and the standard deviation, $\frac{Var(\hat{k})}{k}$, measured in

the experiment, respectively. Because there are too few sources for some spread values in our Internet trace, we divide the horizontal axis into measurement bins of width 25, and measure the bias and standard deviation in each bin. To verify the analytical result in Section 4.4, we also show the standard deviation numerically calculated from (4–26) and (4–24) as the curve under title "CSE_std_cal" in the fourth plot. We have the following experimental results.

- **First and Second Plots:** CSE works far better than OSM when the allocated memory is small. As the memory size increases, the performance of OSM improves and approaches toward the performance of CSE.

- **Third and Fourth Plots:** Both the bias and the standard deviation of CSE are much smaller than those of OSM. Moreover, the third plot shows that OSM is no longer a non-bias estimator when the memory is small. In fact, if we compare the absolute error $|\hat{k} - k|$ (that is not shown in the figures), the maximum absolute errors of CSE over the measurement bins are smaller than the average absolute errors of OSM in all four experiments.

- **Fourth Plot:** For CSE, the numerically-calculated standard deviation, which is the curve titled "CSE_std_cal", matches well with the experimentally-measured value, which is the curve titled "CSE_std_dev". It shows that the approximations made in the analysis do not introduce significant error.

4.5.3 Impact of Different s Values on Performance of CSE

The second set of experiments study the impact of different virtual-vector sizes s on the performance of CSE. We let $m = 1$MB and vary the value of s from 200 to 500,[3] while keeping the other parameters the same as in the previous set of experiments. Fig. 4-7 presents the bias and the standard deviation of CSE. The experimental results

[3] In the experiment of Fig. 4-5, the s value, which minimizes the standard deviation at $k = 250$ as calculated from (4–26), is 286.

show that the performance of CSE is not very sensitive to the choice of s. A wide range of s gives comparable results. In the right plot of the figure, a larger s value leads to a slightly greater standard deviation for sources whose spreads (k) are small and a slightly smaller standard deviation for sources whose spreads are large. when k is larger.

4.5.4 Impact of Different Column Sizes on Performance of OSM

The third set of experiments demonstrate the impact of different column sizes on the performance of OSM. We let $m = 1$MB and vary the column size r from 64 to 512, while keeping the other parameters the same as in the first set of experiments. Fig. 4-8 presents the bias and the standard deviation of OSM. None of the r values makes OSM a non-bias estimator. When r is too large (such as 512), both bias and standard deviation are large. When r is too small (such as 64), its estimated spread does not go beyond 267, as shown in the left plot of Fig. 4-9. Comparing $r = 256$ and $r = 128$, the former leads to a much larger standard deviation, as shown in the right plot of Fig. 4-8. The impact of larger deviation can also be seen by comparing the right plot of Fig. 4-9 where $r = 256$ and the second plot in Fig. 4-4 where $r = 128$.

4.5.5 An Application: Detecting Address Scan

Our last set of experiments compare CSE and OSM using an application for address scan detection. Suppose the security policy is to report all external sources that contact 250 or more internal destination during a day. If a source with a spread less than 250 is reported, it is called a *false positive*. If a source with a spread 250 or above is not reported, it is called a *false negative*. The *false positive ratio* (FPR) is defined as the number of false positives divided by the total number of sources reported. The *false negative ratio* (FNR) is defined as the number of false negatives divided by the number of sources whose spreads are 250 or more. The experimental results are shown in Table 4-2. Clearly, CSE outperforms OSM by a wide margin when we take both FPR and FNR into consideration. The FNR is zero for OSM when $m = 0.5$MB. That is because OSM is a bias estimator in such a small memory. Its FPR is 66.2%

CSE also has non-negligible FPR and FNR because its estimated spread is not exactly the true spread. To accommodate impreciseness to a certain degree, the security policy may be relaxed to report all sources whose estimated spreads are $250 \times (1 - \varepsilon)$ or above, where $0 \leq \varepsilon < 1$. If a source whose true spread is less than $250 \times (1 - 2\varepsilon)$ gets reported, it is called an ε-*false positive*. If a source with a true spread 250 or more is not reported, it is called an ε-*false negative*. The FPR and FNR are defined the same as before. The experimental results for $\varepsilon = 10\%$ are shown in Table 4-3, and those for $\varepsilon = 20\%$ are shown in Table 4-4, where the FPR and FNR for CSE are merely 0.1% and 0.6% respectively when $m = 1$MB.

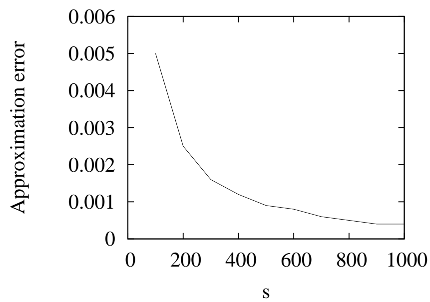


Figure 4-1. The approximation error is very small when s is reasonably large.

Table 4-1. Bias with respect to s and k

	k = 100	200	300	400	500	600	700	800
s= 400	0.54	0.77	1.05	1.47	2.04	2.82	3.85	5.21
s= 600	0.49	0.60	0.75	0.93	1.17	1.47	1.83	2.28
s= 800	0.47	0.54	0.63	0.75	0.88	1.05	1.24	1.47

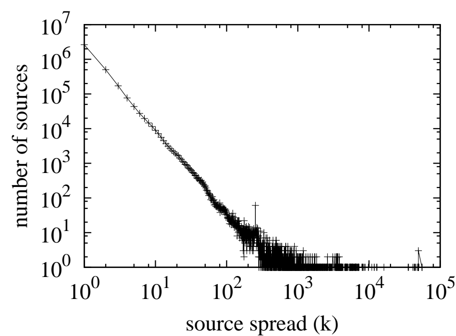


Figure 4-2. Traffic distribution: each point shows the number of sources having a certain spread value.

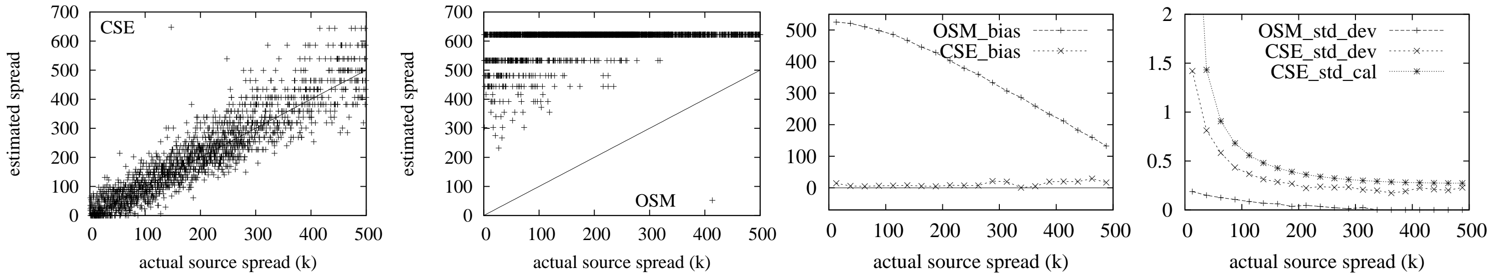


Figure 4-3. $m = 0.5$MB. Each point in the first plot (CSE) or the second plot (OSM) represents a source, whose x coordinate is the true spread k and y coordinate is the estimated spread $\hat{k}$. The third plot shows the bias of CSE and OSM, which is the measured $E(\hat{k} - k)$ with respect to k. The fourth plot shows the standard deviation, which is the *measured* $\frac{\sqrt{Var(\hat{k})}}{k}$ for CSE and OSM, together with the *numerically-calculated* standard deviation for CSE based on (4–26) and (4–24).

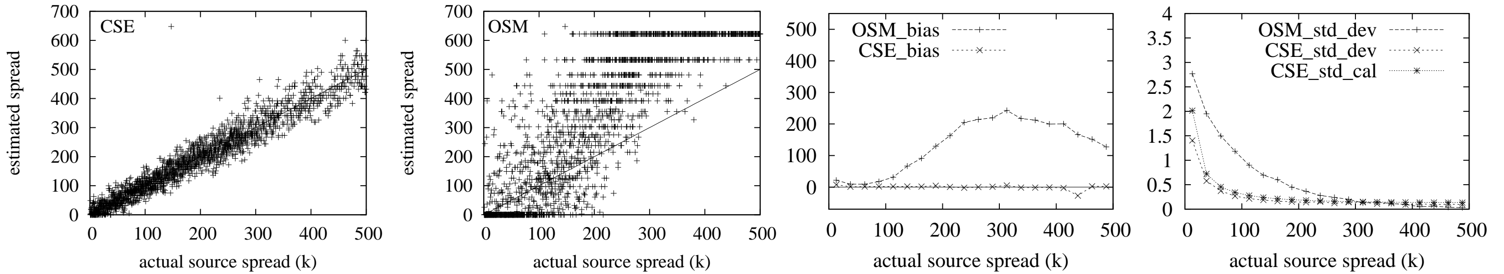


Figure 4-4. $m = 1$M. See the caption of Fig. 4-3 for explanation.

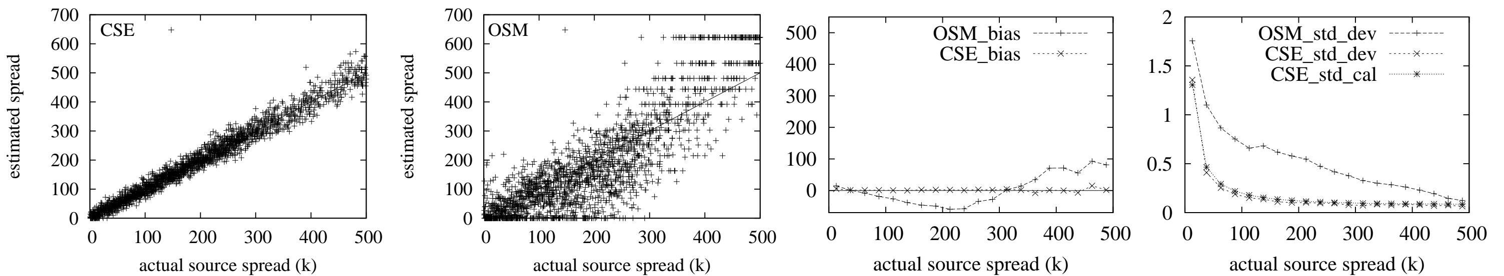


Figure 4-5. $m =$2MB. See the caption of Fig. 4-3 for explanation.

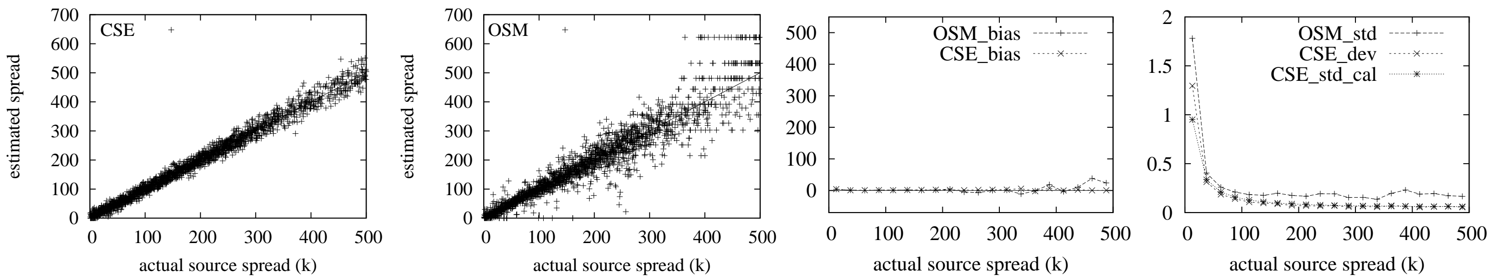


Figure 4-6. $m =$4M. See the caption of Fig. 4-3 for explanation.

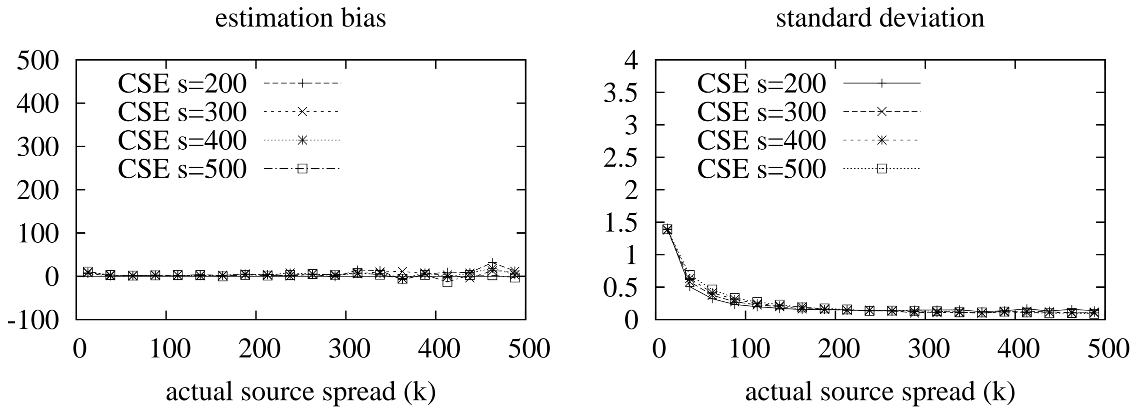


Figure 4-7. Left plot shows the bias of CSE, which is the measured $E(\hat{k}-k)$ with respect to k. Right plot shows the standard deviation of CSE, which is the *measured* $\frac{\sqrt{Var(\hat{k})}}{k}$.

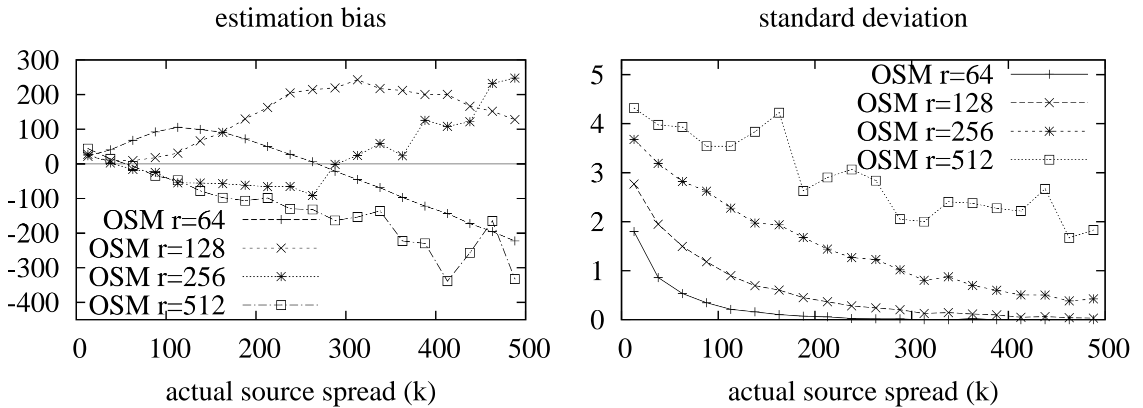


Figure 4-8. Left plot shows the bias of OSM, which is the measured $E(\hat{k}-k)$ with respect to k. Right plot shows the standard deviation of OSM, which is the *measured* $\frac{\sqrt{Var(\hat{k})}}{k}$.

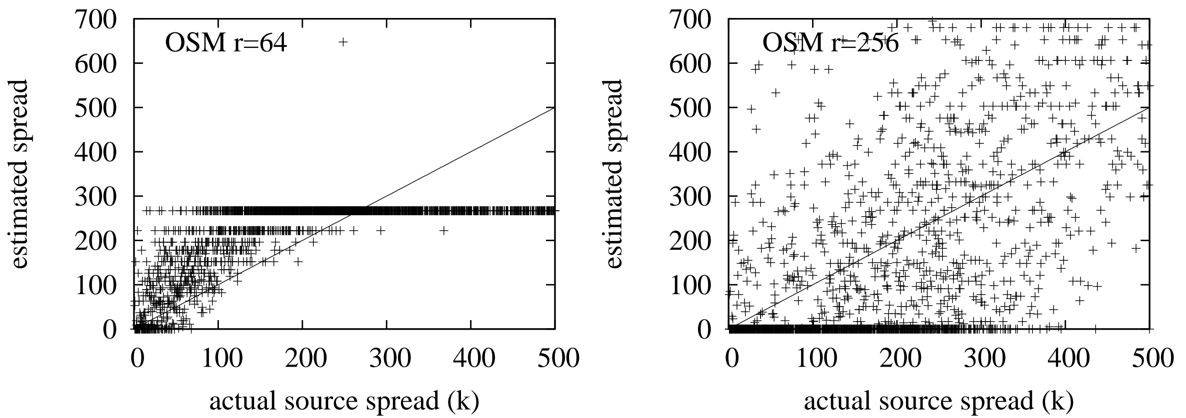


Figure 4-9. Left plot shows the distribution of $(k, \hat{k})$ for all sources under OSM when $r = 64$, where k and $\hat{k}$ are the true spread and the estimated spread, respectively. Right plot shows the distribution of $(k, \hat{k})$ under OSM when $r = 256$.

Table 4-2. False positive ratio and false negative ratio with respect to memory size.

m(MB)	OSM		CSE	
	FPR	FNR	FPR	FNR
0.5	0.662	0.000	0.164	0.123
1	0.424	0.008	0.097	0.094
2	0.116	0.236	0.073	0.056
4	0.108	0.115	0.053	0.062

Table 4-3. With $\varepsilon = 10\%$, false positive ratio and false negative ratio with respect to memory size.

m(MB)	OSM		CSE	
	FPR	FNR	FPR	FNR
0.5	0.532	0.000	0.077	0.057
1	0.251	0.006	0.031	0.027
2	0.041	0.193	0.005	0.014
4	0.023	0.064	0.001	0.002

Table 4-4. With $\varepsilon = 20\%$, false positive ratio and false negative ratio with respect to memory size.

m(MB)	OSM		CSE	
	FPR	FNR	FPR	FNR
0.5	0.401	0.000	0.023	0.022
1	0.135	0.002	0.001	0.006
2	0.013	0.146	0.000	0.002
4	0.006	0.030	0.000	0.000

CHAPTER 5
REAL-TIME DETECTION OF INVISIBLE SPREADERS

Detecting spreaders can help an intrusion detection system identify potential attackers. The existing work can only detect aggressive spreaders that scan a large number of distinct addresses in a short period of time. However, stealthy spreaders may perform scanning deliberately at a low rate. We observe that these spreaders can easily evade the detection because their small traffic footprint will be covered by the large amount of background normal traffic that frequently flushes any spreader information out of the intrusion detection system's memory. We propose a new streaming scheme to detect stealthy spreaders that are invisible to the current systems. The new scheme stores information about normal traffic within a limited portion of the allocated memory, so that it will not interfere with spreaders' information stored elsewhere in the memory. The proposed scheme is light weight; it can detect invisible spreaders in high-speed networks while residing in SRAM. Through experiments using real Internet traffic traces, we demonstrate that our new scheme detects invisible spreaders efficiently while keeping both false-positives (normal sources misclassified as spreaders) and false-negatives (spreaders misclassified as normal sources) to low level.

5.1 Motivation

Monitoring and analyzing network logs is at the heart of identifying attackers in the early stage [68]. These logs can be low-level packet trace generated from routers or high-level audit records from network/host intrusion detection systems. In high-speed networks, such logs can come in large volume. To process them in real time, a fast and lightweight streaming algorithm is required, which should be able to work with limited memory and contiguously process incoming logs.

This chapter studies the problem of detecting spreaders based on incoming logs that are tuples of source/destination addresses. We call an external source address a *spreader* if it connects to more than a threshold number of distinct internal destination addresses

during a period of time (such as a day). We define the *spread* of a source to be the number of distinct destinations that the source have contacted. Similarly, we define the destination spread to be the number of distinct sources that have contacted the destination.

The reason for detecting spreaders is that attackers often begin with a reconnaissance phase of finding vulnerable systems before launching the actual attack. Suppose an attacker knows how to compromise a specific type of web servers. Its first step is to locate such web servers on the Internet. The attacker may probe TCP port 80 on all addresses in a target network by using Nmap. To obtain more specific information, they may run an application-level vulnerability scanner such as Nessus or Paros. An intrusion detection system can inspect the incoming traffic and catch the reconnaissance packets, from which the spreaders are identified as potential attackers that demand extra attention.

It is not possible for a network security administrator to manually analyze the huge volume of logs produced by routers and intrusion detection systems in order to find spreaders. An automatic log-analyzing system is required. In fact, some intrusion detection systems have already implemented functions for identifying spreaders. For example, Snort [65] keeps track of the distinct destinations each source contacts in a recent period, and the length of the period is constrained by the amount of memory allocated to this function. The problem is that the existing systems are designed to catch "elephants" — aggressive spreaders whose cardinalities are so large that they easily stand out from the background of normal traffic. In response, a wily attacker will slow down the rate of its reconnaissance packets and let the normal traffic dilute the footprint of its activity. In the Snort case, the past records must be deleted to free memory once the allocated space is filled up by logs. If the attacker contacts a less-than-threshold number of destinations in each period during which logs of normal traffic will fill up the allocated space, it will stay undetected.

We note that even state-of-the-art intrusion detection systems cannot detect stealthy spreaders if they send their packets at a low rate. These spreaders are called *invisible*

spreaders. To catch them, we must make the detection system more sensitive. It is a cat-and-mouse game between attackers and defenders. As we build more and more sensitive detectors, the attackers will be forced to continuously reduce their reconnaissance rate in order to stay undetected. This will give more time for the network administrators to take action (such as patching systems) against the outbreak of new attacks. The attacks will become less effective if it will take them an exceedingly long time (e.g., months) to complete the reconnaissance phase over the Internet.

To design our new real-time detector for invisible spreaders, we observe (based on real Internet traffic traces) that normal traffic has strong skewness especially in an enterprise (or university campus) network. In particular, most inbound traffic is headed to a small number of servers for web, DNS, email, and business application services. Utilizing such skewness, we propose a new spreader detection scheme that is able to largely segregate the space used to store normal-traffic logs from the space used to store logs of potential spreaders. Due to such segregation, a large volume of normal-traffic logs will not cause the logs of spreaders to be flushed out of the memory. Furthermore, with a compact two-dimensional bit array based on Bloom filters, the new scheme can store a much larger amount of information about the spreaders, allowing previously invisible spreaders to be detected. We perform experiments based on real Internet traffic traces, and the results show that the proposed scheme is able to detect spreaders that are invisible to the existing detection systems and, at the mean time, keep both false positives (normal sources misclassified as spreader sources) and false negatives (spreader sources misclassified as normal sources) to low level.

5.2 Invisible-Spreader Detection

In this section, we propose a new scheme for detecting invisible spreaders. Our main technique is a novel streaming algorithm based on an *invisible-spreader detection filter*.

5.2.1 Invisible-Spreader Detection Filter (ISD)

Consider an intrusion detection system that is deployed to catch all external spreaders whose spread value exceeds a threshold θ. When the small footprint of the stealthy spreaders is sufficiently diluted by normal traffic, the spreaders may even become *invisible* to the current detection system. To catch these invisible spreaders, more sensitive detection systems must be designed. Below we propose a new detector that can catch spreaders invisible to today's detectors (such as [62]).

Our *invisible-spreader detection filter* (ISD) uses an $n \times m$ bit array as its main data structure, which is initialized to be all zeros. Each bit $B(x, y)$ in the array is referenced by a row index x and a column index y. Bits will be set to ones to record the incoming connections made from external sources to internal destinations. A row is *empty* (or *non-empty*) if it has zero bit (or at least one bit) that is set to be one. There is a *row counter* $c(x)$ for each row x, storing the number of bits in the row that are set as one. The *fullness ratio* R of the filter is defined as $\frac{\sum c(x)}{n \times m}$, the percentage of bits in the array that are set to one. Similarly, the fullness ratio of row x is defined as $\frac{c(x)}{m}$, the percentage of bits in the row that are set to one. We define a *system parameter* α, specifying the desirable fullness. If $R > \alpha$, we reset the bit array to zeros.

Next we describe the operations of ISD. When receiving an input source/destination tuple (a, b), the filter computes k row indexes, $x_1 = h_1(a)$, ..., $x_k = h_k(a)$, and one column index, $y = h_{k+1}(b)$, where $h_1, ..., h_k$ are hash functions whose ranges are $[0..n-1]$ and h_{k+1} is a hash function whose range is $[0..m-1]$. The filter sets k bits, $B(x_1, y)$, ..., $B(x_k, y)$, to be one. Note that each column is actually a Bloom filter [69] [70]. The column index y selects a Bloom filter and the row indices specify the bits that together represent the tuple (a, b).

For each $i \in [1..k]$, if $B(x_i, y)$ was set from zero to one, the filter increases the row counter $c(x)$ by one. Rows indexed by x_1 through x_k are called the *representative rows* of source a in the filter. Bits $B(x_1, y)$ through $B(x_k, y)$ are called the *representative bits*

of a. If the fullness of every representative row of source a is above a threshold β (whose value will be determined in the next subsection), ISD executes the following procedure to determine if a is a spreader.

1. For the jth column, let I_j be one if $B(x_i, j) = 1$ for all the representative rows of a. Otherwise, $I_j = 0$. We define
$$a_r = \sum_{j=0}^{m-1} I_j$$
2. The spread of a, denoted as $\hat{a_c}$, can be estimated based on the following formula given in [71].
$$\hat{a_c} = m \times ln(\frac{m}{m - a_r}) \tag{5–1}$$
3. If $\hat{a_c}$ is above θ, we consider source a to be a spreader.

Our column index, $y = h_{k+1}(b)$, is different from [62], which uses $y = h_{k+1}(a|b)$. This subtle yet critical difference helps ISD minimize the diluting effect of normal traffic over the small traffic footprint of invisible spreaders. Suppose a destination address b represents a busy webserver in an enterprise network, and millions of client users connect to b. If $y = h_{k+1}(a|b)$ is used, these clients will fill up the whole bit array with ones since the source addresses a randomizes the column index y. To the contrary, only one column of the bit array will be set to ones if $y = h_{k+1}(b)$ is used. Our Internet trace shows that the vast majority of normal traffic is directed to a small number of servers. Our scheme concentrates such normal traffic to a small number of columns in the bit array, leaving the rest of the array for detecting spreaders. Hash collisions may cause false positives. By tuning the system parameters, we can control the level of false positive, as well as the level of false negative.

5.2.2 Parameter Configuration

The goal of ISD is to identify spreaders whose spread values are larger than θ, which is given as a user requirement. Let M $(= n \times m)$ be the size of the allocated memory. The

performance of ISD is affected by the selection of the following system parameters: α, β, m, and n. Below we discuss how to set these parameters.

1) We first set the values for β and m. According to the previous subsection, a spreader will be detected when the following condition is satisfied.

$$m \times ln(\frac{1}{1-\frac{a_r}{m}}) \geq \theta. \tag{5–2}$$

Based on their definitions, we can approximate β as $\frac{a_r}{m}$. Applying this approximation to (5–2), we have the following formula for setting the value of β and m.

$$\beta = 1 - e^{-\frac{\theta}{m}}. \tag{5–3}$$

Recall that the parameter β is used to trigger the procedure for determining a possible spreader. When the value of β is set by the above formula, once triggered the procedure is likely to find a spreader.

The problem is that there are two undecided parameters in the formula. We observe that small m is desirable for ISD. This is because small m allows large n, which reduces hash collisions among row indices. Consequently, large β is preferable. However, if β is very close to one, ISD may suffer from hash collisions among column indices. In this dissertation, we choose β to be below 0.95, but it can be adjusted according to any specific application or deployment environment. Once β is chosen, m can be set based on (5–2). Alternatively, we may also set m first and then calculate β from (5–2). For example, it is natural to choose m as a multiple of words, which makes it easy to fit the bit array in memory. For each m $(= 32, 64, ...)$, we compute β and choose the largest β below 0.95. Table 5-1 shows some examples for parameter configuration. It shows how β and m are determined for θ from 100 to 900. After m is determined, n is calculated as $\frac{M}{m}$.

2) We now determine the value of α. First we examine how α affects the detection of spreaders. When α is too larger, the bit array of ISD will be overly populated with ones, causing frequent hash collisions and resulting in false positives — a non-spreader is

claimed as a spreader because its representative rows are populated with ones by tuples of other sources (due to hash collisions). If α is too small, false positives may hardly happen, but the filter will be frequently reset to zeros, losing the already-recorded information about spreaders and resulting in false negatives — failure in detecting spreaders. Next we will use some statistical properties to determine the value of α.

Suppose ISD only receives normal traffic for a period of time and its bit array is mostly set by the normal traffic. Let Y be a random variable that represents $c(x)$ for row x in the bit array. The expectation and the variance of Y are given below. We omit the derivation process due to page limit.

$$E(Y) = \alpha \times m \tag{5–4}$$

$$V(Y) = \alpha \times m \times (1-\alpha) \tag{5–5}$$

From $E(Y)$ and $V(Y)$, we can define a statistical upperbound for Y as follows:

$$U(Y) = E(Y) + c\sqrt{V(Y)} \tag{5–6}$$

where statistical error will be small if the constant c is large. Eq. (5–6) means that there is a high probability that $c(x)$ is below $U(Y)$ if x is a representative row for only normal traffic. On the contrary, if x is a representative row for any spreader, $c(x)$ should be larger than $U(Y)$. Hence, based on the above equations, we can set the value of α as follows.

$$\alpha = \frac{2\beta m + c^2}{2(m+c^2)} - \sqrt{(\frac{2\beta m + c^2}{2(m+c^2)})^2 - \frac{m\beta^2}{m+c^2}}. \tag{5–7}$$

Table 5-1 shows α as a result of the proposed heuristic method to configure β, m and α when $c = 10$.

5.3 Experiment

We evaluate *ISD* using real-world Internet traffic traces. We implemented not only *ISD* but also the advanced scheme from [62], which we call *online streaming module* (OSM) as in the previous chapter. We compare their false positives and false negatives.

The experimental results confirm that ISD detects invisible spreaders while minimizing the negative impact of normal traffic.

5.3.1 Traffic Trace and Implementation Details

In these experiments, we set k to 3. Large k is helpful to differentiate sources, but it increases processing time and fills up quickly the bit table. A good argument for $k = 3$ can be found in [62].

We use packet header traces gathered at the gateway routers of the University of Florida. The trace was collected for 24 hours and we take only the inbound session from the Internet. It contains 751,286 distinct source IP addresses, 120,916 distinct destination IP addresses and 2,427,327 distinct source/destination tuples. Note that we denote the source IP address of a packet as a and the destination IP address as b in our notation of packet (a, b). In this sense, the goal of the experiment is to find heavy spreaders of horizontal network scans [60].

Figure 5-1(5-2) illustrates the traffic pattern with respect to source(destination) spread. The x-axis is the number of sources(destinations) whose spread lies between x and $2 \times x - 1$. Each figure has two graphs of cumulative ratios for the number of distinct sources(destinations) and the number of distinct source/destination tuples. In figure 5-1, we see that 86% of the total sources contact less than 4 distinct destinations and 99% of them contact less than 32 distinct destinations. Figure 5-1 shows that the number of source/destination tuples increases just as the number of sources does. Therefore, we cannot see a strong skewness in the figure. However, we can see a different pattern in figure 5-2. The figure shows that only some of the destinations occupy most of the source/destination tuples. For example, at $x = 8$, the accumulated number of destinations is above 97%, but their aggregated source/destination tuples are below 27%. It means that less than top 3% servers occupy more than 73% of the total source/destination tuples. Exploiting this skewness, ISD has the edge on other intrusion detection systems.

For all the experiments, we set θ to be 500. It means that we take any source of spread above 500 as a spreader or scan source. With $\theta = 500$, the original traffic trace already includes 75 spreaders. We also generate some artificial scan packets to simulate invisible spreaders. For each experiment, we add 20 artificial slow scan sources to the original traffic trace. These source addresses are carefully chosen so that the original traffic trace does not include any same source address as the artificial scan sources. Each artificial scan source will send a total of λ distinct scan packets. It generates a scan packet every other μ normal source/destination tuples. The default parameters for the experiments are as follows: $M = 256KB, m = 256, n = 8,192, \alpha = 0.547, \alpha_O = 0.4, \theta = 500, \lambda = 600, \mu = 1,024, k = 3$. Note that β and α are determined by equations 5–3 and 5–7.

For comparison, we also implemented OSM [62]. For a fair comparison, both bit tables of OSM and ISD have the same memory size M. To optimize OSM, the maximum number of one-bits for OSM is set to α_O, which is different from α. Through the experiments, we observe that OSM degrades if α_O is set too large or too small. The default value of α_O is 0.4. Once the ratio of one-bits is above α_O, the decoding process runs and OSM restarts in a clean state.

For each experiment, we compare false negative(positive) sets of OSM and ISD. We use $FN_O(FN_R)$ to denote the false negative set of $OSM(ISD)$. Similarly, we use $FP_O(FP_R)$ to denote false positive sets. Let RS be a set of real spreaders, which has 95 sources (75 spreaders from the original traffic trace and 20 artificial scan sources). Let $D_O(D_R)$ be a set of detected sources by $OSM(ISD)$. We define FN_O, FN_R, FP_O and FP_R as follows: $FN_O = RS - D_O$, $FN_R = RS - D_R$, $FP_O = D_O - RS$, $FP_R = D_R - RS$.

5.3.2 Experimental Results

Figures 5-3-5-6 compare the numbers of false negatives(positives) between ISD and OSM. The x-axis of each figure is μ, the number of normal source/destination tuples between two slow scan packets. A large value of μ implies that the attacker further slows

down in sending the scan packets. In figure 5-3, we have four curves. OSM(total) is the number of false negatives of OSM, so it equals $|FN_O|$ with μ from 128 to 16,384. OSM(slow scans) is the number of false negatives, but we only count the artificial slow scan sources that are not detected. Therefore, its maximum value is 20 as we have 20 artificial slow scan sources. The same notations are used for ISD such as ISD(total) and ISD(slow scans). Note that ISD(total) plots $|FN_R|$.

Figure 5-3 shows that ISD catches most spreaders until μ becomes 4,096. Even when $\mu = 16,384$, ISD catches 17 artificial spreaders out of 20. To the contrary, OSM misses much more spreaders than OSM. Even when $\mu = 128$, it misses 7 non-artificial spreaders. It starts missing artificial scan sources at $\mu = 256$. At $m = 8,192$, OSM cannot detect any slow scan sources while ISD detects 16 out of 20. Note that we trade false positives with false negatives when designing ISD, but false positives should be controlled by setting α to be tight. Figure 5-4 shows it. Even at $\mu = 16,384$, ISD only triggers 9 false positives. Considering that the number of source/destination tuples is above two millions, this false positives may be accepted in most applications.

We repeat the same experiment with different n. Figures 5-5 and 5-6 show the result with $n = 32,768$, which means $M = 1MB$. In this experiment, ISD does not miss any spreaders including slow scan sources except one at $\mu = 16,384$. Note that both ISD(total) and ISD(slow scans) remain zero until $\mu = 8,192$. To the contrary, OSM still misses some spreaders as shown in the figure. It cannot detect 8 out of 20 slow scan sources at $\mu = 16,384$ even though the memory size has quadrupled. It is encouraging that ISD accomplishes better detection accuracy even when M is as small as $256KB$. Figure 5-6 shows that ISD triggers only small false positives.

Table 5-1. Parameter configuration examples ($c = 10$)

θ	100	200	300	400	500	600	700	800	900
β	0.790	0.790	0.904	0.790	0.858	0.904	0.935	0.790	0.828
m	64	128	128	256	256	256	256	512	512
α	0.249	0.365	0.463	0.478	0.547	0.598	0.634	0.571	0.612

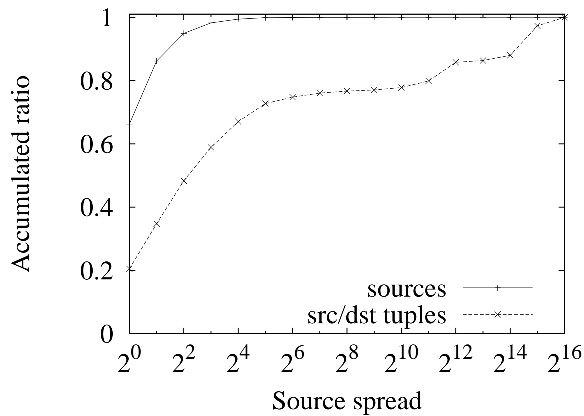


Figure 5-1. Cumulative ratios of the numbers of distinct sources and distinct source/destination tuples with respect to source spread

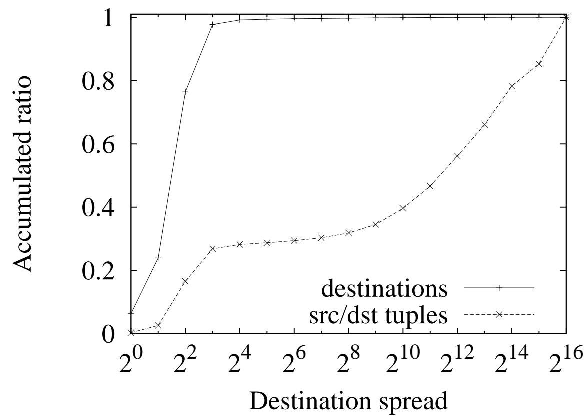


Figure 5-2. Cumulative ratios of the numbers of distinct destinations and distinct source/destination tuples with respect to destination spread

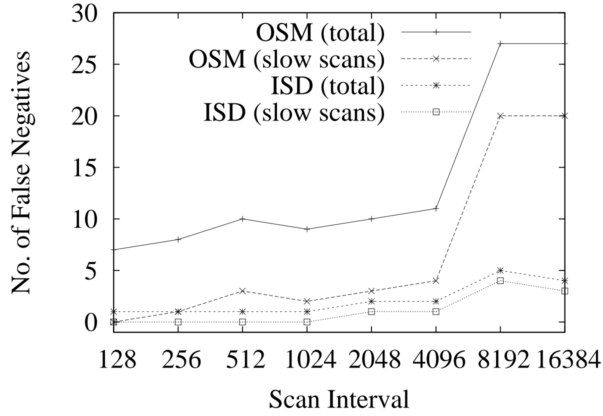


Figure 5-3. Number of false negatives when M=256KB

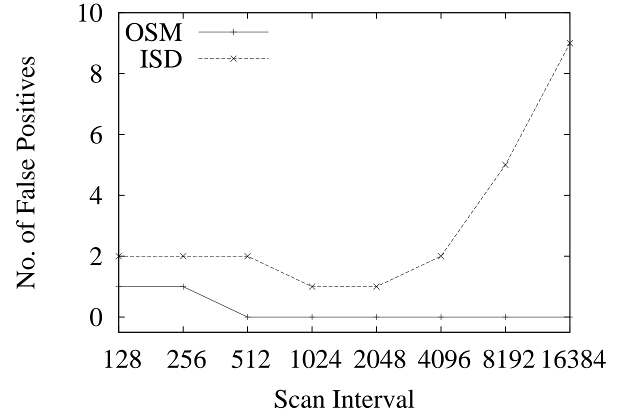


Figure 5-4. Number of false positives when M=256KB

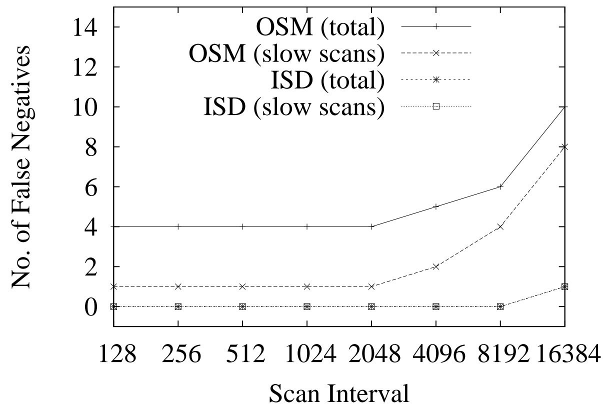


Figure 5-5. Number of false negatives when M=1MB

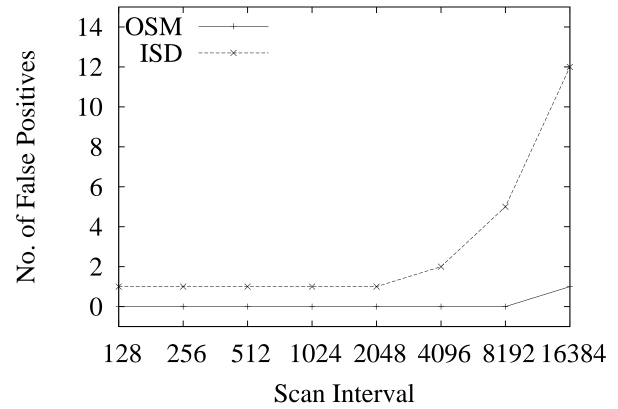


Figure 5-6. Number of false positives when M=1MB

CHAPTER 6
CONCLUSION

This dissertation discusses several novel techniques that secure computer networks.

First, we study the firewall placement problem and its variations. The problem is to place the firewalls in a network topology and find the routing structure such that the maximum size of the firewall rule sets in the network is minimized. We prove the problem is NP-complete and propose a heuristic algorithm, called HAF, to solve the problem approximately. The algorithm can also be used to solve the firewall routing problem as well as weighted firewall placement/routing problems.

Second, we propose a novel path address scheme (PAS) to address the source-address spoofing problem on the Internet. With a completely new design, PAS avoids the performance problems of the best-known scheme, Pi. We discuss how to construct addresses for paths, how to verify if path addresses are authentic, how to store path addresses in the IPv4 header, how to protect PAS against eavesdropping, and how to deal with router compromise. Our analysis and simulations demonstrate that PAS can simultaneously keep the false-positive ratio and false-negative ratio to almost zero. The path address scheme may potentially be used for other network applications. Examples include packet classification and service differentiation based on path addresses.

Third, we study the problems of spreader detection and spread estimation. The proposed spreader detection scheme detects invisible spreaders and mitigates the negative effects of normal traffic. Our spread estimator not only achieves space compactness but also operates more efficiently than the existing work. Our main technical contributions include a novel data structure based on virtual vectors, its operation protocol, and the corresponding formula for spread estimation, which is statistically analyzed and experimentally verified.

REFERENCES

[1] A. Rubin, D. Geer, and M. Ranum, "Web Security Sourcebook," *Wiley Computer Publishing*, 1997.

[2] J. Wack, K. Cutler, and J. Pole, "Guidelines on Firewalls and Firewall Policy," *National Institute of Standards and Technology*, January 2002.

[3] K. N. Y. Bartal, A. Mayer and A. Wool, "Firmato: a novel firewall management toolkit," *ACM Transactions On Computer Systems*, vol. 22, no. 4, pp. 381–420, November 2004.

[4] A. Wool, "A Quantitative Study of Firewall Configuration Errors," *IEEE Computer*, vol. 37, no. 6, pp. 62–67, June 2004.

[5] M. G. Gouda and A. X. Liu, "Firewall Design: Consistency, Completeness and Compactness," *Proc. of ICDCS'04*, pp. 320–327, March 2004.

[6] A. X. Liu and M. G. Gouda, "Diverse Firewall Design," *Proc. of IEEE International Conference on Dependable Systems and Networks (DSN'04)*, pp. 595–604, June 2004.

[7] A. X. Liu, E. Torng, and C. Meiners, "Firewall Compressor: An Algorithm for Minimizing Firewall Policies," *Proc. of IEEE INFOCOM'08*, pp. 595–604, April 2008.

[8] A. X. Liu, E. Torng, and C. Meiners, "The use and usability of direction-based filtering in firewalls," *Computers & Security*, vol. 6, no. 23, pp. 459–468, April 2004.

[9] A. X. Liu, E. Torng, and C. Meiners, "Optimization of Network Firewall Policies Using Ordered Sets and Directed Acyclical Graphs," *Proc. of IEEE Internet Management Conference*, 2005.

[10] E. S. Al-Shaer and H. H. Hamed, "Discovery of Policy Anomalies in Distributed Firewalls," *Proc. of IEEE INFOCOM'04*, March 2004.

[11] R. N. Smith, Y. Chen, and S. Bhattacharya, "Cascade of Distributed and Cooperating Firewalls in a Secure Data Network," *IEEE Trans. On Knowledge and Data Engineering*, vol. 15, no. 5, 2003.

[12] R. N. Smith and S. Bhattacharya, "Firewall Placement in a Large Network Topology," *Proc. of IEEE FTDCS'97*, 1997.

[13] A. El-Atawy, T. Samak, E. Al-Shaer, and H. Li, "On Using Online Traffic Statistical Matching for Optimizing Packet Filtering Performance," *Proc. of IEEE INFOCOM'2007*, May 2007.

[14] H. Hamed, A. El-Atawy, and E. Al-Shaer, "On Dynamic Optimization of Packet Matching in High Speed Firewalls," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 10, Oct 2006.

[15] P. Gupta and N. McKeown, "Algorithms for Packet Classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, March 2001.

[16] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," *Proc. of ACM SIGCOMM'99*, 1999.

[17] T. Lakshman and D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching," *Proc. of ACM SIGCOMM'98*, 1998.

[18] A. Hari, S. Suri, and G. Parulkar, "Detecting and Resolving Packet Filter Conflicts," *Proc. of IEEE Infocom'00*, March 2000.

[19] V.Srinivasan, G.Varghese, S.Suri, and M.Waldvogel, "Fast and Scalable Layer Four Switching," *Proc. of ACM SIGCOMM'98*, 1998.

[20] P. Gupta, "Algorithms for Routing Lookups and Packet Classification," *PhD Thesis, Stanford University*, 2000.

[21] A. X. Liu and M. G. Gouda, "Removing Redundancy from Packet Classifiers," *Poster Session, ACM SIGCOMM'04*, 2004.

[22] H. Court, Knutsford, and Cheshire, "High-Availability: technology brief firewall load balancing," *http://www.High-Availability.Com*, 2008.

[23] N. Networks, "Firewall load balancing," *www.nortel.com*, 2008.

[24] C. Point, "Check Point Firewall-1 Guide," *www.checkpoint.com*, 2008.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, , and C. Stein, "Introduction to Algorithms," *The MIT Press*, 2003.

[26] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical Network Support for IP Traceback," *Proc. of ACM SIGCOMM'00*, August 2000.

[27] D. J. Bernstein, "SYN cookies," *http://cr.yp.to/syncookies.html*, 1997.

[28] A. Juel and J. Brainard, "Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks," *Proc. of Network and Distributed System Security Symposium (NDSS'99)*, February 1999.

[29] K. Park and H. Lee, "On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets," *Proc. of ACM SIGCOMM'01*, August 2001.

[30] A. Bremler-Barr and H. Levy, "Spoofing Prevention Method," *Proc. of INFOCOM'05*, March 2005.

[31] P. Ferguson and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks Which Employ IP Source Address Spoofing," *IETF, RFC 2267*, Janurary 1998.

[32] A. D. Keromytis, V. Misra, and D. Rubenstein, "SOS: Secure Overlay Services," *Proc. of ACM SIGCOMM'02*, August 2002.

[33] A. Yaar, A. Perrig, and D. Song, "FIT: Fast Internet Traceback," *Proc. of IEEE INFOCOM, Miami, Florida*, March 2005.

[34] A. Yaar, A. Perrig, and D. Song, "StackPi: New Packet Marking and Filtering Mechanisms for DDoS and IP Spoofing Defense," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 10, October 2006.

[35] P. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling High Bandwidth Aggregates in the Network," *Computer Communications Review*, vol. 32, no. 3, pp. 62–73, July 2002.

[36] C. Kaufman, R. Perlman, and M. Speciner, "Network Security - Private Communication in a Public World (2nd Edition)," *Prentice Hall PTR*, 2002.

[37] J. Xu and W. Lee, "Sustaining Availability of Web Services under Distributed Denial of Service Attacks," *IEEE Transactions on Computers*, vol. 52, no. 2, pp. 195–208, 2003.

[38] T. Aura, P. Nikander, and J. Leiwo, "DoS-Resistant Authentication with Client Puzzles," *Cambridge Security Protocols Workshop 2000. LNCS, Springer-Verlag*, 2000.

[39] D. Dean and A. Stubblefield, "Using Client Puzzles to Protect TLS," *10th Annual USENIX Security Symposium*, 2001.

[40] X. Wang and M. K. Reiter, "Defending Against Denial-of-Service Attacks with Puzzle Auctions," *2003 IEEE Symposium on Security and Privacy*, May 2003.

[41] D. G. Andersen, "Mayday: Distributed Filtering for Internet Services," *Proc. of 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[42] W. G. Morein, A. Stavrou, D. L. Cook, A. D. Keromytis, V. Misra, and D. Rubenstein, "Using Graphic Turing Tests to Counter Automated DDoS Attacks Against Web Servers," *Proc. of the 10th ACM International Conference on Computer and Communications Security (CCS)*, October 2003.

[43] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford, "CAPTCHA: Using Hard AI Problems For Security," *Proc. of EUROCRYPT'03*, 2003.

[44] R. Stone, "CenterTrack: An IP Overlay Network for Tracking DoS Floods," *Proc. of the 9th USENIX Security Symposium*, August 2000.

[45] A. Yaar, A. Perrig, and D. Song, "Pi: A Path Identification Mechanism to Defend against DDoS Attacks," *IEEE Symposium on Security and Privacy*, May 2003.

[46] S. M. Bellovin, "ICMP Traceback Messages," *Internet Draft: draft-bellovin-itrace-00.txt*, March 2000.

[47] A. C. Snoren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer, "Hash-Based IP Traceback," *Proc. of ACM SIGCOMM'01*, August 2001.

[48] B. R. Smith and J. J. Garcia-Luna-Aceves, "Securing the Border Gateway Routing Protocol," *Proc. of Global Internet'96*, November 1996.

[49] S. Kent, C. Lynn, and K. Seo, "Secure Border Gateway Protocol (Secure-BGP)," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 4, April 2000.

[50] Y. Rekhter and T. Li, "A Border Gateway Protocol 4 (BGP-4)," *IETF Network Working Group, RFC 1771*, March 1995.

[51] T. Li and G. Huston, "BGP Stability Improvements," *IETF Internet-Domain Routing, Internet-Draft, draft-li-bgp-stability-01*, June 2007.

[52] C. Villamizar, R. Chandra, and R. Govindan, "BGP Route Flap Damping," *IETF Network Working Group, RFC 2439*, November 1998.

[53] H. Wang, D. Zhang, and K. G. Shin, "SYN-dog: Sniffing SYN Flooding Sources," *Proc. of 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, July 2002.

[54] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On Power-Law Relationships of the Internet Topology," *Proc. of ACM SIGCOMM'99*, 1999.

[55] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting," *Proc. of ACM SIGCOMM'02*, October 2002.

[56] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-Based Change Detection: Methods, Evaluation, and Applications," *Proc. of IMC'03*, pp. 234–247, 2003.

[57] A. Kumar, M. Sung, J. Xu, and J. Wang, "Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution," *Proc. of ACM SIGMETRICS*, 2004.

[58] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li, "Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement," *Proc. of IEEE INFOCOM*, March 2004.

[59] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lundn, "Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Application," *Proc. of ACM SIGCOMM IMC*, October 2004.

[60] S. Staniford, J. Hoagland, and J. McAlerney, "Practical Automated Detection of Stealthy Portscans," *Journal of Computer Security*, vol. 10, pp. 105 – 136, 2002.

[61] D. Plonka, "FlowScan: A Network Traffic Flow Reporting and Visualization Tool," *Proc. of USENIX LISA*, 2000.

[62] Q. Zhao, J. Xu, and A. Kumar, "Detection of Super Sources and Destinations in High-Speed Networks: Algorithms, Analysis and Evaluation," *IEEE JSAC*, vol. 24, no. 10, October 2006.

[63] S. Venkatataman, D. Song, P. Gibbons, and A. Blum, "New Streaming Algorithms for Fast Detection of Superspreaders," *Proc. of NDSS'05*, Feb. 2005.

[64] C. Estan, G. Varghese, and M. Fish, "Bitmap Algorithms for Counting Active Flows on High-Speed Links," *IEEE/ACM Trans. on Networking*, vol. 14, no. 5, October 2006.

[65] M. Roesch, "Snort–Lightweight Intrusion Detection for Networks," *Proc. of 13th Systems Administration Conference, USENIX*, 1999.

[66] Y. Gao, Y. Zhao, R. Schweller, S. Venkataraman, Y. Chen, D. Song, and M. Kao, "Detecting Stealthy Spreaders Using Online Outdegree Histograms," *Proc. of IEEE International Workshop on Quality of Service'07*, pp. 145–153, June 2007.

[67] K. Whang, B. Vander-Zanden, and H. Taylor, "A Linear Time Probabilistic Counting Algorithm for Database Applications," *ACM Transactions on Database Systems*, June 1990.

[68] B. Schneier, "SIMS: Solution, or Part of the Problem?," *IEEE Security and Privacy*, vol. 2, no. 5, October 2004.

[69] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[70] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, vol. 1, no. 4, June 2002.

[71] K. Hwang, B. Vander-Zanden, and H. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Transactions on Database Systems*, vol. 15, no. 2, June 1990.

BIOGRAPHICAL SKETCH

MyungKeun Yoon was born in Seoul, Republic of Korea, in 1973. He received his BS and MS degrees in computer science at Yonsei University in Korea in 1996 and 1998, respectively. After receiving his master degree, he worked for the Korea Financial Telecommunications and Clearings Institute, where he took the lead in many security related projects. Since 2004, he has been conducting research with Dr. Shigang Chen in the department of Computer and Information Science and Engineering at the University of Florida. His research interests are network security and mobile network.