Traffic Analysis for Network Security using Learning Theory and Streaming Algorithms

Shobha Venkataraman

CMU-CS-08-157
September 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported in part by NSF grants CCR-0105488, NSF-ITR CCR-0122581 and CNS-0433540, and by Cylab at Carnegie Mellon under grant DAAD19-02-1-0381 from the Army Research Office. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of NSF, ARO, Carnegie Mellon, the U.S. government, or any other entity.

Keywords: network security, machine learning, learning theory, traffic analysis, streaming algorithms

Abstract

A recurring problem in network traffic analysis is to automatically distinguish legitimate traffic from malicious or spurious traffic. This problem arises in several guises in network security (e.g., spam mitigation, worm detection), and is, at core, a machine learning or data mining problem. However, traffic analysis for network security has many fundamental challenges that are not present in typical machine learning or data mining problems, and a blackbox application of classical algorithms may not address these challenges adequately. For example, many standard machine learning algorithms may not scale to the volume and diversity of network traffic, or perform well in the presence of a malicious adversary who aims to evade detection. It is, therefore, necessary to design algorithms that meet these challenges, and provide formal guarantees on how well they have been met by the algorithms and the extent to which they can be met by any algorithm.

In this thesis, we consider four problems in network security with these challenges, and we use tools from computational learning theory and streaming algorithm design to address them. In each of these four problems, the difference between the malicious traffic and normal traffic is characterized by a specific structure of the traffic distributions: temporal structure, structure in content, structure in communication patterns of hosts and network structure given by host IP addresses. We present both efficient algorithms as well as fundamental lower bounds for these problems:

- In the stepping-stones problem, we use the temporal structure of the traffic – in particular, the inter-packet timing delays – to identify pairs of streams that are likely to be stepping-stones. We provide algorithms with strong upper bounds on the number of packets they need to observe, to detect attacks with given false positive and false negative rates. We also present lower bounds showing how an adversary, with sufficient chaff, could evade any detection mechanism that is based only on the timing delays between packets.
- When generating signatures for exploits with pattern-extraction techniques, the content structure of network traffic is used to identify packets that are likely to be worms. A sequence of prior work has alternately developed pattern-extraction algorithms for signature-generation, and attacks on these pattern-extraction algorithms. We present lower bounds showing how *any* pattern-extraction algorithm could be misled, in the presence of an adversary with sufficient control over the malicious data.

- We present efficient streaming algorithms to identify *superspreaders*, which are sources that contact many distinct destinations in a short time period. The communication structure of most hosts on Internet makes finding superspreaders of interest to security applications, as they are likely indicators of worms, scanning, or other malicious activity. Our experimental results on real network traces show that our algorithms are substantially more efficient than earlier approaches.
- Finally, we study the problem of tracking regions of the IP address space that send malicious traffic. In the first part of our work, we focus on spam traffic, and explore whether the history and structure of IP addresses could be used to distinguish spammers from senders of legitimate mail. In the second part, we design online algorithms that, with low space requirements, can dynamically track IP prefixes that originate the malicious traffic, and provide a near-optimal prediction of IP addresses that send malicious traffic and normal traffic. Our experimental results demonstrate that our algorithm finds prefixes that are orders of magnitude more accurate than fixed commonly-used IP prefixes.

Acknowledgements

My first thanks go to my advisors, Avrim Blum and Dawn Song. I am extremely fortunate to have had two such wise, patient, and helpful advisors, who gave so generously of their time towards the risk of exploring this very new area of common interest. The efforts that they invested and the perspectives that they provided were instrumental in shaping my research sense in this bridge area, and looking back, it was a very lucky morning in September 2002 when I decided to explore this area!

I thank the rest of my thesis committee – Phillip Gibbons, Bruce Maggs, Subhabrata Sen and Oliver Spatscheck – for agreeing to see this thesis through, and for the valuable advice that they've shared with me over the years – on research, on talks, on papers and on job applications. They have also been among my closest collaborators, and working with them has helped me become a better writer, presenter and researcher. I am deeply grateful to Daphne Koller and Yoav Shoham, my undergraduate thesis advisors, for patiently getting me started in computer science research, and I owe a special thanks to Carlos Guestrin for being an amazing mentor during my senior year at Stanford.

I have had a great many more wonderful collaborators over the last six years, the consequence of working in an area at the intersection of many others. I thank them all – Juan Caballero, Yan Chen, Yan Gao, Patrick Haffner, Min Gyung Kang, Pongsin Poosankam, Robert Schweller, Jennifer Yates, Yao Zhao – for all that I have learnt from them. I have also been very lucky to have had friends who have immensely improved my research and presentation – David Brumley, Hubert Chan, Elena Nabieva, Vyas Sekar, and Runting Shi have proof-read many papers, fixed many talks, combed through many proofs, and served as research sounding boards over the last six years.

The staff in CSD and ECE have been instrumental in making my life smooth over the years. I am especially grateful to Sharon Burks and Deb Cavlovich, who have pulled all kinds of strings for me and patiently resolved one complicated situation after another. And of course, some of my biggest thanks go to all my friends – amidst all the ups and downs, they have made these years wonderful and helped me keep my perspective on life.

Finally, I thank my parents and my brother for their constant support and encouragement through all these years. This thesis would not have been possible without their love and prayers.

Contents

1 Introduction **1**

1.1 Stepping-Stone Detection 4

1.2 Limits of Signature-Generation using Pattern-Extraction Techniques 5

1.3 Streaming Algorithms for Fast Detection of Superspreaders 6

1.4 Tracking Malicious Regions of the IP Space 6

1.4.1 Exploiting IP-based Network Structure for Spam Mitigation 7

1.4.2 Dynamically Tracking IP Prefixes Originating Malicious Traffic 7

1.5 Structure of this Thesis 8

2 Detection of Stepping-Stone Attacks **9**

2.1 Introduction 9

2.2 Related Work 12

2.3 Problem Definition 13

2.4 Main Results: Detection Algorithms and Confidence Bounds 15

2.4.1 A Simple Poisson Model 15

2.4.2 Generalizing the Poisson Model 19

2.5 Chaff: Detection and Hardness Result 21

2.5.1 Algorithm for Detection with Chaff 21

2.5.2 Hardness Result for Detection with Chaff 23

2.6 Conclusion 27

3 Limits of Signature-Generation with Learning-based Algorithms **29**

3.1 Introduction 29
3.2 Definitions and Overview 31
3.2.1 Definitions 31
3.2.2 Overview of Learning Framework 32
3.3 Reflecting Set 34
3.3.1 Motivation & Definitions 34
3.3.2 Finding Reflecting Sets 35
3.4 General Adversarial Model 36
3.4.1 Learning Model 36
3.4.2 Results 38
3.4.3 Practical Implications 43
3.5 Exploiting Gaps in Traffic 45
3.5.1 Defining the Gap 46
3.5.2 Learning Model 47
3.5.3 Results 47
3.5.4 Practical Implications 51
3.6 Related Work 52
3.7 Conclusion 53

4 Streaming Algorithms for Fast Detection of Superspreaders 55

4.1 Introduction 55
4.2 Problem Definition and Previous Approaches 58
4.2.1 Problem Definition 58
4.2.2 Related Work and Previous Approaches 59
4.3 Algorithms for Finding Superspreaders 61
4.3.1 One-level Filtering Algorithm 61
4.3.2 Two-Level Filtering Algorithm 64
4.4 Extensions 68
4.4.1 Superspreaders with Deletion 68
4.4.2 Superspreaders over Sliding Windows 69

4.4.3 Distributed Superspreaders 71
4.5 Experimental Results 72
4.5.1 Experimental evaluation of precision 72
4.5.2 Memory usage on long traces 74
4.5.3 Comparison with an Alternate Approach 78
4.6 Conclusion 79

5 Exploiting IP-based Network Structure for Spam Mitigation 81
5.1 Introduction 81
5.2 Analysis of IP-Address Characteristics 83
5.2.1 Data 83
5.2.2 Analysis of IP Addresses 84
5.3 Analysis of Cluster Characteristics 92
5.3.1 Cluster Granularity 93
5.3.2 Persistence 95
5.4 Spam Mitigation under Mail Server Overload 97
5.4.1 Server Overload Problem 98
5.4.2 Design and Algorithms 99
5.4.3 Evaluation 102
5.5 Related Work 107
5.6 Conclusion 108

6 Tracking IP Prefixes Originating Malicious Traffic Dynamically 109
6.1 Introduction 109
6.2 Definitions and Preliminaries 110
6.3 Algorithms and Analysis 112
6.3.1 Overview of *TrackIPTree* 112
6.3.2 Subproblems of *TrackIPTree* 113
6.3.3 The Complete *TrackIPTree* Algorithm 116
6.3.4 Analysis 117
6.3.5 Modifying TrackIPTree for Experimental Issues 119

6.4 Evaluation 120

6.4.1 Data and Preliminaries 120

6.4.2 A Baseline Algorithm 121

6.4.3 Comparisons with Baseline Algorithm 121

6.4.4 Effect of Changing m 122

6.4.5 Cluster Distribution over Time 125

6.5 Related Work 125

6.6 Conclusion 126

7 Conclusion **127**

Bibliography **129**

Chapter 1

Introduction

Network traffic analysis is an important part of computer security, yet it grows more challenging each day: network traffic grows in volume and complexity, hosts on the Internet grow in number and diversity, and attacks grow in variety and sophistication. Identifying patterns of normal and anomalous traffic automatically, and using these patterns to monitor unseen traffic is, therefore, of prime importance.

A major line of research has focused on using machine learning and data mining algorithms to automatically distinguish normal traffic from attacks and anomalies. Machine learning and data mining algorithms are attractive in this area as they can automatically extract the distinctive features of anomalies and attacks out of vast quantities of data, and therefore, they have the potential to produce highly accurate detection over the data with little manual effort. A second advantage these algorithms offer is speed of detection and mitigation – when they can quickly detect drastic changes in traffic patterns and allow the associated traffic to be filtered out, they can enable attacks and anomalies to be mitigated much faster, with little human intervention. Many systems have demonstrated the feasibility of using machine learning techniques in a variety of security applications, e.g., anomaly and intrusion detection [14, 74, 110, 75, 101, 54, 53, 76], traffic profiling [87, 126, 130], worm detection [70, 103, 71, 91, 77], spam filtering [102, 86, 8, 83]. Most of this research has, however, focused on applying machine learning and data mining algorithms as blackboxes to network traffic analysis. Because of this, many challenges fundamental to network security problems may go unresolved, as they do not occur in the typical machine learning problems.

In security problems, one important challenge comes from the presence of an intelligent adversary who aims to evade detection. Unlike the typical machine learning problem, an adversary is in control of one type of data (malicious data), and can often manipulate features of the malicious data over time to evade detection. The algorithm might also be forced to mislearn – i.e., to learn a function that labels normal traffic incorrectly – and cause so many false positives that it becomes completely unusable. The history of spam-filtering [83, 39] is a compelling example of how this kind of adversarial reaction could affect performance – even as spam filters get more and more sophisticated, spammers get more and more adept at desiging spam to evade the existing spam-filters. Thus, an algorithm's performance on the evaluated data may not necessarily reflect its future performance: the algorithms may learn irrelevant artefacts (or artefacts maliciously added just to mislead it) as the distinctive features of malicious data, and so may not detect future attacks. As a result, it is important to provide algorithms with mathematical guarantees on their performance in adversarial environments. The challenge is to design detection algorithms that are resilient to adversaries, quantifying the resilience as a function of the adversary's power, and to develop well-specificed guarantees on detection that go beyond the standard training and test error analysis.

Further, for sufficiently powerful adversaries, there are fundamental limits to the kind of detection that might be possible using machine learning approaches. With enough control over the data, he might not need to know the specific parameters, or the exact algorithm used, in order to evade detection or force mislearning: merely the structure of the solution used – i.e., knowledge of the features used and the distribution of normal traffic – might be sufficient. In many problems, the adversary does have a significant amount of such power. For example, he can choose how to craft the content of the malicious packets, and affect all algorithms that use content. By choosing when to attack different hosts, he can make attacks appear dissimilar, and he can evade detection algorithms that use timing-related features across different hosts. With botnets that are widespread, attackers can also control where to launch attacks, and affect detection algorithms that use IP addresses of hosts. To understand the kinds of fundamental limits that are applicable, we aim to quantify them with lower bounds on the performance of different classes of algorithms, and in terms of the kind of power the adversary has.

The volume and diversity of network traffic poses another challenge – the algorithms in use need to be extremely scalable, and utilize available resources very efficiently. ISPs routinely collect tens or hundreds of gigabytes of data everyday, and many standard machine learning algorithms would not scale, even if learning or analysis can be performed offline. With this kind of data, if an algorithm's running time or memory requirements is quadratic in the size of the data, it is almost certainly impossible to run it over hundreds of millions of packet flows on a daily basis. For this reason, it is important to quantify the kind of performance guarantees that can be obtained for offline analysis, when only a small set of features or small subset of the data can be used. It is also important to design more efficient approximations of these algorithms, that can scale over the data, perhaps by taking into consideration the particular structure of network traffic in that problem.

Algorithms that need to be operated online, or perhaps deployed at high-speed monitoring points, pose even more challenges. These algorithms may have to operate at links that see on order of millions of packets a minute, in order to perform effective real-time attack detection and mitigation. These algorithms cannot even afford to operate even in space linear in the data set; they have to be sub-linear in the size of the data set. The algorithm may observe each packet only once, and will also need to make very few computational operations per packet. These constraints imply that one-pass streaming algorithms are required to address these problems, and in addition to their usual challenges in the streaming model, the algorithms are allowed only a very small amount of per-packet processing.

Our work: The approach we take in this area, instead of blackbox application of standard algorithms, is to pose a problem with a set of formal objectives that need to be achieved, and then design algorithms and analyze how well these objectives have been met by the algorithms. The formal guarantees are needed to ensure that the challenges of the specific problem have been adequately met, because, as discussed earlier, it is not sufficient to take learning algorithms' performance on an existing data set as an indicator of future performance. In addition, it also allows us to analyze when fundamental guarantees are achievable for a particular security problem, as a function of the adversary's power and the amount of data seen. To develop algorithms in this manner, the concepts and analytical techniques developed to address different issues (e.g, in machine learning, sample complexity, representation of hypothesis, tail inequalities) turn out to be more important than the original algorithms. While the resulting algorithms may not be as general as those in a standard machine learning or data mining formulation, they may be able to take advantage of the particular structure of the problem in question, and may therefore allow better scalability and more resilence to adversaries.

In this thesis, we take this approach to address four problems in network security, and we use tools from computational learning theory and streaming algorithm design to solve them. In each of these problems, the difference between the malicious and normal traffic is characterized by a specific kind of structure in network traffic: temporal structure, structure in content, structure in communication patterns of hosts and network structure given by host IP addresses. In each case, the malicious traffic is generated by an adversary, and the adversary may be able to manipulate the structure of the malicious traffic to evade detection. However, the adversary may or may not have any power to manipulate normal traffic. In these problems, we explore both the algorithmic aspects as well as the fundamental limits of security analysis. In what follows, we first briefly outline each of the problems below, and then elaborate our results on each problem in the rest of the section.

- The first problem we consider is the detection of *stepping-stone* attacks, i.e. , attacks launched through a chain of hosts on the Internet used as relay machines. Here, we use the temporal structure of the traffic – in particular, the inter-packet timing delays – to identify pairs of streams that are likely to be stepping-stones. We provide algorithms with strong upper bounds on the number of packets they need to observe, to detect attacks with given false positive and false negative rates. We also present lower bounds showing how an adversary, with sufficient chaff, could evade any detection mechanism that is based only on the timing delays between packets [17].

- In the second problem, we explore how an adversary could evade a class of signature-generation mechanisms that defend against fast-spreading worms, specifically, pattern-extraction techniques for signature-generation. These techniques use structure in the content of network traffic to identify packets that are likely to be worms, and a sequence of prior work [70, 104, 71, 91, 77, 97, 92, 61] has alternately developed pattern-extraction algorithms for signature-generation, and attacks on these pattern-extraction algorithms. We present lower bounds showing how *any* pattern-extraction algorithm could be misled, in the presence of an adversary with sufficient control over the malicious data [114].

- In the third problem, we present efficient streaming algorithms to identify *superspreaders*, which are sources that contact many distinct destinations. Identifying superspreaders is of great interest to security applications, as they are likely indicators of worms, scanning activity and malicious (or perhaps unwanted) activity, because most hosts contact only a small number of destinations in a short time period. We extend the algorithms to allow for distributed monitoring, streams with deletion, and to operate under sliding windows, and our experimental results on real network traces show that our algorithms are substantially more efficient than earlier approaches [116].

- In the fourth problem, we explore how to track regions of the IP space that send malicious traffic. In the first part of our work, we focus on spam mitigation as a concrete case study, and explore whether the history and structure of IP addresses can be used to distinguish spammers from senders of legitimate mail [115]. In the second part, we design online algorithms that, with low space requirements, can dynamically track IP prefixes that originate the malicious traffic, and provide a near-optimal prediction of IP addresses that send malicious traffic and normal traffic, over adversarially-generated data.

1.1 Stepping-Stone Detection

Intruders on the Internet often prefer to launch network intrusions indirectly, i.e., through a chain of hosts on the Internet as relay machines using protocols such as Telnet or SSH. The attacker types commands on his local machine and then the commands are relayed via the chain of "stepping stones" until they finally reach the victim. This type of attack is called a *stepping-stone* attack. Because the victim sees traffic only from the last hop of the chain, it is difficult for the victim to identify the attacker, and the volume of the traffic on the Internet makes such attacks extremely difficult to record or trace back.

We propose and analyze algorithms for stepping-stone detection using ideas from computational learning theory and the analysis of random walks. Like earlier work, we assume that detection is done at a monitoring point – we examine the traffic that goes in and out of routers, and try to detect which streams, if any, are part of a stepping-stone attack. Also like earlier work, we use timing delays between packets on streams, rather than content or the sizes of the packets, since the packets may be encrypted. Our results are the first to achieve provable (polynomial) upper bounds on the number of packets needed to confidently detect and identify encrypted stepping-stone streams with proven guarantees on the probability of falsely accusing non-attacking pairs. In addition, our methods and analysis rely on mild assumptions, especially in comparison to previous work.

Furthermore, we examine the consequences when the attacker inserts chaff – unnecessary dummy packets – into the stepping-stone traffic, in order to make the streams look uncorrelated. We show how our algorithm can still detect stepping-stone attacks with a limited amount of chaff. We then give lower bounds on the amount of chaff that an attacker would have to send to evade detection by *any* algorithm that uses only packet timing information.

Our results on the stepping-stone problem are based on a new approach with connections to sample-complexity bounds in learning theory, and allow us to detect correlation of streams at a fine-grained level. Our results may also apply to more generalized traffic analysis domains, such as anonymous communication. This is joint work with Avrim Blum and Dawn Song.

1.2 Limits of Signature-Generation using Pattern-Extraction Techniques

A *signature* of an exploit is a function that distinguishes malicious byte strings from non-malicious ones. Automatic signature generation is necessary because there may often be little time between the discovery of a vulnerability, and exploits developed to target the vulnerability. One major line of research effort on automatic signature generation has focused on pattern-extraction techniques to find signatures for exploits, *i.e.*, by extracting byte patterns that uniquely distinguish exploits using network traffic statistics [70, 103, 71, 91, 77]. Pattern-extraction techniques are attractive for signature-generation because signatures can be generated and matched efficiently, and several systems have demonstrated the existence of the necessary distinguishing byte patterns (invariants) in the exploit. This research has then led to interest in how pattern-extraction algorithms may be evaded [97, 92, 61].

We show fundamental limits on the accuracy of pattern-extraction algorithms for signature-generation in an adversarial setting. We formulate a natural framework that allows a unified analysis of these algorithms, and prove lower bounds on the number of mistakes any pattern-extraction learning algorithm must make for an arbitrary exploit, under common assumptions. While previous work has targeted specific algorithms and systems, our work generalizes these attacks through theoretical analysis to any algorithm with similar assumptions, not just the techniques developed so far.

We also analyze when pattern-extraction algorithms may work, by showing conditions under which these lower bounds are weakened. Our results show that there may be many classes of exploits for which these algorithms do work well, e.g., for exploits where there is a significant gap between the byte patterns seen in the normal traffic and the invariants of the exploit. Our results also illustrate the extent to which the difficulty of evasion depends on the complexity of the signature function learned. Our results are applicable to other kinds of signature-generation algorithms as well (e.g., COVERS [78]), those that use properties of the exploit that can be manipulated. This is joint work with Avrim Blum and Dawn Song.

1.3 Streaming Algorithms for Fast Detection of Superspreaders

We consider the problem of detecting *superspreaders*, which are sources that connect to a large number of *distinct* destinations. Superspreaders could be responsible for fast worm propagation – the Slammer worm, for instance, caused some infected hosts to send up to $26,000$ scans a second [88] – and thus it is important to detect them quickly. Monitoring on high-speed links, for example, on a large enterprise network or an ISP network, is desirable for real-time attack detection and mitigation; however, such high-speed network monitoring requires fast streaming algorithms that use very little memory space.

We propose new streaming algorithms for detecting superspreaders and prove guarantees on their accuracy and memory requirements. We also show experimental results on real network traces. Our algorithms are substantially more efficient (both theoretically and experimentally) than previous approaches. We extend our algorithms to identify superspreaders in a distributed setting, with sliding windows, and when deletions are allowed in the stream (which lets us identify sources that make a large number of failed connections to distinct destinations). In addition, one of our algorithms is also based on a novel two-level sampling scheme which may be of independent interest.

More generally, our algorithms are applicable to any problem that can be formulated as follows: given a stream of (x, y) pairs, find all the x's that are paired with a large number of distinct y's. We call this the *heavy distinct-hitters* problem. There are many network security applications of this general problem: detecting ports which have high ICMP traffic without storing per-port information, detecting spammers who send the same emails to many distinct destinations within a short period, identifying sources in peer-to-peer networks that communicate with many different hosts. Our algorithms apply to all of these problems; however, in our experiments, we focus on the superspreader problem. This is joint work with Dawn Song, Phillip B. Gibbons, and Avrim Blum.

1.4 Tracking Malicious Regions of the IP Space

It is well-known that tracking individual IP addresses, has a limited scope in reducing malicious traffic (e.g. blacklisted spamming bots [66, 99, 115]). With the growing trend of attackers to use botnets for their attacks, these results are unsurprising, as individual bots are expendable and the attacker's anonymity is preserved. Instead, there has been recent interest in correlating *regions of the IP space* that originate the malicious traffic. Several studies have demonstrated that significant amount of spam originates from a relatively small number of $/16$ or $/24$ IP prefixes [125, 99, 29].

In the first part of our work, we explore whether the history and structure of IP addresses can be used to distinguish spammers from senders of legitimate mail [115], focusing on the IP structure afforded by using network-aware clusters. In the second part of our work, we study the problem of finding the clusters that best partition the IP space, in order to allow us to predict the IP addresses that send malicious traffic and normal traffic.

1.4.1 Exploiting IP-based Network Structure for Spam Mitigation

From the perspective of spam mitigation, the IP address of the sender's mail server is an attractive discriminatory feature of spammers and legitimate senders: IP address information is computationally-efficient to extract and store, and cannot be camouflaged as easily as the email content. Our work analyzes the extent to which IP address information could be used to enhance spam mitigation, through measurement analysis and simulations. Our work shows the importance of using the history of both the legitimate senders and spammers in IP addresses. Our work also demonstrates that network-aware clusters may be a good way to aggregate the history of spamming IP addresses.

Through measurement analysis over a 6 month-long corpus of 28 million email messages, we analyzed how the history and structure of IP addresses could be useful for spam mitigation. Our results showed significant differences in the behaviour of legitimate senders and spamming IPs. We found that the bulk of the legitimate mail was sent by a small number of IPs that appeared frequently and sent little spam. However, IPs sending mostly spam did not last long, e.g., spammers responsible for over 80% of the total spam appeared no more than on 5 distinct days. When examining network-aware clusters, though, we observed that IP addresses appeared from the *same* spamming clusters over and over again: the cluster sending a lot of spam "lives" for a very long time, even if the spamming IP address was ephemeral.

To see if these differences could be used for prediction, we examined the mail-server overload problem – when the mail server receives much more mail than it can process, is there a way for the mail server to increase the legitimate mail accepted over the default selection? This problem has been routinely observed at the mail servers of many ISPs, because it is in the interest of the spammer to overload the mail server, if the mail server selects the mail to accept at random. Through simulation over email logs, we demonstrated that the historical behaviour of IP addresses and clusters can be utilized to increase the legitimate mail accepted by a factor of 3. Thus, even this fixed set of clusters affords significant discriminatory power, and suggests the importance of finding the optimal clusters. This is joint work with Subhabrata Sen, Oliver Spatscheck, Patrick Haffner and Dawn Song.

1.4.2 Dynamically Tracking IP Prefixes Originating Malicious Traffic

Our earlier research (Section 1.4.1) and related concurrent work [125, 99, 29, 85] indicate that many kinds of malicious traffic, (e.g,. spam, scans) are indeed correlated with relatively small regions of the IP space. The kind of correlation observed is unsurprising, as bots originate much of this malicious traffic, and networks that are easily compromised contain more bots than others. Prior work has, so far, focused on finding correlations with a *fixed* set of IP clusters: the analysis typically takes as input a set of IP clusters, and finds clusters among those that originate the most of the malicious traffic. Instead, we focus on automatically tracking the best IP clusters that isolate the malicious traffic.

We design online algorithms to identify the origin of malicious traffic with the best possible IP clusters, using only limited space. The question that we address is: can we partition the IP space into clusters that predict the IP addresses that are likely to send malicious traffic? Such clusters may have many applications: they may help identify future spammers or predict future botnet addresses; they may be useful for network management in discovering compromised subnets. In addition, the regions that originate malicious traffic may change over time for many reasons: attackers may be able to compromise more hosts, some networks or hosts may get patched and no longer send malicious traffic. Our algorithm also adapts when the clusters originating malicious traffic (and likewise, benign traffic) change over time.

We evaluate our algorithm on empirically real data of spam and legitimate mail from an enterprise network of a large corporation. Our algorithm is highly efficient, and produces predictions that are orders of magnitude better than fixed sets of IP clusters, such as network-aware clusters and /24 IP blocks.

This is joint work with Avrim Blum, Subhabrata Sen, Oliver Spatscheck and Dawn Song.

1.5 Structure of this Thesis

The rest of this thesis is organized as follows. In Chapter 2, we present algorithms and lower bounds for stepping-stones problem. In Chapter 3, we present fundamental limits of analysis in pattern-extraction algorithms for signature generation. Chapter 4 presents our work on designing streaming algorithms to find superspreaders. Chapter 5 presents the results of our analysis on using the history and structure of IP addresses for spam mitigation. In Chapter 6, we present algorithms that can track dynamically malicious regions of the IP address space. Chapter 7 summarizes and concludes the thesis.

Chapter 2

Detection of Stepping-Stone Attacks

2.1 Introduction

Intruders on the Internet often launch network intrusions indirectly, in order to decrease their chances of being discovered. One of the most common methods used to evade surveillance is the construction of *stepping stones*. In a stepping-stone attack, an attacker uses a sequence of hosts on the Internet as relay machines and constructs a chain of interactive connections using protocols such as Telnet or SSH. The attacker types commands on his local machine and then the commands are relayed via the chain of "stepping stones" until they finally reach the victim. Because the final victim only sees traffic from the last hop of the chain of the stepping stones, it is difficult for the victim to learn any information about the true origin of the attack. The chaotic nature and sheer volume of the traffic on the Internet makes such attacks extremely difficult to record or trace back.

To combat stepping-stone attacks, the approach taken by previous research (e.g., [106, 128, 127, 43]), and the one that we adopt, is to instead ask the question "What can we detect if we monitor traffic at the routers or gateways?" That is, we examine the traffic that goes in and out of routers, and try to detect which streams, if any, are part of a stepping-stone attack. This problem is referred to as the *stepping-stone detection problem*. A *stepping-stone monitor* analyzes correlations between flows of incoming and outgoing traffic which may suggest the existence of a stepping stone. Like previous approaches, we consider the detection of *interactive* attacks: those in which the attacker sends commands through the chain of hosts to the target, waits for responses, sends new commands, and so on in an interactive session. Such traffic is characterized by streams of packets, in which packets sent on the first link appear on the next a short time later, within some *maximum tolerable delay* bound Δ. The detection of non-interactive connections (i.e., those without a maximum delay bound Δ) is much harder, as there is no bounded time frame within which packets on different streams need to compared. Like previous approaches, we assume traffic is encrypted, and thus the detection mechanisms cannot rely on analyzing the content of the streams. We will call a pair of streams an *attacking pair* if it is a stepping-stone pair, and we will call a pair of streams a *non-attacking pair* if it is not a stepping-stone pair.

Researchers have proposed many approaches for detecting stepping stones in encrypted traffic. (e.g., [106, 128, 127]. See more detailed related work in Section 2.2.) However, most previous approaches in this area are based on ad-hoc heuristics and do not give any rigorous analysis that would provide provable guarantees of the false positive rate or the false negative rate [128, 127]. Donoho et al. [43] proposed a method based on wavelet transforms to detect correlations of streams, and it was the first work that performed rigorous analysis of their method. However, they do not give a bound on the number of packets that need to be observed in order to detect attacks with a given level of confidence. Moreover, their analysis requires the assumption that the packets on the attacker's stream arrive according to a Poisson or a Pareto distribution — in reality, the attacker's stream may be arbitrary. Wang and Reeves [121] proposed a watermark-based scheme which can detect correlation between streams of encrypted packets. However, they assume that the attacker's timing perturbation of packets is independent and identically distributed (*iid*), and their method breaks when the attacker perturbs traffic in other ways.

Thus, despite the volume of previous work, an important question still remains open: how can we design an efficient algorithm to detect stepping-stone attacks with (a) provable bounds on the number of packets that need to be monitored, (b) a provable guarantee on the false positive and false negative rate, and (c) few assumptions on the distributions of attacker and normal traffic?

Our work sets off to answer this question. In particular, in this thesis, we use ideas from Computational Learning Theory to produce a strong set of guarantees for this problem:

Objectives: We explicitly set our objective to be to distinguish attacking pairs from non-attacking pairs, given our fairly mild assumptions about each. In contrast, the work of Donoho et al. [43] detects only if a pair of streams is correlated. This is equivalent to our goal if one assumes non-attacking pairs are perfectly uncorrelated, but that is not necessarily realistic and our assumptions about non-attacking pairs will allow for substantial coarse-grained correlation among them. For example, if co-workers work and take breaks together, their typing behavior may be correlated at a coarse-grained level even though they are not part of any attack. Our models allow for this type of behavior on the part of "normal" streams, and yet we will still be able to distinguish them from true stepping-stone attacks.

Fewer assumptions: We make very mild assumptions, especially in comparison with previous work. For example, unlike the work by Donoho et al., our algorithm and analysis do not rely on the Poisson or Pareto distribution assumption on the behavior of the *attacking* streams. By modeling a non-attack stream as a sequence of Poisson processes with varying rates and over varying time periods, our analysis results can apply to almost any distribution or pattern of usage of non-attack and attack streams. This model allows for substantial high-level correlation among non-attackers.

Provable bounds: We give the first algorithm for detecting stepping-stone attacks that provides (a) provable bounds on the number of packets needed to confidently detect and identify stepping-stone streams, and (b) provable guarantees on false positive rates. Our bounds on the number of packets needed for confident detection are only quadratic in terms of certain natural parameters of the problem, which indicates the efficiency of our algorithm.

Stronger results with chaff: We also propose detection algorithms and give a hardness result when the attacker inserts "chaff" traffic in the stepping-stone streams. Our analysis shows that our detection algorithm is effective when the attacker inserts chaff that is less than a certain threshold fraction. Our hardness results indicate that when the attacker can insert chaff that is more than a certain threshold fraction, the attacker can make the attacking streams mimic two independent random processes, and thus completely evade any detection algorithm. Note that our hardness analysis will apply even when the monitor can actively manipulate the timing delay. Our results on the chaff case are also a significant advance from previous work. The work of Donoho et al. [43] assumes that the chaff traffic inserted by the attacker is a Poisson process independent from the non-chaff traffic in the attacking stream, while our results make no assumption on the distribution of the chaff traffic.

The type of guarantee we will be able to achieve is that given a confidence parameter δ, our procedure will certify a pair as attacking or non-attacking with error probability at most δ, after observing a number of packets that is only quadratic in certain natural parameters of the problem and logarithmic in $1/\delta$. Our approach is based on a connection to sample-complexity bounds in Computational Learning Theory. In that setting, one has a set or sequence of hypotheses $h_1, h_2, \ldots$, and the goal is to identify which if any of them has a low true error rate from observing performance on random examples [68, 113, 18]. The type of question addressed in that literature is how much data does one need to observe in order to ensure at most some given δ probability of failure. In our setting, to some extent packets play the role of examples and pairs of streams play the role of hypotheses, though the analogy is not perfect because it is the relationship *between* packets that provides the information we use for stepping-stone detection.

The high-level idea of our approach is that if we consider two packet streams and look at the *difference* between the number of packets sent on them, then this quantity is performing some type of random walk on the one-dimensional line. If these streams are part of a stepping-stone attack, then by the maximum-tolerable delay assumption, this quantity will never deviate too far from the origin. However, if the two streams are *not* part of an attack, then even if the streams are somewhat correlated, say because they are Poisson with rates that vary in tandem, this walk *will* begin to experience substantial deviation from the origin. There are several subtle issues: for example, our algorithm may not know in advance what an attacker's tolerable delay is. In addition, new streams may be arriving over time, so if we want to be careful not to have false-positives, we need to adjust our confidence threshold as new streams enter the system.

Outline. In the rest of the chapter, we first discuss related work in Section 2.2, then give the problem definition in Section 2.3. We then describe the stepping-stone detection algorithm and confidence bounds analysis in Section 2.4. We consider the consequences of adding chaff in Section 2.5. We finally conclude in Section 2.6.

2.2 Related Work

The initial line of work in identifying interactive stepping stones focused on *content*-based techniques. The interactive stepping stone problem was first formulated and studied by Staniford and Heberlein [106]. They proposed a content-based algorithm that created thumbprints of streams and compared them, looking for extremely good matches. Another content-based approach, Sleepy Watermark Tracing, was proposed by Wang et al. [123]. These content-based approaches require that the content of the streams under consideration do not change significantly between the streams. Thus, for example, they do not apply to encrypted traffic such as SSH sessions.

Another line of work studies correlation of streams based on *connection timings*. Zhang and Paxson [128] proposed an algorithm for encrypted connection chains based on periods of activity of the connections. They observed that in stepping stones, the ON-periods and OFF-periods will coincide. They use this observation to detect stepping stones, by examining the number of consecutive OFF-periods and the distance of the OFF-periods. Yoda and Etoh [127] proposed a deviation-based algorithm to trace the connection chains of intruders. They computed deviations between a known intruder stream and all other concurrent streams on the Internet, compared the packets of streams which have small deviations from the intruder's stream, and utilize these analyses to identify a set of streams that match the intruder stream. Wang et al. [122] proposed another timing-based approach that uses the arrival and departure times of packets to correlate connections in real-time. They showed that the inter-packet timing characteristics are preserved across many router hops, and often uniquely identify the correlations between connections. These algorithms based on connection timings, however, are all vulnerable to active timing pertubation by the attacker – they will not be able to detect stepping stones when the attacker actively perturbs the timings of the packets on the stepping-stone streams.

We are aware of only two papers [43, 121] that study the problem of detecting stepping-stone attacks on encrypted streams with the assumption of a bound on the maximum delay tolerated by the attacker. In Section 2.1, we discuss the work of Donoho et al. [43] in relation to our work. We note that their work does not give any bounds on the number of packets needed to detect correlation between streams, or a discussion of the false positives that may be identified by their method. Wang and Reeves [121] proposed a watermark-based scheme, which can detect correlation between streams of encrypted packets. However, they assume that the attacker's timing perturbation of packets is independent and identically distributed (*iid*). Our algorithms do not require such an assumption. Further, they need to actively manipulate the inter-packet delays in order to embed and detect their watermarks. In contrast, our algorithms require only passive monitoring of the arrival times of the packets.

Wang [120] examined the problem of determining the serial order of correlated connections in order to determine the intrusion path, when given the complete set of correlated connections.

2.3 Problem Definition

Our problem definition essentially mirrors that of Donoho et al. [43]. A *stream* is a sequence of packets that belong to the same connection. We assume that the attacker has a maximum delay tolerance Δ, which we may or may not know. That is, for every packet sent in the first stream, there must be a corresponding packet in the second stream between 0 and Δ time steps later. The notion of maximum delay bound was first proposed by Donoho et al. [43]. We also assume that there is a maximum number of packets that the attacker can send in a particular time interval t, which we call p_t. We note that p_Δ is unlikely to be very large, since we are considering interactive stepping-stone attacks. As in prior work, we assume that a packet on either stream maps to only one packet on the other stream (i.e., packets are not combined or broken down in any manner).

Similar to previous work, we do not pay attention to the content or the sizes of the packets, since the packets may be encrypted. We assume that the real-time traffic delay between packets is very small compared to Δ, and ignore it everywhere. We have a stepping-stone monitor that observes the streams going through the monitor, and keeps track of the total number of packets on each stream at each time of observation. We denote the total number of packets in stream i by time t as $N_i(t)$, or simply N_i if t is the current time step.

By our assumptions, for a pair of stepping-stone streams S_1, S_2, the following two conditions hold for the true packets of the streams, i.e., not including chaff packets:

1. $N_1(t) \geq N_2(t)$.
   Every packet in stream 2 comes from stream 1.

2. $N_1(t) \leq N_2(t + \Delta)$.
   All packets in stream 1 must go into stream 2 — i.e., no packets on stream 1 are lost enroute to stream 2, and all the packets on stream 1 arrive on stream 2 within time Δ.

If the attacker sends no chaff on his streams, then all the packets on a stepping stone pair will obey the above two conditions.

We will find it useful to think about the number of packets in a stream in terms of the total number of the packets observed in the union of two streams: in other words, viewing each arrival of a packet in the union of the two streams as a "time step". We will use $\overline{N}_i(w)$ for the number of packets in stream i, when there are a total of w packets in the union of the two streams.

In Section 2.4.1, we assume that a normal stream i is generated by a Poisson process with a constant rate λ_i. In Section 2.4.2, we generalize this, allowing for substantial high-level correlation

Table 2.1: Summary of notation

Δ	maximum tolerable delay bound
p_Δ	maximum number of packets that may be sent in time interval Δ.
δ	false positive probability
S_i	stream i
M	number of packets that we need to observe on the union of the two streams in the detection algorithms
$N_i(t)$	number of packets sent on stream i in time interval t.
$\overline{N}_i(w)$	number of packets sent on stream i when a total of w packets is present on the union of the pair of stream under consideration.

between non-attacking streams. Specifically, we model a non-attacking stream as a “Poisson process with a knob”, where the knob controls the rate of the process and can be adjusted arbitrarily by the user with time. That is, the stream is really generated by a sequence of Poisson processes with varying rates for varying lengths of time. Even if two non-attacking streams correlate by adjusting their knobs together — e.g., both having a high rate at certain times and low rates at others — our procedure will nonetheless (with high probability) not be fooled into falsely tagging them as an attacking pair.

The guarantees produced by our algorithm will be described by two quantities:

- a *monitoring time* M measured in terms of total number of packets that need to be observed on both streams, before deciding whether the pair of streams is an attack pair, and

- a *false-positive probability* δ, given as input to the algorithm (also called the confidence level), that describes our willingness to falsely accuse a non-attacking pair.

The guarantees we will achieve are that (a) any stepping-stone pair will be discovered after M packets, and (b) any normal pair has at most a δ chance of being falsely accused. Our algorithm will never fail to flag a true attacking pair, so long as at least M packets are observed. For instance, our first result, Theorem 2.1, is that if non-attacking streams are Poisson, then $M = 2(p_\Delta + 1)^2 \log \frac{1}{\delta}$ packets are sufficient to detect a stepping-stone attack with false-positive probability δ. One can also adjust the confidence level with the number of pairs of streams being monitored, to ensure at most a δ chance of *ever* falsely accusing a normal pair.

All logarithms in this chapter are base 2. Table 2.1 summarizes the notation we use in this chapter.

2.4 Main Results: Detection Algorithms and Confidence Bounds

In this section, we give an algorithm that will detect stepping stones with a low probability of false positives. We only consider streams that have no chaff, which means that every packet on the second stream comes from the first stream, and packets can only be delayed, not dropped. We will discuss the consequences of adding chaff in Section 2.5.

Our guarantees give a bound on the number of packets that need to be observed to confidently identify an attacker. These bounds have a quadratic dependence on the maximum tolerable delay Δ (or more precisely, on the number of packets p_Δ an attacker can send in that time frame), and a logarithmic dependence on $1/\delta$, where δ is the desired false-positive probability. The quadratic dependence on maximum tolerable delay comes essentially from the fact that on average it takes $\Theta(p^2)$ steps for a random walk to reach distance p from the origin. Our basic bounds assume the value of p_Δ is given to the algorithm (Theorems 2.1 and 2.2); we then show how to remove this assumption, increasing the monitoring time by only an $O(\log\log p_\Delta)$ factor (Theorem 2.3).

We begin in Section 2.4.1 by considering a simple model of normal streams — we assume that any normal stream S_i can be modeled as a Poisson process, with a fixed Poisson rate λ_i. We then generalize this model in Section 2.4.2. We make no additional assumptions on the attacking streams.

2.4.1 A Simple Poisson Model

We first describe our detection algorithm and analysis for the case that p_Δ is known, and then later show how this assumption can be removed.

The Detection Algorithm

Our algorithm is simple and efficient: for a given pair of streams, the monitor watches the packet arrivals, and counts packets on both streams until the total number of packets (on both streams) reaches a certain threshold $2(p_\Delta + 1)^2$.[1] If in this time, the difference in the number of packets of the two streams ever exceeds the packet bound p_Δ, we know the streams are normal; otherwise, the monitor restarts. If the difference stays bounded for a sufficiently long time ($\log \frac{1}{\delta}$ such trials of $2(p_\Delta + 1)^2$ packets), the monitor declares that the pair of streams is a stepping stone. The algorithm is shown in Fig. 2.1.

We note that the algorithm is memory-efficient — we only need to keep track of the number of packets seen on each stream. We also note that the algorithm does not need to know or compute the Poisson rates; it simply needs to observe the packets coming in on the streams.

[1] The intuition for the parameters as well as the proof of correctness is in the analysis section.

```
DETECT-ATTACKS $(\delta, p_\Delta)$
    Set $m = \log \frac{1}{\delta}$, $n = 2(p_\Delta + 1)^2$.
    For $m$ iterations
        For $w = 1$ to $n$ packets observed on $S_1 \cup S_2$.
            Compute $d(w) = N_1(w) - N_2(w)$
            If $|d(w)| > p_\Delta$ return NORMAL.
        Reset $N_1 = N_2 = 0$.
    return ATTACK.
```

Figure 2.1: Algorithm for stepping-stone detection (without chaff) with a simple Poisson model

Analysis

We first note that, by design, *our algorithm will always identify a stepping-stone pair, providing they send M packets*. We then show that the false positive rate of δ is also achieved by the algorithm. Under the assumption that normal streams may be modeled as Poisson processes, we show three analytical results in the following analysis:

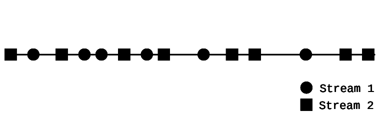
1. When p_Δ is known, the monitor needs to observe no more than $M = 2(p_\Delta + 1)^2 \log \frac{1}{\delta}$ packets on the union of the two streams under consideration, to guarantee a false positive rate of δ for any given pair of streams (Theorem 2.1).

2. Suppose instead that we wish to achieve a δ probability of false positive over *all* pairs of streams that we examine. For instance, we may wish to achieve a false positive rate of δ over an entire day of observations, rather than over a particular number of streams. When p_Δ is known, the monitor needs to observe no more than $M = 2(p_\Delta + 1)^2 \log \frac{i(i+1)}{\delta}$ packets on the union of the ith pair of streams, to guarantee a δ chance of false positive among all pairs of streams it examines (Theorem 2.2).

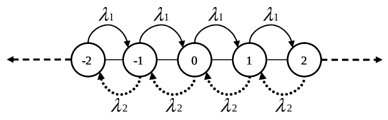
3. When p_Δ is unknown, we can achieve the above guarantees with only an $O(\log \log p_\Delta)$ factor increase in the number of additional packets that need to observe (Theorem 2.3).

Below, we first give some intuition and then the detailed theorem statements and analysis.

Intuition We first give some intuition behind the analysis. Consider two normal streams as Poisson processes with rates λ_1 and λ_2. We can treat the difference between two Poisson processes as a random walk, as shown in Fig. 2.2. Consider a sequence of packets generated in the union of the two streams. The probability that a particular packet is generated by the first stream is $\frac{\lambda_1}{\lambda_1+\lambda_2}$ (which we denote μ_1), and probability that it is generated by the second stream is $\frac{\lambda_2}{\lambda_1+\lambda_2}$ (which we call μ_2). We can define a random variable Z to be the difference between the number of packets generated by the streams. Every time a packet is sent on either S_1 or S_2, Z increases by 1 with



(a)



(b)

Figure 2.2: (a) Packets arriving in the two streams. (b) Viewing the arrival of packets as a random walk with rates λ_1 and λ_2.

probability μ_1, and decreases by 1 with probability μ_2. It is therefore a one-dimensional random walk. Assuming that our observation of the random walk begins at some unknown position x, we care about the expected time for Z to exit the bounded region $[x - p_\Delta, x + p_\Delta]$. Without loss of generality, we may take $x = 0$. Then, if $|Z| > p_\Delta$, the delay bound has to be violated for some packet.

Theorem 2.1. *Under the assumption that normal streams behave as Poisson processes, the algorithm* DETECT-ATTACKS *will correctly detect stepping-stone attacks with a false positive probability at most δ for any given pair of streams, after monitoring $2(p_\Delta + 1)^2 \log \frac{1}{\delta}$ packets on the union of the two streams.*

Proof: Let $Z = N_1(w) - N_2(w)$. We first bound the probability that Z of n packets. Let T be the time taken for a one-dimensional random walk starting the origin to reach $p_\Delta + 1$ or $-p_\Delta - 1$ for the first time. Then, as in Feller[48], for a fair random walk,

$$E[T] = (p_\Delta + 1)^2.$$

For a biased random walk starting at the origin, $E[T]$ is always strictly less than $(p_\Delta + 1)^2$.

By Markov's inequality,

$$Pr[T \geq 2(p_\Delta + 1)^2] \leq \frac{1}{2}.$$

Thus, the probability that Z remains in the interval $[-p_\Delta, p_\Delta]$ throughout the arrival of n packets on the union of the streams is bounded by $\frac{1}{2}$.

To ensure that this is bounded by the given confidence level, we take m such observations of n time steps, so that $\left(\frac{1}{2}\right)^m \leq \delta$, or

$$m \quad \geq \quad \log \frac{1}{\delta} \;.$$

We need to observe m sets of n packets; therefore, we need $\log \frac{1}{\delta}$ intervals. □ ■

We have just shown in Theorem 2.1 that our algorithm in Fig. 2.1 will identify any given stepping-stone pair correctly, and will have a probability δ of a false positive for any given non-attacking pair of streams. We can also modify our algorithm so that it only has a probability δ of a false positive among *all* the pairs of streams that we observe. That is, given δ, we distribute it over all the pairs of streams that we can observe, by allowing only $\frac{\delta}{i(i+1)}$ probability of false positive for the ith pair of streams, and using the fact that $\sum_{i=1}^{\infty} \frac{\delta}{i(i+1)} = \delta$. To see why this might be useful, suppose $\delta = 0.001$. Then, we would expect to falsely accuse one pair out of every 1000 pairs of (normal) streams. It could be more useful at times to be able to give a false positive rate of 0.001 over an entire month of observations, rather than give that rate over a particular number of streams.

Theorem 2.2. *Under the assumption that normal streams behave as Poisson processes, the algorithm* DETECT-ATTACKS *will have a probability at most δ of a false positive among all the pairs of streams it examines if, for the ith pair of streams, it uses a monitoring time of* $2(p_\Delta + 1)^2 \log \frac{i(i+1)}{\delta}$ *packets.*

Proof: We need to split our allowed false positives δ among all the pairs we will observe; however, since we do not know the number of pairs in advance, we do not split the δ evenly.

Instead, we allow the ith pair of streams a false positive probability of $\frac{\delta}{i(i+1)}$, and then use the previous algorithm with the updated false positive level. The result then follows from Theorem 2.1 and the fact that $\sum_{i=1}^{\infty} \frac{\delta}{i(i+1)} = \delta$. □ ■

The arguments so far assume that the algorithm *knows* the quantity p_Δ. We now remove this assumption by using a “guess and double” strategy. Let $p_j = 2^j - 1$. When a pair of streams is “cleared” as not being a stepping-stone attack with respect to p_j, we then consider it with respect to p_{j+1}. By setting the error parameters appropriately, we can maintain the guarantee that any normal pair is falsely accused with probability at most δ, while guaranteeing that any attacking pair will be discovered with a monitoring time that depends only on the *actual* value of p_Δ. Thus, we can still obtain strong guarantees. In addition, even though this algorithm “never” finishes monitoring a normal pair of streams, the time between steps at which the monitor compares the difference $N_1 - N_2$ increases over the sequence. This means that for the streams that have been under consideration for a long period of time, the monitor tests differences less often, and thus does not need to do substantial work, so long as the stream counters are running continuously.

Theorem 2.3. *Assume that normal streams behave as Poisson processes. Then, even if p_Δ is unknown, we can use algorithm* DETECT-ATTACKS *as a subroutine and have a false positive probability at most δ, while correctly catching stepping-stone attacks within $O(p_\Delta^2(\log\log p_\Delta + \log \frac{1}{\delta}))$ packets, where p_Δ is the* actual *maximum value of $N_1(t) - N_2(t)$ for the attacker.*

Proof: As discussed above, we run DETECT-ATTACKS using a sequence of “p_Δ” values p_j, where $p_j = 2^j - 1$, incrementing j when the algorithm returns NORMAL. As in Theorem 2.2, we use $\frac{\delta}{j(j+1)}$ as our false-positive probability on iteration j, which guarantees having at most a δ false-positive probability overall. We now need to calculate the monitoring time. For a given attacking pair, the number of packets needed to catch it is at most:

$$\sum_{j=1}^{\lceil \log p_\Delta \rceil} 2 \cdot 2^{2j} \log \frac{j(j+1)}{\delta}.$$

Since the entries in the summation are more than doubling with j, the sum is at most twice the value of its largest term, and so the total monitoring time is $O(p_\Delta^2(\log\log p_\Delta + \log\frac{1}{\delta}))$. $\square$ $\blacksquare$

2.4.2 Generalizing the Poisson Model

We now relax the assumption that a normal process is Poisson with a fixed rate λ. Instead, we assume that a normal process can be modeled as a sequence of Poisson processes, with varying rates, and over varying time periods. From the point of view of our algorithm, one can view this as a Poisson process with a user-adjustable "knob" that is being controlled by an adversary to fool us into making a false accusation.

Note that this is a general model; we could use it to coarsely approximate almost any distribution, or pattern of usage. For example, at a high level, this model could approximately simulate Pareto distributions which are thought to be a good model for users' typing patterns [96], by using a Pareto distribution to choose our Poisson rates for varying time periods, which could be arbitrarily small. Correlated users can be modeled as having the same sequence of Poisson rates and time intervals: for example, co-workers may work together and take short or long breaks together.

Formally, for a given pair of streams, we will assume the first stream is a sequence given by (λ_{11}, t_{11}), (λ_{12}, t_{12}), $\ldots$, and the second stream by (λ_{21}, t_{21}), $(\lambda_{22}, t_{22}), \ldots$. Let $N_i(t)$ denote the number of packets sent in stream i by time t. Then, the key to the argument is that over any given time interval T, the number of packets sent by stream i is distributed according to a Poisson process with a single rate $\hat{\lambda}_{i,T}$, which is the weighted mean of the rates of all the Poisson processes during that time. That is, if time interval T contains a sequence of time intervals $j_{start}, \ldots, j_{end}$, then $\hat{\lambda}_{i,T} = \frac{1}{|T|}\sum_{j=j_{start}}^{j_{end}} \lambda_{ij}t_{ij}$ (breaking intervals if necessary to match the boundaries of T).

Theorem 2.4. *Assuming that normal streams behave as sequences of Poisson processes, the algorithm* DETECT-ATTACKS *will have a false positive rate of at most δ, if it observes at least $\frac{7}{2}\log\frac{1}{\delta}$ intervals of n packets each, where $n = 8(p_\Delta + 1)^2$.*

Proof: Let $S(t)$ be the number of packets on the union of the streams at time t. Let $D(t)$ be the difference in the number of packets at time t, i.e. $N_1(t) - N_2(t)$. Let $\hat{n} = 2(p_\Delta + 1)^2$. Let $\mathcal{E}_t$ be the event that at some time $t' \leq t$ the quantity $|D(t')|$ exceeded p_Δ. We define T to be the time when $Pr[S(T) \geq \hat{n}] = \frac{1}{2}$. Then, from the proof of Theorem 2.1, for fixed T, $\hat{n} = 2(p_\Delta + 1)^2$, we know

$$\begin{aligned} Pr[\neg\mathcal{E}_T | S(T) \geq \hat{n}] &\leq \frac{1}{2}. \\ \text{Therefore, } Pr[\mathcal{E}_T] &\geq Pr[\mathcal{E}_T | S(T) \geq \hat{n}] Pr[S(T) \geq \hat{n}] \\ &\geq \frac{1}{2}(1 - \frac{1}{2}) = \frac{1}{4}. \end{aligned}$$

By definition, for all $t > T$, $Pr[\mathcal{E}_t] \geq Pr[\mathcal{E}_T]$, which, by the above, is at least $\frac{1}{4}$.

However, our algorithm does not know T; it can only observe the number of packets that appear on the streams. We therefore have to estimate the probability that the time T has passed when we have observed, say, $k\hat{n}$ packets on the union of the streams, for some suitable k. In other words, if we have observed $k\hat{n}$ packets at time t, we need to estimate the probability that $t \geq T$, in addition to the event $\mathcal{E}_T$ that we want.

We define t_n to be the time t when the number of packets on the union of the streams is n (i.e., $S(t_n) = n$).

Note that, by definition of T, we have $Pr[S(T) \geq k\hat{n}] \leq \frac{1}{2^k}$. So, $Pr[t_{k\hat{n}} < T] \leq \frac{1}{2^k}$. Setting $k = 4$, $Pr[t_{4\hat{n}} < T] \leq \frac{1}{16}$. Therefore,

$$Pr[t_{4\hat{n}} \geq T] \leq \frac{15}{16}.$$

Therefore,

$$Pr[\mathcal{E}_T \wedge (t_{4\hat{n}} \geq T)] = 1 - Pr[\neg\mathcal{E}_T \vee (t_{4\hat{n}} \geq T)] \geq 1 - (\frac{3}{4} + \frac{1}{16}) = \frac{3}{16}.$$

Note that $Pr[\mathcal{E}_T \wedge (t_{4\hat{n}} \geq T)] \leq Pr[\mathcal{E}_{t_{4\hat{n}}}]$, therefore,

$$Pr[\mathcal{E}_{t_{4\hat{n}}}] \geq \frac{3}{16}.$$

Thus, $Pr[\neg\mathcal{E}_{t_{4\hat{n}}}] \leq \frac{13}{16}$.

To bound this quantity by the given confidence level, we need to take m such observations of $4\hat{n}$ packets in the union of the streams, so that:

$$\begin{aligned} \left(1 - \frac{3}{16}\right)^m &\leq \delta. \\ m &\geq \frac{\log\frac{1}{\delta}}{\log\left(\frac{16}{13}\right)}. \end{aligned}$$

Since $\frac{1}{\log\left(\frac{16}{13}\right)} < \frac{7}{2}$, we set $m \geq \frac{7}{2}\log\frac{1}{\delta}$.

■

Likewise, we have the analogues of Theorem 2.2 and Theorem 2.3 for the general model. We omit their proofs, since they are very similar to the proofs of Theorem 2.2 and Theorem 2.3.

Theorem 2.5. *Assuming that normal streams behave as sequences of Poisson processes, the algorithm* DETECT-ATTACKS *will have a probability at most δ of a false positive over all pairs of streams it examines, if, for the ith pair of streams, it observes $\frac{7}{2}\log\frac{i(i+1)}{\delta}$ intervals of n packets each, where $n = 8(p_\Delta + 1)^2$.*

Theorem 2.6. *Assuming that normal streams behave as sequences of Poisson processes, then if p_Δ is unknown, we can use repeated-doubling and incur an extra $O(\log\log p_\Delta)$ factor in the number of packets over that in Theorem 2.5, to achieve false-positive probability δ.*

2.5 Chaff: Detection and Hardness Result

All the results in Section 2.4 rely on the attacker streams obeying two assumptions in Section 2.3 — in a pair of attacker streams, every packet sent on the first stream arrives on the second stream, and any packet that arrives on the second stream arrives from the first stream. In this section, we examine the consequences of relaxing these assumptions.

Notice that only the packets that must reach the target need to obey these two assumptions. However, the attacker could insert some superfluous packets into either of the two streams, that do not need to reach the target, and therefore, do not have to obey the assumptions. Such extraneous packets are called *chaff*. By introducing chaff into the streams, the attacker would try to ensure that the number of packets observed in his two streams appear less correlated, and thus reduce the chances of being detected.

Donoho et al. [43] also examine the consequences of the addition of chaff to attack streams. They show that under the assumption that the chaff in the streams is generated by a Poisson process that is independent of the non-chaff packets in the stepping-stone streams, it is possible to detect correlation between stepping-stone pairs, as long as the streams have sufficient packets. However, an attacker may not wish to generate chaff as a Poisson process. In this section, we assume that a clever attacker will want to optimize his use of chaff, instead of adding it randomly to the streams. In Section 2.5.1 we explain how to detect stepping stones using our algorithm when the attacker uses a limited amount of chaff (Theorem 2.7). In Section 2.5.2 we describe how an attacker could use chaff to make a pair of stepping-stone streams mimic two independent Poisson processes, and thus ensure that the pair of streams are not correlated. We then give upper bounds on the minimum chaff the attacker needs to do this (Theorems 2.8 and 2.9).

2.5.1 Algorithm for Detection with Chaff

Recall that our algorithm DETECT-ATTACKS is based on the observation that, with high probability, two independent Poisson processes will differ by any fixed distance given sufficient time. An attacker can, therefore, evade detection with our algorithm by introducing a sufficient difference between the streams all the time. Specifically, our algorithm checks if the two streams have a difference that is greater than p_Δ packets every time either stream gets a packet, until there are $2(p_\Delta + 1)^2$ packets in the union of the streams. To evade our algorithm as it stands (in Fig. 2.1), all that the attacker might need to do is to send one packet of chaff on the faster stream.

```
DETECT-ATTACKS-CHAFF $(\delta, p_\Delta)$
    Set $m = \log \frac{1}{\delta}$, $n = 8(p_\Delta + 1)^2$.
    For $m$ iterations
        For $w = 1$ to $n$ packets observed on $S_1 \cup S_2$.
            Compute $d(w) = N_1(w) - N_2(w)$
            If $|d(w)| > 2p_\Delta$ return NORMAL.
        Reset $N_1 = N_2 = 0$. NORMAL.
    return ATTACK.
```

Figure 2.3: Algorithm for stepping-stone detection with fewer than p_Δ packets of chaff every $8(p_\Delta + 1)^2$ packets.

Algorithm

We now modify DETECT-ATTACKS slightly, to detect stepping-stone attacks under a limited amount of chaff. Instead of waiting for the difference to exceed p_Δ packets between the two streams, we could wait for the difference to exceed $2p_\Delta$ packets. The independent Poisson processes would eventually get a difference of $2p_\Delta + 1$, but now, the attacker would need to send more than p_Δ packets in chaff in order to evade detection. He could get away with exactly $p_\Delta + 1$ packets if he sends all of the chaff packets in the same time interval, on the same stream. However, as long as he sends fewer than p_Δ packets of chaff in every time interval, the monitor will flag his streams as stepping stones.[2] The complete algorithm is shown in Fig. 2.3.

Analysis

We now show that DETECT-ATTACKS-CHAFF will correctly identify stepping stones with chaff, as long as the attacker sends no more than p_Δ packets of chaff for every $8(p_\Delta + 1)^2$ packets. Further, any given non-attacking pair of streams will have no more than a δ chance of being called a stepping stone.

Theorem 2.7. *Under the assumption that normal streams behave as Poisson processes, and the attacker sends fewer than p_Δ packets of chaff every $8(p_\Delta + 1)^2$ packets, the algorithm* DETECT-ATTACKS-CHAFF *will have a false positive rate of utmost δ, if we observe $\log \frac{1}{\delta}$ intervals of $8(p_\Delta + 1)^2$ packets each.*

Proof: The analysis is similar to that of Theorem 2.1.

Let $Z = N_1(w) - N_2(w)$, and let T be the time taken for a one-dimensional random walk

[2]We choose to wait for a difference of $2p_\Delta$ packets here, because it is the integral multiple of p_Δ that maximizes the rate at which the attacker may send chaff. with the non-integral multiple of p_Δ that maximizes the rate at which the attacker must send chaff, but we omit the details here.

starting the origin to reach $2p_\Delta + 1$ or $-2p_\Delta - 1$ for the first time. Again, as in Feller[48],

$$E[T] \leq (2p_\Delta + 1)^2 \leq 4(p_\Delta + 1)^2.$$

By Markov's inequality,

$$Pr[T \geq 8(p_\Delta + 1)^2] \leq \frac{1}{2}.$$

Thus, the probability that Z remains in the interval $[-2p_\Delta, 2p_\Delta]$ throughout the arrival of n packets on the union of the streams is bounded by $\frac{1}{2}$.

On the other hand, for an attack pair with no chaff, we know that $\overline{N}_1(w) - \overline{N}_2(w) \leq p_\Delta$. When the attacker can add less than p_Δ packets of chaff in $8(p_\Delta + 1)^2$ packets, $\overline{N}_1(w+n) - \overline{N}_2(w+n) < 2p_\Delta$, and thus, difference in packet count an attack pair cannot exceed $2p_\Delta$ in n packets. $\square$ ■

Note that Theorem 2.7 is the analogue of Theorem 2.1 when the chaff rate is bounded as described above. The analogues to the other theorems in Section 2.4 can be obtained in a similar manner.

Obviously, the attacker can evade detection by sending more than p_Δ packets of chaff for every $8(p_\Delta + 1)^2$ packets. Further, if we count in pre-specified intervals, the attacker would only need to send p_Δ packets of chaff in *one* of the intervals, since the algorithm only checks if the streams differ by the specified bound in *any* of the intervals.

We could address the second problem by sampling random intervals, and checking if the difference Z in those intervals is at least $2p_\Delta$. We could also modify our algorithm to check if the difference Z stays outside $2p_\Delta$ for at least a fourth of the intervals, and analyze the resulting probabilities with Chernoff bounds. To defeat this, the attacker would have to send at least $\frac{1}{8(p_\Delta+1)}$ fraction the total packets on the union ($p_\Delta + 1$ packets of chaff every $8(p_\Delta + 1)^2$ packets) in an independent interval, so that every (sufficiently long) interval is unsuspicious.

However, if the attacker just chooses to send a lot of chaff packets on his stepping-stone streams, then he will be able to evade the algorithm we proposed. This type of evasion is, to some extent, inherent in the problem, not just the detection strategy we propose. In the next section, we show how an attacker could successfully mimic two independent streams, so that no algorithm could detect the attacker. We also give upper bounds on the minimum chaff the attacker needs to add to his streams, so that his attack streams are completely masked as independent processes.

2.5.2 Hardness Result for Detection with Chaff

If an attacker is able to send a *lot* of chaff, he can in effect ride his communication on the backs of two truly independent Poisson processes. In this section, we analyze how much chaff this would require. This gives limitations on what we could hope to detect if we do not make additional assumptions on the attacker.

Specifically, in order to simulate two independent Poisson processes exactly, the attacker could first generate two independent Poisson processes, and then send packets on his streams to match them. He needs to send chaff packets on one of the streams, when the constraints on the other stream do not allow the non-chaff packet to be forwarded to/from it. In this way, he can mimic the processes exactly, and pair of streams will not appear to be a stepping-stone pair, to any monitor watching it. Note that even if the inter-packet delays were actively manipulated by the monitor, the attacker can still mimic two independent Poisson processes, and therefore, by our definition, will be able to evade detection.

Let λ_1 be the rate of the first Poisson process, and λ_2 be the rate of the second Poisson process. In our analysis, we assume $\lambda_1 = \lambda_2 = \lambda \gg \frac{1}{\Delta}$. If $\lambda_1 \gg \lambda_2$, or $\lambda_1 \ll \lambda_2$ the attacker will need to send many more chaff packets on the faster stream, so $\lambda_1 = \lambda_2$ will be the best choice for the attacker.

We model the Poisson processes as binomials. We choose to approximate the two independent Poisson processes of rate λ as two independent binomial processes, for cleaner analysis. To generate these processes, we assume that the attacker flips two coins, each with λ bias (of getting a head), at each time step.[3] He has to send a packet (either a real packet or chaff) on a stream when its corresponding coin turns up heads, and should send nothing when the coin turn up as tails. That way, he ensures that the two streams model two independent binomial processes exactly. Since the attacker generates the independent binomial processes, he could flip coins Δ or more time steps ahead, and then decide whether a non-chaff packet can be sent across for a particular coin flip that obeys all constraints, or if it has to be chaff.

We now show how the attacker could simulate two independently-generated binomial processes with minimum chaff. First, the attacker generates two sequences of independent coin flips. The following algorithm, BOUNDED-GREEDY-MATCH, then produces a strategy that minimizes chaff for the attacker, for any pair of sequences of coin flips. Given two sequences of coin flips, the attacker matches a head in first stream at time t to the first unmatched head in the second stream in the time interval $[t, t + \Delta]$. All matched heads become real (stepping-stone) packets, and all the remaining heads become chaff. An example of the operation of the algorithm is shown in Fig. 2.5.2.

The following theorem shows that BOUNDED-GREEDY-MATCH will allow the attacker to produce the minimum amount of chaff needed, when the attacker simulates two binomial processes that were generated independently.

Theorem 2.8. *Given any pair of sequences of coin flips generated by two independent binomial processes,* BOUNDED-GREEDY-MATCH *minimizes the chaff needed for a pair of stepping-stone streams to mimic the given pair of sequences.*

Proof: Suppose not, i.e., suppose there exists a sequence pair of coin flips σ for which BOUNDED-GREEDY-MATCH is not optimal. Let S be the strategy produced by BOUNDED-GREEDY-MATCH

[3]We could, equivalently, assume that the attacker flips a coin with $\frac{\lambda}{k}$ bias k times in a time step. As $k \to \infty$, the binomial approaches a Poisson process of rate λ.

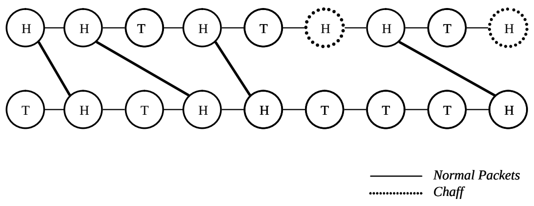


Figure 2.4: An illustration of the matching produced by the algorithm BOUNDED-GREEDY-MATCH on two given sequences, with $\Delta = 2$.

for σ. Let S' be a better matching strategy, so that $\mathit{Chaff}(S) > \mathit{Chaff}(S')$. Then there exists a head in σ such that h is matched with a head h' through S', but not through S.

Assume, wlog, that h is on the first stream at time t, and h' on the second stream. For S to be a valid match, h' should be in $[t, t + \Delta]$, and h' must be unmatched under S' to any other head. Let us suppose that h' is matched to another (earlier than t) head on the first stream under S (otherwise BOUNDED-GREEDY-MATCH would have generated a match between h and h' on S).

We track chain of the matching heads in the sequence backwards (starting from h) in this way: we take the currently matched head in one strategy, and look for the head that matches it in the other strategy. When this chain of matchings stops, we must have an unmatched head, and one of following two cases (the manner in which we trace the chain of matching heads, along with the assumption that the unmatched head h is on the first stream, implies that we find only matched heads on the second stream of S, and the first stream of S'):

- *Case 1*: The unmatched head is in stream 1 of S'. In this case, an unmatched head in S correlates with an unmatched head in S', and therefore, this particular case is not our counterexample, since each unmatched head under S will correspond to an unmatched head under S'.
- *Case 2*: The unmatched head is in stream 2 of S. In this case, we have to have reached this head (call it g_0) from its matching head g_1 in S'; we have to reach g_1 from matched head g_2 in S. Since we are tracing backwards in time, time of g_2 is greater than the time of g_0. However, since g_0 can be matched to g_1, we have a contradiction, since we are not matching the head g_1 to the earliest available head g_0, as per BOUNDED-GREEDY-MATCH.

The analysis when h is on the second stream of S is similar.

Thus, with the algorithm BOUNDED-GREEDY-MATCH, every unmatched head in S must have a corresponding unmatched head in S', therefore, $\mathit{Chaff}(S) \leq \mathit{Chaff}(S')$, creating a contradiction.

□ ■

Now we examine upper bounds on the chaff that will need to be sent by the attacker, in terms

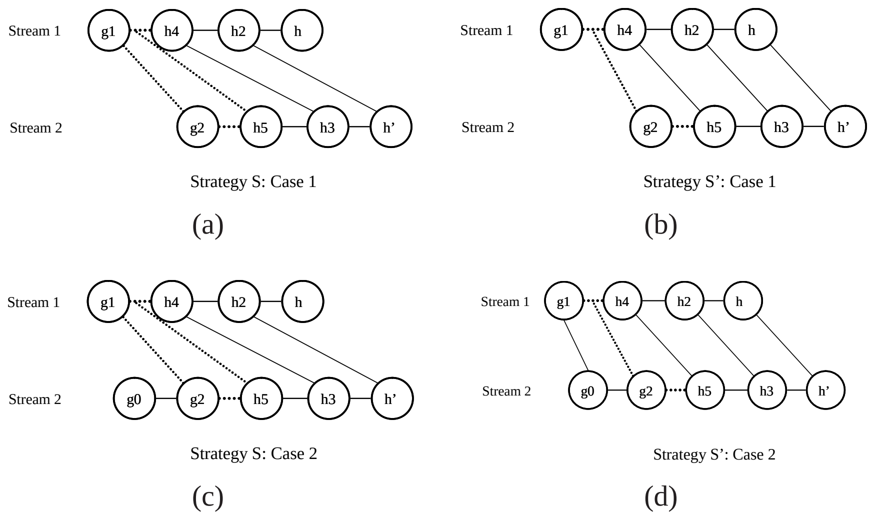


Figure 2.5: The proof of Theorem 2.8. All the figures give an illustration of how the heads are traced back. (a) and (b) show case 1 of the proof, and (c) and (d) show case 2 of the proof. By assumption, h is unmatched in S and matched in S'. h is matched to h' in the strategy S'; in S, h' is matched to $h2$; then, we look at $h2$'s match in S', call it $h3$; use $h3$ to find $h4$ in S, $h4$ to find $h5$ in S', and so on. We continue tracing the matches of heads backwards in this manner until we stop, reaching either case 1 or case 2. In case 1, $g1$ is unmatched in strategy S', and in S, $g0$ is unmatched in S, but $g1$ is not matched greedily.

of the total packets sent. We give an upper bound on the amount of chaff that the attacker must send in BOUNDED-GREEDY-MATCH. We note that our analysis shows how the attacker could do this if he mimics two independent Poisson processes, but it may not be necessary for him to do this in order to evade detection.

Theorem 2.9. *If the attacker ensures that his stepping-stone streams mimic two truly independent Poisson processes, then, under* BOUNDED-GREEDY-MATCH, *the attacker will not need to send more than* $\frac{1}{\sqrt{2\lambda\Delta - 2\sqrt{2\lambda(1-2\lambda)\Delta}}} + 0.05$ *fraction of packets as chaff in expectation, when the Poisson rates of the streams are equal with rate* λ.

Proof: We divide the total time (coin flips) into intervals that are Δ long, and examine the expected difference in one of these intervals. Notice that for the packets that are within a specific Δ interval, matches are not dependent on the times when they were generated. (i.e., any pair of packets in this interval is no further than Δ apart in time, and therefore, could be made a valid match). Many more packets than this can be matched, across the interval boundaries, but this gives us an easy upper bound.

Consider the packets in the union of the two streams in this interval. Each packet in this union can also be considered as though it were generated from a (different) unbiased coin, with heads as stream 1 and tails as stream 2; once again, we have a uniform random walk. Since every head can be matched to any available tail, the amount of chaff is the expected (absolute) difference in the number of heads and tails. Call this difference Z, and the packets on the union of the streams X. X is then a binomial with parameters 2λ, and Δ. Therefore, $E[X] = 2\lambda\Delta$. The expectation of $\frac{Z}{X}$ is then the following:

$$\begin{aligned} E[\frac{Z}{X}] &= \sum_x \frac{1}{x} E[Z|X=x]P(X=x) \\ &= \sum_x \frac{1}{\sqrt{x}} P(X=x) \\ &\leq 0.05 + \frac{1}{\sqrt{2\lambda\Delta - 2\sigma}}, \text{ where } \sigma = \sqrt{2\lambda(1-2\lambda)\Delta}. \end{aligned}$$

Since every interval of size Δ is identical, the attacker needs to send $\frac{1}{\sqrt{2\lambda\Delta - 2\sqrt{2\lambda(1-2\lambda)\Delta}}} + 0.05$ fraction of packets as chaff, at most, in expectation. □ ■

2.6 Conclusion

In this chapter, we have proposed and analyzed algorithms for stepping-stone detection using techniques from Computational Learning Theory and the analysis of random walks. Our results are the first to achieve provable (polynomial) upper bounds on the number of packets needed to confidently detect and identify encrypted stepping-stone streams with proven guarantees on the probability of falsely accusing non-attacking pairs. Moreover, our methods and analysis rely on very mild assumptions, especially in comparison with previous work. We also examine the consequences when the attacker inserts chaff into the stepping-stone traffic, and give bounds on the amount of chaff that an attacker would have to send to evade detection. Our results are based on a new approach which can detect correlation of streams at a fine-grained level. Our approach may apply to more generalized traffic analysis domains, such as anonymous communication.

The results presented in this chapter are joint work with Avrim Blum and Dawn Song, and have previously appeared at the 7^th International Sympoisum on Recent Advances in Intrusion Detection, 2004 [17].

Chapter 3

Limits of Signature-Generation with Learning-based Algorithms

3.1 Introduction

It is well-known that automatic signature generation is important – often, there may be small time-windows between when a vulnerability is discovered, and when fast-spreading exploits that target it appear. Generating signatures manually is slow and error-prone, and thus, we need automatic signature generation.

However, there are some requirements for the generated signatures to be useful. These signatures need to identify most of the exploits (have low false negatives), and falsely identify very few non-exploits (have low false positives). They need to capture properties of exploits that do not appear in normal traffic. They also need to be efficient so that signatures can be generated quickly. These requirements make automatic signature generation a hard problem.

A major line of research effort has focused on finding signatures using *pattern*-based analysis, *i.e.*, by extracting byte patterns that uniquely distinguish exploits using network traffic statistics [70, 104, 71, 91, 77]. Such *pattern-extraction algorithms* are attractive because the signatures can be efficiently generated and matched. Pattern-extraction algorithms are, at core, machine learning algorithms: they use a pool of data containing exploits and normal traffic (called the *training pool*), and look for *invariant* byte strings that are present across all exploit packets, but do not occur in the normal traffic. Earlier work has shown that such distinguishing invariants exist, even when the payloads are self-encrypting, e.g., even in polymorphic worms, the high-order bits of the return address of buffer overflows and protocol framing bytes are found to be invariant[91, 77]. This research has led to interest in how pattern-extraction algorithms could be attacked or evaded [97, 92, 61].

In this work, we show fundamental limits on the accuracy of a large class of pattern-extraction algorithms in an adversarial setting. We formulate a framework that allows unified analysis of all such pattern-extraction algorithms, and show lower bounds on the mistakes all pattern-extraction algorithms need to make under some common assumptions, by showing how to adapt results from learning theory. At a high level, our results show that algorithms for pattern-extraction signature-generation can be forced into making a significant number of false positives or false negatives. Earlier work on the limitations of pattern-extraction algorithms have focused on individual algorithms and specific systems. For example, Perdisci et al. [97] demonstrate if an attacker can systematically inject noise into the training pool, Polygraph [91] fails to generate good signatures. Newsome et al. [92] illustrate similar results in Paragraph even when adversary cannot inject arbitrary noise into the training pool. Our results generalize these earlier results through theoretical analysis, demonstrating that similar attacks are possible on all such algorithms, with similar assumptions.

The central conclusion of our theoretical analysis is that any pattern-extraction (and similar learning-based) algorithms could be manipulated into making a number of mistakes on arbitrary exploits, as a function of the adversary's power to add misleading information to his exploits. Because we cannot predict how future exploits would look, it is important to know (and this result shows us) when and how much pattern-extraction algorithms could be fooled. These results hold when there is a valid signature of invariants, the signature-generator uses randomized algorithms, whose output the adversary cannot predict, and even if host-monitoring techniques like taint analysis [33, 37, 93, 108] are used to identify exactly which packets are exploits and which are not.

Our results are independent of the kind of function that the algorithm tries to learn over the byte sequence, (i.e., the algorithm is allowed to learn any arbitrary complex function over the invariant bytes), or computational complexity of the algorithm. Our analysis also offers insight into algorithms that refuse to tolerate one-sided error, and the lower bounds for these algorithms are much higher than results for more general algorithms. These results show that, it is indeed much easier for an adversary to manipulate an algorithm that makes very few false positives, or very few false negatives.

Existing experimental results (Perdisci et al. [97] and Paragraph[92]) already illustrate that the assumptions for our analysis hold, at least for current families of pattern-extraction algorithms. Our results demonstrate that if pattern-extraction algorithms (and similar signature-generation algorithms) need to work in an adversarial environment, they need to be designed so that the assumptions do not hold; i.e., that the adversary cannot find a large set of byte strings, resemble the exploit's invariants in traffic frequency statistics.

We also explore when pattern-extraction algorithms (and similar signature-generation algorithms) may work. For example, if an exploit contains invariants that are never present in normal traffic, then it seems likely that the exploit can be identified. Our results show that the lower bounds of some families of algorithms are noticeably weakened, under some conditions, *i.e.,* when there is a gap between the distribution of tokens in normal traffic and the invariants of exploits. Our analysis also offers insight into the kind of algorithms that may work and highlights the importance of the function (over the invariants) that the algorithm learns: algorithms that look for a simple set of invariants (learning a simple conjunction of invariants) have far worse lower bounds than algorithms that look for more complex functions over the invariants; this implies that, unlike the results for arbitrary exploits, it is far easier for the adversary to manipulate those simple functions when there is a gap in the traffic.

Our results are also applicable to other signature-generation techniques besides pattern-extraction algorithms. If, for example, a signature-generation algorithm looks for protocol fields exceeding specific lengths, but chooses the lengths based on malicious traffic (e.g., COVERS [78]), our results would still hold. The key limitation in pattern-extraction algorithms is that the adversary can easily add patterns that are similar to exploit's invariants, and algorithm cannot distinguish between the invariants and those added by the adversary (red herrings). Our results are applicable as long as this kind of limitation holds: the adversary can embed similar properties to those invariant to the exploit, and the algorithm cannot distinguish between them.

3.2 Definitions and Overview

We now present the main definitions and assumptions that we use throughout the paper.

3.2.1 Definitions

A *signature* is a function σ that classifies a given byte sequence (or, equivalently, packet) as malicious or non-malicious, *i.e.*, $\sigma(y)$ = *Malicious*, when byte sequence y is an exploit, and $\sigma(y)$ = *Non-Malicious*, when y is benign.

A signature may be based on various properties of the byte sequence, and we denote these properties under consideration for signature generation as *attributes*. An attribute A is a function whose input is the byte sequence and output is boolean: e.g., A could be whether $aaaaa$ is present in a byte sequence. For this attribute, for byte sequence $aaaaattttt$, $A(aaaaattttt) = true$, while for byte sequence $aabbcctttttt$, $A(aabbcctttttt) = false$. A signature could then be considered a function of attributes; thus, in effect, a signature is a function over boolean conditions. e.g., if $A(y)$ and $B(y)$ are attributes, a signature could be $\sigma(y) = \{Malicious : A(y) \wedge B(y)\}$. We say that an attribute is *satisfied* if it evaluates to true.

Recall pattern-extraction algorithms look for invariant byte strings that are present in the exploit, and not in normal traffic, and these invariants are byte strings that must all be present in the exploit for it to be malicious, e.g., a signature reported by Polygraph [91] is a pair of byte strings, '\xFF\xBF' and '\x00\x00\FA' (the Lion worm exploiting the BIND TSIG vulnerability). We refer to each such byte string as a *token*. In our terminology, the attributes test for whether each of these tokens is present, and signature is a conjunction of the two attributes. We could denote this signature as $\sigma(y) = \{Malicious :$ '\xFF\xBF'$\in y \wedge$ '\x00\x00\FA' $\in y\}$.

More generally, for pattern-extraction algorithms, an attribute tests for the presence of a particular token, perhaps at a particular location. For algorithms that use more information, other kinds of attributes would also be needed, e.g., COVERS [78] considers lengths of fields, so an attribute would also represent whether a particular field in the byte sequence is longer than a specific value.

For a fixed set of attributes G, we can represent a byte sequence by the attributes in G that it satisfies, and we describe how to do so now. We define an *instance* i for a byte sequence and a set of attributes G to be a boolean m-tuple, i.e., $i \in \{0,1\}^m$, where the ith bit is 1 if the ith property holds true for the byte sequence. An instance thus is a representation of a byte sequence for a set of attributes. So, if G consists of the two attributes "Is $\{aaaaa\}$ present?" and "Is $\{bbbbb\}$ present?", the byte sequence $aaaaaxxxxbbbb$ would be represented as $(1,1)$, and $cccccxxxxxbbbb$ would be represented by $(0,1)$. The *instance space* is the set of all instances $I = \{i \in \{0,1\}^m\}$. For the rest of this paper, we consider the set of possible attributes G to be fixed. Every byte sequence is represented as a vector in the instance space, and our discussion will be in this instance space $\{0,1\}^m$.

We also introduce some machine learning terminology. The algorithm is given some *training* data, which consists of malicious and non-malicious instances, along with the *labels* of each instance, so that it knows which instances are malicious and which are not. The algorithm finds a *hypothesis*, a function that classifies a given instance as malicious or non-malicious.

We define the *true signature* to be the signature that achieves 0 false positives and 0 false negatives, on any set of instances presented to it. In learning terminology, the true signature is the *target hypothesis* that needs to be found by the learning algorithm. Once the space of attributes is fixed, we assume there is only one true signature; of course, there may be multiple functions to represent this true signature.(If there are multiple signatures that can always achieve 0 false positives and false negatives, then the bounds apply for each signature, so this assumption is not limiting.) We refer to the attributes in the true signature as *critical attributes*.

3.2.2 Overview of Learning Framework

The central question that we want to answer is the following: to what extent can an adversary force every learning algorithm to learn the signature slowly? We answer this question by presenting lower bounds on the algorithm's performance. To do so, we need some assumptions, and in this section we describe and justify some basic assumptions we use. Our goal in choosing these assumptions is to give the algorithms in the signature generator as much power as possible. The lower bounds show that, even so, the adversary can evade detection for a long time.

We focus on four different assumptions here: the learning model, the form of the true signature, the label correctness, and the adversary's knowledge of the algorithms.

Learning model: Our analysis assumes that the algorithm is allowed to update its internal state after each batch of data that it sees, and these updates may be made over all of the data accumulated so far. For example, if the algorithm has 100 packets in its initial training pool, and then gets 50 packets in the next batch, the algorithm may update its signature after seeing the second batch, and may then use all 150 packets to generate the updated signature.

This is a little different from the typical machine learning setting, where the algorithm is given a large batch of training data, allowed to learn a function over it, and then tested on new testing data. However, since we have a malicious adversary who controls part of the data and aims to delay learning, the adversary could ensure that, without updates, the algorithm never learns a good signature. By allowing updates, algorithm might find a good signature over a longer period of time. We can also perform a more informative analysis about how the algorithm's performance evolves with more data over time. The learning algorithm still does get an initial training pool, which can contain any number of malicious and non-malicious samples, as long as the assumption of Section 3 is obeyed.

In addition, since the adversary wants the algorithm to make as many errors as possible, the adversary aims to release information about the true signature as slowly as possible. The adversary can present information about exactly one new instance in each batch (e.g., all the malicious instances in the batch can be the same, so a mistake on the instance would cause a 100% false negative). [1] In effect, it is as if the algorithm gets one new instance at a time, classifies it, and updates its internal state based on that instance. Our bounds will be in terms of the number of mistakes the algorithm makes in this setting, which also corresponds to the number of updates it requires. In the learning theory literature, this is known as the mistake-bound model [79].

Form of the true signature: We assume that the true signature of the exploit is a conjunction of attributes – i.e., all attributes in the conjunction must be satisfied by the packet. We do so because conjunctions are the simplest form of signatures that have been historically considered, and lower bounds for conjunctions imply lower bounds for more complex functions that can represent conjunctions. For example, these lower bounds are also lower bounds for regular expressions, because regular expression signatures can represent conjunctions.

However, we do not make any assumption on the form of hypotheses chosen by the algorithms for its internal state. The algorithm could use, for example, a weighted combination of tokens as its classifier. The learning algorithm is *not* required to learn a conjunction.

Label Correctness: We assume that every label given to the algorithm is correct. This means the adversary is forced to be truthful, and cannot decide to change the signature (target hypothesis) after the algorithm has been given data. This affords the algorithm a lot of power: it is as if the algorithm has an oracle like a dynamic runtime checker, and can test each input on it. If the adversary can lie, by adding carefully crafted noise for some instances, or change the target, the lower bounds would only increase.

Adversary knowledge: We assume that the adversary knows the kinds of attributes considered by the algorithm in question (and thus knows the instance space I). In some of our bounds (for

[1] Indeed, if the algorithm can update its hypothesis only every batch, it is optimal for the adversary to present exactly one instance at a time.

deterministic algorithms), the adversary needs to know the algorithm he aims to mistrain, but this is not required for the bounds on the randomized algorithms. These randomized bounds hold when the adversary (a) knows the algorithm, but has no access to its private randomness, (b) does not know the algorithm, or (c) does not know the parameters (e.g., statistical algorithms using soft decision boundaries).

3.3 Reflecting Set

In this section we describe the formal framework we will use to analyze limitations on learning-based signature generation. Our key assumption is that the adversary has the ability to construct *reflecting sets*: spurious attributes (e.g., tokens) that, to the learning algorithm, look at least as plausible apriori as the actual attributes in the signature. These take the role of the ”concept class” in learning theory, and the larger the set, the stronger the limitation. Below, we motivate the notion of reflecting sets and give formal definitions, focusing on pattern-extraction algorithms, especially Polygraph [91] & Hamsa [77].

3.3.1 Motivation & Definitions

We begin by observing a common property of many strategies proposed to evade detection by pattern-extraction algorithms. A wide range of strategies have been proposed for evasion, and all of them succeed because the adversary can increase the number of tokens that resemble the tokens critical to the signature. For example, in red herring attacks [91], the attacker adds spurious tokens to the true signature, and the attack succeeds when the algorithm mistakenly considers those as part of the true signature. Likewise, in noise injection attacks [97], allergy attacks [61] and suspicious/innocuous pool poisoning attacks [92], the adversary manipulates the token distribution in the training or testing pool, by adding well-crafted (malicious or normal) packets with carefully chosen tokens, and changing the distributions of various tokens in the training pool. Here again, how effective the attack is depends on how much the attacker can change the tokens considered by the algorithm.

Thus, these attacks succeed when the algorithm is unable to apriori distinguish between the tokens critical to the true signature, and any spurious tokens that happen to resemble these critical tokens. The attacker forces the algorithm to fail by carefully increasing the appropriate resemblance between the critical and spurious tokens, and he may be able to do this for other kinds of attributes as well. We will use the term *reflecting sets* to describe these sets of resembling attributes, as each attribute within a reflecting set appears to reflect all of the other attributes in that set.

Definition: We now define reflecting sets formally: Let S denote the true signature for an exploit, and let A denote a signature generation algorithm. Let $Pr_A[S']$ denote the probability that A gives to the function S' being the true signature. Let $C_1, C_2, \ldots C_j$ be sets of attributes, such that the signature S contains an attribute s_i in each C_i. Let $\mathcal{T}$ be the set of functions obtained by choosing one or more attributes $c_i \in C_i$, to replace the corresponding property s_i in S. Let W_m and W_{nm} be the malicious and non-malicious instances seen by the algorithm so far, and let $W = W_m \cup W_{nm}$. Let $\mathcal{T}_W$ be the set of functions in $\mathcal{T}$ consistent with W. If $Pr_A[T] = Pr_A[T']$, for any pair of functions $T, T' \in \mathcal{T}_W$, for all W, then the sets $C_1, C_2 \ldots C_j$ are *reflecting sets* for the signature S and the algorithm A.

Thus, from the point of the view of the algorithm, it is as if any combination of attributes, as long as one is picked from each reflecting set, could be the true signature even after analysis over all of the training data. If $\mathcal{T}$ denotes the set of all combinations of attributes that includes one from each reflecting set, then *to the algorithm*, the true signature S appears to be drawn at random from $\mathcal{T}$.

An additional aspect of this definition is that reflecting sets are specific to an algorithm (or a family of algorithms). We define the reflecting sets this way because different algorithms could use different aspects of possible attributes to identify a likely signature, and therefore, a reflecting set for one algorithm may not be a reflecting set for another algorithm. For example, the conjunctions algorithm of Polygraph uses every infrequent token that appears in all of the malicious instances as its signature. For this algorithm, a reflecting set is very easily constructed by simply adding more infrequent tokens to all of the malicious instances. Such a simple reflecting set, however, would not work for other algorithms, e.g., the naive Bayes algorithm in Polygraph, or Hamsa's algorithm.

Learning with Reflecting Sets: In this paper, we analyze the problem of learning a signature with a malicious adversary as the following: for every critical attribute, the adversary may include the respective reflecting set in the packets (normal or malicious, as needed). The goal is to find the true signature by identifying the critical attributes, isolating them from their reflecting sets. We define the problem formally in the next section.

3.3.2 Finding Reflecting Sets

The results of this paper are applicable to algorithms where it is possible/easy for the adversary to construct reflecting sets (or sets with a bias away from the true signature) for the attributes in the true signature, e.g., pattern-extraction algorithms. In general, this could be done for algorithms that require information from (adversarially-generated) exploits, but cannot identify the true cause of the exploit, and therefore, the attribute or parameter they learn can be forged by the attacker.

It is also not strictly necessary for all the attributes in the reflecting set to have identical traffic statistics: the goal is to capture the algorithm's inability to distinguish between different attributes inside the set, and therefore, unable to bias any selection towards the true signature. If the reflecting sets are chosen so that the algorithm's choice of signature is *less* likely to be the true signature, then the lower bounds would only increase, e.g., Hamsa's algorithm prefers tokens with the smallest frequency in normal traffic pool, and the attack suggested adds spurious tokens that are even less frequent, and therefore, would cause the algorithm to make at least as many mistakes.

The quantitative bounds on algorithms' errors are related to the size of the reflecting sets that can be found for the attributes. The size of any particular reflecting set depends on the nature of the exploit (e.g., its distinguishing properties, the protocols applicable), the adversary's ability to manipulate the training and testing pool, and the kinds of signatures that the algorithm aims to learn for it. The adversary may craft these reflecting sets either by explicitly including selected attributes in the malicious instances, or sending specific types of instances in the training data. Because the adversary crafts the reflecting set for the signature generators, the adversary knows the reflecting set.

Earlier experimental work (e.g., in Paragraph) has demonstrated that reflecting sets can be found for current generations of pattern-extraction algorithms. Further, polymorphic blending attacks [50] suggest that it may be possible to find such reflecting sets for many pattern-extraction algorithms, as long as the algorithms use byte-based traffic statistics for finding the priors of the critical tokens in the signature (e.g. [119]). We believe it would be typically possible to find reflecting sets for pattern-extraction algorithms in general, especially those which use the traffic statistics of individual tokens, due to the heavy-tailed nature of normal traffic distribution, i.e., if tokens in signatures typically consist of combinations that are rare in normal traffic, and the distribution of token combinations in normal traffic is heavy-tailed, then it is likely that an adversary will be able to easily find reflecting sets consisting of rare token combinations.

3.4 General Adversarial Model

In this section, we consider a general adversarial setting, and we present impossibility results on learning algorithms that generate signatures in this model.

3.4.1 Learning Model

We present our analysis in the mistake-bound model of learning. As described in Section 3.2.2, we choose this model because it affords the algorithm significant power, but even with this power, the adversary can delay signature generation. In this model, the algorithm gets an initial training pool (of any size), and then gets one instance at a time to classify, classifies it as malicious or non-malicious, and is then told the correct label of the instance. The algorithm then updates its hypothesis. The algorithm's goal is to converge to the true signature while minimizing the mistakes made.

Each instance given to the learning algorithm is an m-tuple boolean vector, i.e., a point in $\{0,1\}^m$. The true signature, or target hypothesis, is a conjunction of n attributes: an instance must contain all n attributes to be malicious. As discussed in Section 3.3, we assume the adversary can find a reflecting set C_i of size k for each critical attribute i, and the algorithm cannot distinguish between the attributes inside C_i. It may, however, be able to distinguish between attributes in different reflecting sets, and we need to account for this in the lower bounds. Thus, the set of all valid hypotheses H is the set of all conjunctions containing an attribute from each reflecting set; thus $|H| = k^n$. We refer to the n bits in the true signature as the *target bits*. Because the adversary crafts the malicious data, he can ensure that even with an initial training pool, no information is released about the critical attribute, to distinguish it in its reflecting set. The total number of attributes $m = nk$, the product of the number of critical attributes and the size of each reflecting set.

Our bounds are in terms of the number of mistakes made by the algorithm. The mistakes made can be interpreted as the number of updates required to converge to the true signature, when the algorithm receives the correct label right away. The mistakes in this model imply false positives and negatives in the standard batch setting: a mistake on a malicious instance is a false negative, and a mistake on a non-malicious instance is a false positive. The exact false positive and negative rate that a mistake (or a sequence of mistakes) causes depends on the specific algorithm, but a worst-case estimate on any particular batch can be seen: whenever the algorithm makes a mistake, the adversary can generate a distribution that causes a 100% false negative rate (for a malicious instance), or potentially a large false positive rate (for a normal instance).

There are two ways in which a target hypothesis can be chosen for the lower-bounds analysis. The adversary can choose the target hypothesis from the set H, or nature picks the target at random from the set H, and the adversary knows the target hypothesis selected. Lower bounds for the second way of choosing the target clearly imply lower bounds for the first.

Representation of Hypothesis Even though the target hypothesis is a conjunction of the target bits, there is no requirement that the learning algorithm learn a conjunction of the target bits. That is, the learning algorithm is free to choose any function, as long as it agrees with the target hypothesis on all the instances seen.

Formally, let $x_1, \ldots, x_m$ denote m bits of an instance, where $x_j = \{0, 1\}$. In this context, a conjunction hypothesis is a function $x_a \wedge x_b \wedge \ldots x_r$, for some r values, and evaluates to true if all bits $x_a \ldots x_r$ are 1. A *linear separator* hypothesis is a function of the form $\sum_{i \in [1,m]} w_i x_i > q$ where the weights $w_i \in \Re$. All instances that satisfy the condition (i.e., weighted combinations of bits exceeds the threshold q) evaluate to true.

The *representation* of the hypothesis is the type of function learnt by the algorithm, e.g. a linear separator or conjunction. Polygraph uses both conjunctions and linear separators, and Hamsa uses conjunctions. The results in this section are independent of the hypothesis representation chosen.

3.4.2 Results

We now present our results in the learning model described above. Each of these lower bounds can be derived from more general results in learning theory; our proofs show an explicit construction of instances that achieve the bounds for our setting. In the proof of each theorem, we show a sequence of instances for which any algorithm must achieve the stated mistake-bound. [2]

We first present bounds on the overall number of mistakes that any deterministic or randomized algorithm could be forced to make. Theorem 3.1 shows that every deterministic algorithm, regardless of what it learns, could be forced to make at least $n \log \frac{m}{n}$ mistakes by an adversary – thus, the mistakes grow linearly in the size of the signature, but only logarithmically in the size of the reflecting sets $k = \frac{m}{n}$.

Theorem 3.1. (Deterministic Algorithms) *For every deterministic algorithm, an adversary can generate a sequence of instances such that the algorithm is forced to make at least $n \log k$ mistakes, where k is the size of the reflecting sets.*

Before presenting the proof, we discuss some common elements of all proofs in this section. As described in Sec. 3.2.2, the adversary is in control of the malicious instances presented in the training data. The adversary's goal is for the algorithm to learn as little as possible, and make many mistakes. Thus, it is optimal for the adversary to generate all malicious instances identically, so that each instance contains all k attributes of every reflection set. Note that this does not conflict with our assumption that there is no noise in the training data, or that the adversary is required to be truthful.

Formally, in the instance space $\{I = i \in \{0, 1\}^m\}$, the argument above says that adversary gives the algorithm many copies of the instance $i = \{1, 1, \ldots 1\}$ in the training pool. For example, in red-herring attacks on pattern-extraction signature generators, this instance can be thought of the initial input given to the learning algorithms: all initial instances contain all red herrings as well as the invariants. Note that the target hypothesis is selected at adversarially or at random from the set H, and all hypotheses in H are indistinguishable to the algorithm. This implies that the adversary can ensure that the algorithm gains no additional information about the target bits from the training data.

Proof: Our proof is an application of the bounds proven in [79] to our setting. For completeness, we present the whole proof here to illustrate the sequence of instances that the attacker can present, in order to force the mistakes indicated in the lower bound.

Let us assume that the algorithm can divide bits into sets that correspond to a critical attribute and its reflections. By definition of reflection, the algorithm cannot distinguish between these k properties, even if the algorithm is powerful enough to distinguish properties into n sets of k.

[2]The adversary does not require knowledge of the algorithm's behaviour to generate the next instance. He would for the lower bounds on deterministic algorithms, but the bounds for the randomized algorithms apply even if the algorithm is a blackbox to the adversary.

We show a sequence of instances that force the algorithm to make $\log k$ mistakes, for a single reflecting set of k properties. Since there are n such sets, and no reflecting set can provide information about targets in any other reflecting set, using this strategy on each of will generate $n \log k$ mistakes.

With knowledge of the deterministic algorithm, the adversary can decide where to place the target bit in the reflecting set as follows: the adversary chooses a set of t bits. If the algorithm labels it positive, it places the target bit in the other set of $k - t$ bits, otherwise it places it in this set. By observing the actions of the algorithm, the adversary has chosen a set of min(t, $k - t$) bits in which to place the target bit. The adversary can repeat this process until it isolates where to place the target bit.

Thus, because the algorithm is deterministic, it is equivalent to the adversary deciding which bit to choose as the target bit, rather than deciding where to place the target bit.

The adversary begins by setting $t = k/2$, i.e., it presents an instance with $k/2$ bits set. After the algorithm makes a mistake on the instance, the adversary presents an instance with $k/4$ bits from the $k/2$ bits where the target bit needs to be present. This process continues until the number of bits is reduced to 1. The ith instance presented by the adversary has $k/2^i$ bits set to 1, and forces the adversary to commit to the presence of the the target bit in one of $k/2^i$ bits, and the adversary forces a mistake on each instance. Thus, algorithm is forced to make $\log k$ mistakes on this sequence of examples.

If there are multiple critical attributes within a single reflection set, the adversary can treat this set as two separate reflection sets, each of size k, and achieve the same number of mistakes. ■

Since the bound of Theorem 3.1 scales logarithmically with the number of spurious attributes, it is natural to ask whether this lower bound is tight. The Winnow algorithm [79] achieves a bound within $n \log n$ additive factor, showing that the bound is nearly tight.

However, much of the error in the previous theorem comes from the adversary's ability to predict what the algorithm would do next. A common solution is to allow the algorithm to use randomization. Theorem 3.2 analyzes the number of mistakes made if the algorithm is randomized (or equivalently, unknown) to the adversary. It shows that even if the signature generator uses a randomized algorithm, the algorithm can be forced to generate a lot of mistakes in expectation, half the mistakes of the deterministic case.

Theorem 3.2. (Randomized Algorithms) *For any randomized algorithm, an adversary can generate a sequence of instances so that the algorithm will make, in expectation, at least* $\frac{1}{2} n \log \frac{m}{n}$ *mistakes, where* k *is the size of the reflecting sets.*

Proof: Our proof is an application of the bounds proven in [79] to our setting. For completeness, we present the whole proof here to illustrate the sequence of instances that the attacker can present, in order to force the mistakes indicated in the lower bound.

The proof is similar to Theorem 3.1. The initial instance presented by the adversary, as before, contains all attributes, or equivalently, all m bits set to 1. As before, the algorithm may be sufficiently powerful to distinguish the reflecting sets, but it cannot identify the critical attributes within each reflecting set.

The sequence of instances that the adversary presents is similar, but chosen randomly. For each reflecting set, the adversary does the following before the algorithm identifies the target bit. The adversary chooses a set of $k/2$ bits at random from all sets of $k/2$ bits, and presents an instance with this set of $k/2$ bits. Then, with probability 1/2, the target bit is present in this set. The probability that any decision given by the algorithm on this randomly chosen set of $k/2$ bits is correct is $1/2$. Thus, the algorithm has a chance of $1/2$ of making a mistake on this step.

Once the label is given by the adversary, the information about the target bit is reduced to $k/2$, as in the deterministic case. The adversary then picks a set of size $k/4$ from the set of size $k/2$ containing the target bit. This continues until the target bit is isolated, which takes $\log k$ steps. Thus, the algorithm has an expected error of $\frac{1}{2}\log k$.

For every reflecting set of size k, the algorithm will make an expected $\frac{\log k}{2}$ mistakes, and so the total expected number of mistakes will be $\frac{1}{2}n\log k$, i.e., $\frac{1}{2}n\log\frac{m}{n}$. ■

Theorem 3.2 shows that an arbitrary deterministic algorithm is not too much worse than a randomized algorithm, and suggests that some deterministic algorithms may not fare too poorly. This result is, however, dependent on the nature of determinism in the algorithm. For example, one kind of extreme determinism is to guarantee no false positives or no false negatives. Such algorithms are attractive, since it seems better to have to tolerate only one kind of error.

We now consider one-sided algorithms: algorithms which are not allowed to make (many) false positives or (many) false negatives. Our results show that one-sided algorithms can be forced into making many more errors than algorithms with an arbitrary break-down of mistakes (e.g., in comparison to Theorem 3.1). Guaranteeing a small number of mistakes of either false positives or false negatives forces the algorithm to make a large number of mistakes of the other kind.

Theorem 3.3. (Bounded False Positives) *If an algorithm is not allowed to make any mistakes on non-malicious instances, there exists a sequence of instances such that it is forced to make at least* $n(k-1)$ *mistakes on malicious instances. More generally, consider an algorithm that is forced to make fewer than* t *mistakes on the non-malicious instances, for* $t \leq n$*. Then the algorithm must make at least* $(n-t)(k-1)$ *mistakes on the malicious instances.*

Proof: Our proof for the case of $t = 0$ is an application of the theorems in [9] to our problem, a restriction of the general learning problem. We extend this for $t > 0$ in our proof. For completeness, we present the whole proof here to illustrate the sequence of instances that the attacker can present, in order to force the mistakes indicated in the lower bound.

Let $t = 0$. Then, we need to show a sequence of instances such that the algorithm makes at least $n(k-1)$ mistakes. If the algorithm is allowed to make *no* mistake on non-malicious instances, it must always label an instance to be non-malicious when it is uncertain of the label of an instance.

In order to have the algorithm make a lot of mistakes, the adversary has to present a sequence of instances such that the algorithm is always forced to label it non-malicious. The adversary does this as follows: in the ith *epoch*, he picks one reflecting set C_i to focus on, and the instances presented have the bits that correspond to all the other reflecting sets (i.e., other than C_i) all set to 1 (e.g., in the first epoch, all bits that correspond to reflecting sets C_2 to C_n are always set to 1). He starts the epoch by presenting the instance with all bits in C_i set to 1, and he chooses one additional non-target bit from the current instance, and sets it to 0 to generate the instance that follows.

Thus, within an epoch, every instance received by the algorithm (subsequent to the first instance) has one fewer bit set to 1 than the previous instance. However, it does not know whether the target bit has been set to 0, by definition of the reflecting set, and therefore has to label the instance non-malicious. As the adversary does not set the target bit to 0, each instance presented to the algorithm is indeed malicious. The adversary can present $k-1$ such instances for each reflecting set, and thus, there are $n(k-1)$ mistakes made by the algorithm.

When $t \geq 1$, the algorithm may make at most t incorrect guesses on non-malicious instances. The adversary may use the same sequence of instances as described above, and because the algorithm is deterministic, the adversary knows when the algorithm will label an instance to be malicious. The adversary can then choose the target hypothesis so that, at that point, a non-malicious instance is presented, i.e., it is the target bit of the relevant reflecting set that is dropped at the instance (though all bits dropped earlier in the epoch are still non-target bits, as before). With this change, algorithm has then made a mistake on a non-malicious instance, but also knows the target bit for the relevant reflecting set, and will not make any more mistakes within that epoch. Thus, each mistake on a non-malicious instance in this sequence reveals the target bit of one reflecting set, but no information about any other reflecting set. When the algorithm is allowed t such mistakes, the adversary can force the algorithm to make at least $(n-t)(k-1)$ mistakes on the malicious instances. ■

Such large mistakes are not special to only algorithms that require a small number of false positives. Theorem 3.4 shows mistake-bounds for algorithms that must make very few false negatives. Indeed, these mistake-bounds are much larger than those in Theorem 3.3, for $kn \gg n$ (i.e., since reflecting sets may be large, but contain only one target bit each).

Theorem 3.4. (Bounded False Negatives) *If an algorithm is allowed to make no mistakes on malicious instances, an adversary can generate a sequence of instances so that the algorithm is forced to make $k^n - 1$ mistakes on non-malicious instances. More generally, consider a deterministic algorithm that is forced to make fewer than t mistakes on malicious instances, for $t < n$. Then the algorithm must make at least $k^{\frac{n}{t+1}} - 1$ mistakes on non-malicious instances.*

Proof:

Our proof for the case of $t = 0$ is an application of the theorems in [9] to our problem, a restriction of the general learning problem. We extend this for $t > 0$ in our proof. For completeness, we present the whole proof here to illustrate the sequence of instances that the attacker can present, in order to force the mistakes indicated in the lower bound.

Once again, we begin with the case of $t = 0$. Now, the adversary is allowed to make no mistakes on the malicious instances. Therefore, any time the algorithm receives an instance, it must label it malicious, unless the algorithm is certain that the instance is not malicious.

The adversary presents any non-malicious instance with n bits present, subject to the following two conditions: (1) exactly one bit is present from each reflecting set, (2) all target bits are not present in the instance (this follows by definition of a non-malicious instance). Each instance in this sequence is non-malicious, but the algorithm is forced to label it malicious: the algorithm does not have enough information to distinguish whether any particular bit present from a reflecting set is truly the target bit for the reflecting set, until a malicious instance has been presented.

More formally, let A denote the set of all instances that satisfy the above two conditions: each instance in A contains exactly n bits set to 1, and only one bit is set from each reflecting set. Let i_{mal} denote the sole malicious instance in A. The adversary presents instances $i_1, i_2, \ldots$ from $A - \{i_{mal}\}$ to the algorithm, one at a time. Define I_w to be the set of the first w instances presented, for $w < |A - \{i_{mal}\}|$. With the non-malicious instances in I_w, it is consistent for any instance in remaining in $A - I_w$ to be the malicious instance, and so the algorithm must continue to classify the next instance as malicious. Thus, the algorithm is forced to make a mistake on every instance in $A - \{i_{mal}\}$. There are $k^n - 1$ such instances, and therefore, the algorithm makes $k^n - 1$ mistakes.

A similar analysis can be applied for $0 < t < n$. We focus on $t = 1$ for simplicity. The adversary now divides the reflecting sets into two equal groups, A_1 and A_2, and each reflecting set goes into one of A_1 or A_2; so, each group A_1 and A_2 will account for $m/2$ bits. The adversary chooses instances in two phases: in the first phase, all instances set all bits from A_1 to 1, but only set one bit from each reflecting set in A_2 to 1 (so $m/2$ bits in A_1 but only $n/2$ bits in A_2). In the second phase, the roles of A_1 and A_2 are reversed: all instances set all bits from A_2 to 1, but only set one bit from each reflecting set in A_1 to 1 (so $m/2$ bits in A_2 but only $n/2$ bits in A_1). There are $k^{n/2} - 1$ non-malicious instances in each phase.

Recall that the algorithm may make at most one mistake on a malicious instance, and thus it can call an uncertain instance non-malicious at most once. Because the algorithm is deterministic, the adversary knows when the algorithm will classify an instance to be non-malicious, and chooses, ahead of time, the target hypothesis appropriately to ensure that at that point, a malicious instance can be presented.

We term the algorithm to be *non-conservative* if it labels an instance malicious in the first $k^{n/2} - 1$ instances that it sees, otherwise we term to be *conservative*. For a non-conservative algorithm, the adversary presents non-malicious instances from Phase 1 until the point where it would label an instance malicious, and then the malicious instance from Phase 1, to ensure that the algorithm makes a mistake on the malicious instance. It then presents the non-malicious instances from Phase 2 to the algorithm. With this sequence of instances, the algorithm needs to identify the $n/2$ target bits in Phase 2 without making any mistakes on the malicious instances, and every hypothesis is consistent with the instances and labels presented in Phase 1. Thus, the mistake-bound of Phase 2 reduces to the case of $t = 0$, with a target hypothesis of size $n/2$, and so a non-conservative algorithm makes at least $k^{n/2} - 1$ mistakes.

The analysis for a conservative algorithm is similar, except that the malicious instance is presented at the appropriate point in Phase 2. Thus, this algorithm has to make at least $k^{n/2} - 1$ mistakes on the non-malicious instances in Phase 1. The analysis when $t < n$ is also similar, the adversary simply divides the reflecting sets into $\frac{n}{t+1}$ groups, instead of dividing it into 2 groups, when $t = 1$. ■

We note briefly that lower bounds in Theorems 3.3 and 3.4 may not be tight for large values of t. Nevertheless, they still serve to illustrate the effect of allowing very few false positives or false negatives.

We note also that the bounds of Theorems 3.3 and 3.4 are very different. Intuitively, the basic difference between them arises from the kind of the information that is encoded in an exploit (malicious instance), when compared to a non-exploit (non-malicious instance) packet. In Theorem 3.3, the adversary forces the algorithm to learn the exploit from only exploit information, while in Theorem 3.4, the adversary forces the algorithm to learn the exploit from only non-exploit information. As there may be far more non-exploit packets than exploits, each of which encodes very little information about the exploit, the adversary can be able to force many more errors in Theorem 3.4.

3.4.3 Practical Implications

Discussion The central conclusion of the theoretical analysis is that any pattern-extraction (and similar learning-based signature-generation) algorithms could be manipulated into making a significant number of mistakes, in terms of the total number of false positives and false negatives generated. This holds when the signature-generator uses randomized algorithms, whose output the adversary cannot predict. It holds even if host-monitoring techniques like taint analysis [35, 37, 93, 108] are used to identify exactly which packets are malicious and which are not. Our analysis suggests that these algorithms could work only when they are designed so that a large reflecting set cannot be found.

Existing experimental research has already demonstrated the feasibility of these attacks on real systems, e.g. Paragraph shows that it is feasible to add a large number of tokens to a real buffer-overflow exploit against the ATPhttp web server, and shows how this affects the detection of polymorphic worms by Polygraph and Hamsa. This disruption is caused by the sequence of the instances presented to the algorithms, so that the algorithm does not have enough information to infer the correct target. In our proofs, we show constructions of sequences of instances that force every algorithm to make a lot of mistakes.

The results also show that if pattern-extraction algorithms need to be used, an algorithm like

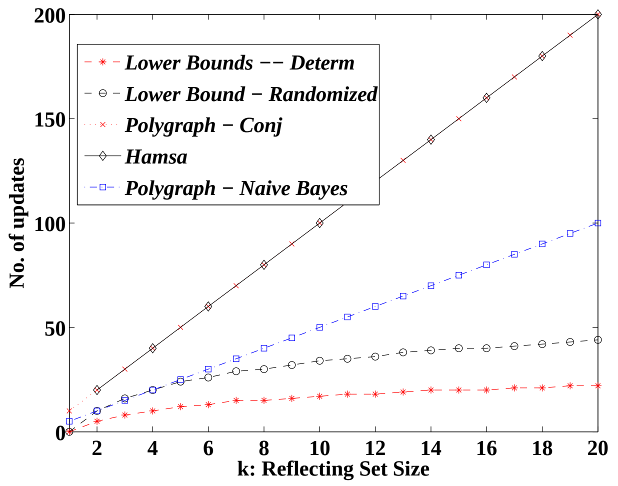


Figure 3.1: Comparison of lower bounds and current algorithms for general case: number of updates required before convergence to the true signature.

Winnow [79] may guarantee better accuracy in adversarial settings. Using a more complex algorithm would not gain a significant improvement. This is especially so these bounds are independent of the representation of the algorithms used (one could learn any arbitrary complex function over the instance space), or the algorithm's computational complexity. Still, Winnow's mistake-bound offers insight into using a more expressive representation than the basic conjunction used in several algorithms.

Lastly, our analysis offers insight into algorithms that refuse to tolerate one-sided error. The mistake bounds for these results are much higher than results for more general algorithms. These results show that, it is indeed much easier for an adversary to manipulate an algorithm that makes very few false positives, or very few false negatives. Specifically, note differences in their dependence on k, the size of the reflecting set: Theorems 3.1 and 3.2 have a logarithmic dependence on k, while Theorems 3.3 and 3.4 have a polynomial dependence on k. Thus, for an arbitrary algorithm, the adversary would not gain significantly from simply padding the malicious packets with red herrings, though he does so for one sided algorithms.

Comparison to Existing Systems As a specific illustration of the bounds, we compare lower bounds with the estimated mistakes (through calculation) that would be made by the Polygraph suite of algorithms and Hamsa. These mistakes are made when reflecting sets can be found for these pattern-extraction algorithms, but this has already been demonstrated in Paragraph.

For our comparisons, we use the attacks suggested in Paragraph for each of the algorithms. In the red herring attack suggested on Polygraph's conjunction algorithm, a mistake can be induced on every presented instance by dropping a token each time. Likewise, for the Hamsa's algorithm, a mistake can be induced for each spurious token by dropping the token with the smallest frequency at the time. On the naive Bayes algorithm, mistakes can be induced by the correlated outlier attacks shown: each malicious instance presented is crafted with tokens appearing in normal traffic, forcing the algorithm to classify it as non-malicious.

Fig. 3.1 shows the number of iterations in which mistakes would be made, as a function of the size of the reflecting set, for a signature with 10 tokens. Each of these algorithms require iterations linear in nk, where n is the number of true critical attributes in the signature, and k is the size of the reflecting set. The lower bounds, on the other hand, grow logarithmically in k. Of course, it is harder to find a reflecting set for the Bayes algorithm than the conjunction algorithm in Polygraph, and it is similarly harder to find a reflecting set for Hamsa's algorithm. For example, if there are 10 tokens in the true target signature, a reflecting set of size 10 for each token would mean that the lower bounds for deterministic algorithms require 34 updates, and the randomized algorithms require 17 updates. In contrast, the Polygraph conjunction algorithm and Hamsa's algorithm could be manipulated into requiring 100 updates.

Further, all of these calculations assume that there is an effective way to ensure that the presented instances are (later, at time of update) correctly classified, and that updates are immediate. A lag in updates would increase the number of batches seen (and therefore, mistakes seen) before converging. If, for example, the algorithm gets the correct label only every 10 iterations, the number of mistakes could increase by a factor of 10.

3.5 Exploiting Gaps in Traffic

In this section, we examine when signature-generation algorithms would work, even in the presence of adversaries, and when there may be large reflecting sets for the signatures. For example, if an exploit's invariant tokens *never* appeared in normal traffic, it ought to be possible to identify this exploit with pattern-extraction algorithms. Our goal is to understand conditions under which these learning algorithms might work, even if they must learn over properties which have reflecting sets.

The lower bounds analysis in the previous section was based on the existence of a sequence of instances (or equivalently, an adversary who generated a sequence of instances) for the algorithm to classify, and these instances could be drawn from any point in the instance space. Thus, effectively, the algorithm needed to be able to classify every single instance correctly, and was required to have a small number of mistakes on any sequence of instances. The hypothesis found by the algorithm was required to agree with the target hypothesis on every single instance in the instance space.

However, there might be situations when such a requirement is more stringent than necessary. For example, while we would certainly want the algorithm to be able to classify all malicious instances generated by the adversary, perhaps we do not need the algorithm to classify all possible non-malicious instances, unless they are regularly present in normal traffic. A reasonable goal might be to ask an algorithm to only classify correctly the non-malicious instances that are truly present in normal traffic, rather than any arbitrary combination of properties generated by an adversary. In this situation, an algorithm would need to agree with the target hypothesis only on the malicious instances, and the non-malicious instances *present in normal traffic*.

Thus, the algorithm can disagree with the target hypothesis on a region of the instance space, the region where the non-malicious instances are not present in normal traffic. In this case, it might be possible to make fewer mistakes, as a function of how large the gap between the malicious instances and the normal instances are. The analysis in Sec. 3.4 addresses the case when there is no gap between normal instances and malicious instances.

Recall that the instance space is a boolean hypercube in $\{0, 1\}^m$. The malicious instances are instances which have all n target bits set to 1, regardless of the values of the remaining $m - n$ bits. The non-malicious instances are all the remaining instances. The non-malicious instances truly present in normal traffic may be only a small subset of these instances. We need a way to quantify the region of the instance space that the algorithm does not need to classify correctly. We do this by defining how to measure the gap between the two types of traffic in Section 3.5.1. Then, in Section 3.5.2, we describe the learning model. We describe results in Section 3.5.3, and their practical implications in Section 3.5.4.

3.5.1 Defining the Gap

Intuitively, our goal is to measure how close the normal traffic is to the malicious instances, e.g., if few attributes of the malicious instances are present in the normal instances, we would like the gap to be large. Further, we would like the gap to capture some intrinsic property between the normal traffic and the malicious instances, which the adversary cannot manipulate over time. That way, we can then measure the effect of the adversary's manipulation of the malicious instances for different kinds of gap.

We measure the gap in the following manner: let Z be the set of target attributes – the attributes that must truly be present in the malicious instances (in our notation $n = |Z|$). We define the *instance-overlap* of a normal instance i to be the fraction of attributes of Z that is present in the instance. We define the *overlap-ratio* of the normal traffic to be the maximum instance-overlap of *any* instance in normal traffic. In other words, the fraction of target attributes present in a normal instance is, at most, the overlap-ratio. So, for example, an exploit whose invariant is a single token that never appears in normal traffic has overlap-ratio 0. Our definition is motivated by the observation that tokens extracted in signatures are very rare in normal traffic, and the appearance of multiple tokens together is even rarer.

3.5.2 Learning Model

The learning model in this section is similar to the one in Section 3.4, however, we need to make some crucial changes. A hypothesis is *overlap-equivalent* to the target hypothesis if the two hypotheses agree on all the malicious instances, and all non-malicious instances truly present in the normal traffic. The goal of the learning algorithm is to find an overlap-equivalent target hypothesis, when the target hypothesis is drawn at random from the set of all valid hypotheses. As in Section 3.4, we give mistake bounds for algorithms that are allowed any number of samples, and any kind of running time. However, the bounds now depend on the representation that the algorithm uses to find an overlap-equivalent hypothesis.

We use d to denote the overlap-ratio of the normal traffic distribution with the target hypothesis. The overlap-ratio also has an implication for the reflecting sets that the adversary chooses. The attributes in the reflecting sets may also need to obey the overlap-ratio, otherwise, they may not be reflecting sets for some algorithms anymore. That is, these sets may need to be chosen so that no more than d fraction of the reflected attributes from different reflecting sets can be present together in any instance in normal traffic.

3.5.3 Results

We now present lower bounds on the mistakes made in this model. Unlike the previous model, these results depend on the representation used by the algorithm, whenever the overlap-ratio $d < 1$. This is because there is always a signature that can be represented in the disjunctive normal form: the signature just looks for the presence of *any* of the k^n possible combinations of the attributes is always correct. [3] To our knowledge, this model has not been analyzed before.

We describe lower bounds for two commonly used representations: conjunctions and linear combinations of attributes, and we show lower bounds on both deterministic and randomized algorithms. Our bounds are in terms of the mistakes made on a sequence of instances *consistent* with a given overlap-ratio d: every non-malicious instance in the sequence has an instance-overlap of at most d. As these instances are consistent with overlap-ratio d, they could potentially appear in normal traffic. In other words, our theorems imply that, when the overlap-ratio is d, there exists normal traffic for which every algorithm has to make a certain number of errors (as a function of d).

Theorems 3.5 and 3.6 show lower bounds for learning conjunctions for deterministic and randomized algorithms. They show that the mistakes made by any algorithm that is forced to learn conjunctions of attributes scales linearly with the number of attributes, as well as the overlap-ratio of the normal traffic distribution.

[3] Alternately, one can consider this signature to be an OR function of the set of all valid hypotheses described in Section 3.4.1.

Theorem 3.5. (Deterministic Algorithms using Conjunctions) *Let the overlap-ratio of the normal traffic be d, and let k be the number of attributes in each reflecting set. For any d, there exists a sequence of instances consistent with overlap-ratio d such that any deterministic algorithm that learns an overlap-equivalent conjunction will need to make at least $(k-1)(dn+1)$ mistakes.*

Proof: We prove this in two parts: we first prove that any deterministic algorithm learning a conjunction with $dn+1$ bits may be forced to make a lot of mistakes, and then we show that there exists a sequence of non-malicious instances consistent with overlap-ratio d, so that the adversary can force algorithm to learn a conjunction with $dn+1$ bits.

We first show that any deterministic algorithm that learns a conjunction with $dn+1$ bits could be forced to make $(k-1)(dn+1)$ mistakes, when reflecting sets are of size k. We count only the mistakes made on malicious instances, and therefore each of instances must contain all n target bits. The adversary may generate a sequence of instances in the following manner: he starts with the malicious instance that has all bits set to 1, and in each subsequent instance, he sets one additional non-target bit (from any reflecting set) to be 0. Because the algorithm is deterministic, the adversary can choose the target hypothesis and the bit that is set to 0 at each point and ensure that the algorithm makes a mistake on each instance. This way, for each reflecting set, the algorithm will need to have $k-1$ bits set to 0 before the target bit is revealed. As this procedure can be done for each of the reflecting sets included in the learned conjunction, the algorithm makes $(k-1)(dn+1)$ mistakes.

Now, we show that all deterministic algorithms have to learn a conjunction with at least $dn+1$ attributes. To do this, we use the following definitions. We term a *block* to be all of the bits corresponding to a reflecting set. We say that a block is set to 0 if all bits in the block are 0, and that a block is set to 1 if all bits in the block are set to 1. We will term a *zero-information* instance to be one that has d blocks set to 1, and has all the remaining $n-d$ blocks set to 0. The set K is the set of all zero-information instances.

Each instance in K may appear in normal traffic: the instance contains no more than d target bits set to 1. Each instance in K is also non-informative about the true target – all bits in the reflecting set always appear simultaneously. With this set K, if the algorithm does not have at least $dn+1$ bits (each from a different reflecting set) in its conjunction at any point, it can be forced to make an error on a non-malicious instance: the adversary simply chooses non-malicious instance from K that satisfies the algorithm's conjunction, and forces it to make an error on a zero-information instance. This mistake reveals no additional information to the algorithm about the target hypothesis, and the adversary can force the algorithm to make it as long as the algorithm's conjunction has fewer than $dn+1$ bits. Thus, the algorithm makes fewer mistakes if it always learns a conjunction of size at least $dn+1$. ■

Theorem 3.6. (Randomized Algorithms using Conjunctions) *Let the overlap-ratio of the normal traffic be d, and let k be the number of attributes in each reflecting set. For any d, there exists a sequence of instances consistent with overlap-ratio d such that any randomized algorithm that learn an overlap-equivalent conjunction will make, in expectation, at least $\frac{k-1}{k}(dn+1)$ mistakes.*

Proof: The proof is similar to that of the previous theorem, but with two modifications. In our proof, we use K, the set of zero-information instances defined in the previous proof.

First, the adversary no longer knows when the conjunction used by the algorithm contains at least $dn + 1$ attributes; however, he can force the algorithm to contain such a conjunction with high probability by giving the algorithm, at random points in the sequence of instances, a non-malicious instance from K. Let $\mathcal{T}$ be the event that the algorithm uses a conjunction with fewer than $dn + 1$ attributes. For any $\epsilon > 0$, if $Pr[\mathcal{T}] > \epsilon$, an instance drawn at random from K will force the algorithm to make a mistake with probability $\epsilon / \binom{n}{dn}$. Thus, there is always a constant chance of error on the non-malicious instances if $Pr[\mathcal{T}] > \epsilon$. Therefore, if the algorithm uses a conjunction with few attributes, a long sequence of random instances drawn from K could generate, in expectation, many non-informative errors on the non-malicious instances.

Now, if the algorithm tries to find a conjunction with at least $dn+1$ attributes (and thus include attributes from at least $dn+1$ reflecting sets), it makes at least $\frac{k-1}{k}$ mistakes in expectation for each of the $(dn+1)$ attributes. As before, the adversary starts with the malicious instance that has all bits set to 1, and in each subsequent round, picks one additional non-target bit (from any reflecting set) to set to 0 in the instance that is presented. For every reflecting set that appears in the algorithm's conjunction, the algorithm has a $1/k$ chance of making a mistake when any non-target bit is set to 0. Because the adversary can do this $k-1$ times within a single reflecting set, the expected number of mistakes is $\frac{k-1}{k}$, within one set. The adversary can ensure that $dn + 1$ reflecting sets are used with probability $1 - \epsilon$, so the number of mistakes it makes, in expectation, is $(1 - \epsilon)\frac{k-1}{k}(dn + 1)$, for any $\epsilon > 0$. ■

Next we consider the minimum number of malicious instances that an adversary can send through undetected, if the learning algorithm learns linear separators. Theorems 3.7 and 3.8 show lower bounds for algorithms that need to learn overlap-equivalent linear separators.

Theorem 3.7. (Deterministic Algorithms using Linear Separators) *Let the overlap-ratio of the normal traffic be d, and let k be the number of attributes in each reflecting set. For any d, there exists a sequence of instances consistent with overlap-ratio d such that any deterministic algorithm that learns overlap-equivalent linear separators will need to make at least $\log_{1/d} k$ mistakes.*

Proof: Recall that C_i denotes the ith reflecting set. Let $U = \{C_i\}_i$. Without loss of generality, we will assume that the bits in the instance are reordered so that the first k bits correspond to the attributes in reflecting set C_1; the next k bits correspond to the attributes in the reflecting set C_2, and so on. Let $x_{i,j}$ be 1 if the jth property of the reflecting set C_i is present in the instance ($ik+j$th bit is 1 in the reordered instance) and 0 otherwise.

A linear separator that identifies malicious instances needs to be of the form $\sum_{i,j} w_{i,j}x_{i,j} > t$, where $w_{i,j}$ is a weight of token, and t is any fixed value with $t > 0$. For the proof, we will use K, the set of zero-information instances defined in the proof of Theorem 3.5. Let D be a set that contains exactly d-fraction of the reflecting sets. The adversary can then force the following constraints to hold at every point of time: for every D, $\sum_{a \in D} \sum_j w_{a,j} \leq t$. This is because if the constraints do not hold, the adversary can force the algorithm to make a mistake on a zero-information instance from K, and thus the algorithm makes a mistake that does not help in identifying the target hypothesis.

As in the proof of Theorem 3.5, we show how the attacker generates mistakes on the malicious instances with these constraints. The attacker constructs malicious instances as follows: for each reflecting set C_i, the attacker chooses the p bits with the lowest weights, and sets the malicious instance to have these p bits to be 1.

Let q_i be the sum of the weights of the p bits for a reflecting set C_i. Then, for every set D as defined earlier, $\sum_{i \in D} q_i \leq t\frac{p}{k}$. Let $\mathcal{D}$ be the set of all such sets D. Then, we have $\sum_{D \in \mathcal{D}} \sum_{i \in D} q_i \leq |\mathcal{D}| t \frac{p}{k}$. This implies that $\binom{n-1}{dn-1} \sum_{i \in U} q_i \leq \binom{n}{dn} t \frac{p}{k}$, giving $\sum_{i \in U} q_i \leq \frac{tp}{kd}$. By setting $p \leq kd$, $\sum_{i \in U} q_i \leq t$. Thus, the attacker can send a malicious instance with the appropriate p bits set, and the algorithm will make a mistake by labelling it non-malicious.

With this mistake, the attacker has reduced the size of every reflecting set to effectively be kd from the original size of k: the algorithm now knows that the target bit has to be among the kd bits that were set in the malicious instance just presented. The adversary can recurse this procedure with the new reflecting sets, until their size has effectively reduced to 1, and this allows the attacker to force $\log_{1/d} k$ mistakes, or $\log_{1/d} \frac{m}{n}$. ■

Theorem 3.8. (Randomized Algorithms using Linear Separators) *Let the overlap-ratio of the normal traffic be d, and let k be the number of attributes in each reflecting set. For any d, there exists a sequence of instances consistent with overlap-ratio d such that any randomized algorithm that learns overlap-equivalent linear separators will need to make, in expectation, at least $\frac{1}{2} \log_{1/2d} k$ mistakes to converge to a hypothesis equivalent to the target.*

Proof: The proof is similar to that of Theorem 3.7; however, we need to make two modifications, because the adversary does not always know the internal state of the algorithm. In our proof, we use K, the set of zero-information instances defined in the earlier proofs.

First, the adversary no longer knows whether the constraint $\sum_{i,j} w_{i,j} \leq t/d$ is disobeyed; however, he can force it to hold with high probability by presenting the algorithm, at randomly chosen points in sequence, a non-malicious instance from K. In particular, for any $\epsilon > 0$, if $Pr[\sum_{i,j} w_{i,j} > t/d] > \epsilon$, an instance drawn at random from K will cause the algorithm to make a mistake with probability $d\epsilon$. Thus, there is always a constant chance of error on the non-malicious instances if $Pr[\sum_{i,j} w_{i,j} > t/d] > \epsilon$ – this chance of error does not approach 0 as long as $\sum_{i,j} w_{i,j} > t/d$. Therefore, if $\sum_{i,j} w_{i,j} > t/d$, an arbitrarily long sequence of random instances drawn from K could generate, in expectation, arbitrarily many non-informative errors on the non-malicious instances.

The second modification needed is that the adversary constructs a slightly different sequence of malicious instances present to the algorithm. In this situation, the adversary cannot pick the p smallest weights, since the adversary does not know the p smallest weights. Instead, the adversary picks $p/2$ weights at random, from each reflecting set, and constructs an instance with those bits

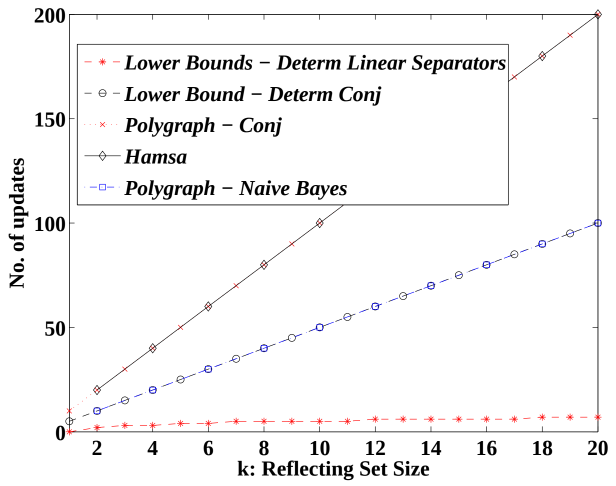


Figure 3.2: Comparison of lower bounds and algorithms when there is a large gap between the normal traffic and malicious samples: no. of updates required before converging to true signature, as a function of reflecting set

set to 1, and the rest set to 0. The probability that these $np/2$ weights exceed tp/kd is at most 1/2, which means that the probability that an mistake is caused is at least 1/2, by Markov's inequality. Thus, at each step, the algorithm makes a mistake with probability 1/2, and the number of attributes in each reflecting set reduces to $kd/2$. Thus, the algorithm makes $1/2 \log_{1/2d} \frac{m}{n}$ mistakes on the malicious instances in expectation, when the constraint $\sum_{i,j} w_{i,j} > t/d$ holds. As this can be set to hold with probability $1 - \epsilon$, for any $\epsilon > 0$, the expected number of mistakes is at least $\frac{1}{2}(1-\epsilon)\log_{1/2d} \frac{m}{n}$. ■

Since these lower bounds are representation-dependent, they cannot be directly compared to the ones in Section 3.4.2. However, the results for learning overlap-equivalent linear separators are comparable to the lower bounds of Theorems 3.1 and 3.2: we know that the lower bounds of Theorems 3.1 and 3.2 are tight, and the Winnow algorithm learns a linear separator. We note that $d = \frac{n}{n-1}$, these lower bounds approach those of Section 3.4.2. Thus, when the gap between the normal traffic and malicious exploits are large, it may be possible to learn with few mistakes.

3.5.4 Practical Implications

The results of this section suggest that pattern-extraction (and similar signature-generation) algorithms would work in practice for some kinds of exploits – they would work better when the overlap between tokens present in normal traffic and exploits is large. Our analysis suggests an easy way of quickly quantifying the exploits such algorithms may work well for. In addition, it highlights the importance of choosing an appropriate representation to learn from: even if all signatures are conjunctions of tokens (attributes), choosing a more flexible representation like linear separators allows the adversary fewer ways to manipulate the algorithm's behaviour.

Fig. 3.2 also shows that the representations chosen by the learning algorithm determine its accuracy significantly. It shows the number of updates required to learn the true signature, when there are 10 tokens in the true signature, and the overlap-ratio is 0.5 (i.e., any normal instance has at most half the critical tokens). When the reflecting set is 10, algorithms learning conjunctions still require 50 mistakes, while those learning linear separators require only 5. Conjunctions are easy for an adversary to manipulate, and therefore can be forced to make far many more errors than linear separators. The errors on linear separators also illustrate the extent to which the bounds are weakened with a gap in traffic: the corresponding mistake-bound for arbitrary exploits is about 30.

Of course, normal traffic is difficult to model and may undergo rapid changes. It may be difficult to tell what the overlap-ratio of an exploit might be, and how likely it is to change, and of course, one cannot predict the overlap-ratios of future exploits. Further, the data captured to find the overlap-ratio might not be sufficient to identify a very rare token, and one might think that the overlap-ratio is smaller than it truly is, which would cause false positives. However, if normal traffic continues to originate from the same kind of distribution as the data captured, such false positives are likely to be few and infrequent.

3.6 Related Work

As we have discussed pattern-extraction signature-generation algorithms throughout this paper, we do not discuss them further here. Signature generation algorithms that use semantic information have taken many different directions; some examples to point at directions are [21, 73, 117, 38, 34]. As it is not immediately clear when reflecting sets would exist if semantic information is used, our results may not apply to these algorithms. A notable exception is COVERS [78], that uses protocol semantics, but generates a property that can be manipulated by the adversary.

We next discuss prior attacks on pattern-extraction algorithms. Perdisci et al. [97] showed that if the adversary could add malicious noise to suspicious pool and the normal pool, Polygraph fails to generate good signatures. Paragraph [92], demonstrates that even with a truthful adversary, Polygraph and Hamsa [77] are vulnerable to attacks. Allergy attacks, forcing many false positives and DoS against the network, also demonstrated on Polygraph and Hamsa [27]. However, these papers demonstrate attacks on specific algorithms and systems, while our work shows general lower bounds. Gundy et al [61] present a different kind of attack showing that polymorphic worms do not need to have invariant bytes. Our work differs as it shows lower bounds even when there invariant bytes. A related attack on intrusion detection systems are the polymorphic blending attacks by Fogla et al. [50]. These attacks match all byte frequency statistics of normal traffic under consideration by an IDS, and thus evade detection. This is different from our situation, as we do already have the appropriate target attributes under consideration, and these do uniquely identify the exploit. Our work is also complementary to that of Crandall et al [36], as their work explores the extent to which pattern-based signatures may need to be present at all in the packets containing exploits. Our work shows that even if they are present, it is quite easy for the adversary to mislead signature generators.

Finally, we discuss related work in learning in adversarial settings. The learning theory community has explored theoretical questions on learning with malicious adversaries and malicious noise [69, 10, 23]. In this regard, the most related work is mentioned in Sec. 3.4. Experimentally, there have been a few studies on learning adverserially. Lowd and Meek [82] study the problem of an adversary reverse engineering classifiers, and show applications to reverse-engineering spam filters [95]. Dalvi et al. [39] present a game-theoretic analysis of how an algorithm and adversary could adapt to each other, and show applications to spam filtering. Barreno et al. [15] examine when machine learning could be more secure at a more general level, presenting a framework, and a lower bound on the work that an attacker must to evade an IDS. However, none of this work is directly applicable to our problem.

3.7 Conclusion

We have shown fundamental limits on the accuracy of a large class of pattern-extraction algorithms in an adversarial setting. Our work generalizes earlier work on attacks which have focused on individual algorithms and current systems. We also analyzed and shown conditions under which pattern-extraction may work. Our results are applicable to other kinds of signature-generation algorithms that use easily forgeable properties of an exploit.

The results presented in this chapter are joint work with Avrim Blum and Dawn Song, and have previously appeared at the $15^t h$ Annual Network and Distributed Systems Security Symposium, 2008 [114].

Chapter 4

Streaming Algorithms for Fast Detection of Superspreaders

4.1 Introduction

Internet attacks such as distributed denial-of-service (DDoS) attacks and worm attacks are increasing in severity. Network security monitoring can play an important role in defending against and mitigating such large-scale Internet attacks – it can be used to detect drastic traffic pattern changes that may indicate attacks or, more actively, to identify misbehaving hosts or victims being attacked, in order to throttle attack traffic automatically.

For example, a compromised host doing fast scanning for worm propagation often makes an unusually high number of connections to distinct destinations within a short time. The Slammer worm, for instance, caused some infected hosts to send up to $26,000$ scans a second [88]. We call such a host a *superspreader*. (Note that a superspreader may also be known as a port scanner in certain cases.) By identifying in real-time any source IP address that makes an unusually high number of distinct connections within a short time, a network monitoring point can identify hosts that may be superspreaders and take appropriate action. For example, the identified potential attackers (and victims) can be used to trigger the network logging system to log attacker traffic for detailed real-time and post-mortem analysis of attacks, in order to throttle subsequent (similar) attack traffic in real-time.

In this chapter, we study the problem of identifying *superspreaders*. A superspreader is defined to be a host that contacts at least a given number of distinct destinations within a short time period. Superspreaders could be responsible for fast worm propagation, so detecting them early is of paramount importance. Thus, given a sequence of packets, we would like to design an efficient monitoring algorithm to identify in real-time which source IP addresses have contacted a high number of distinct hosts within a time window.

$$(s1, d1), (s2, d2), (s1, d1), (s3, d3), (s1, d1), (s2, d3), (s4, d1), (s2, d4), (s1, d1), (s5, d4), (s6, d6)$$

Figure 4.1: Example stream of (source, destination) pairs, starting with $(s1, d1)$ and ending with $(s6, d6)$.

Note that a superspreader is different from the usual definition of a heavy-hitter ([55, 26, 46, 84, 41, 67]). A heavy-hitter might be a source that sends a lot of packets, and thus exceeds a certain threshold of the total traffic. A superspreader, on the other hand, is a source that contacts many *distinct* destinations. So, for instance, a source that is involved in a few extremely large file transfers may be a heavy-hitter, but is not a superspreader. On the other hand, a source that sends a single packet to many destinations might not create enough traffic to be a heavy-hitter, even if it is a superspreader – some of the sources in our traces that are superspreaders create less than 0.004% of the total traffic analyzed; heavy-hitters typically involve a significantly higher fraction of the traffic.

It is desirable to be able to do the monitoring on high-speed links, for example, on a large enterprise network or an ISP network for a large number of home users. A major difficulty with detecting superspreaders on a high-speed monitoring point is that the traffic volume on high speed links can be tens of gigabits per second and can contain millions of flows per minute. In addition, within such a great number of flows and high volume of traffic, most of the flows may be normal flows. The attack traffic may be an extremely small portion of the total traffic. Many traditional approaches require the network monitoring points to maintain per-flow state. Keeping per-flow state, however, often requires high memory storage, and hence is not practical for high speed links. We need, therefore, efficient algorithms to find superspreaders that use memory sparingly.

The superspreader problem is an instance of a more general problem that we term *heavy distinct-hitters*, which may be formulated as follows: given a stream of (x, y) pairs, find all the x's that are paired with a large number of distinct y's. Figure 4.1, for example, depicts a stream where source $s2$ is paired with three distinct destinations, whereas all other sources in the stream are paired with only one distinct destination; thus $s2$ is a heavy distinct-hitter for this (short) stream.

An algorithm for the heavy distinct-hitters problem has a wide range of networking applications. Clearly, we can solve the dual of the superspreader problem – finding the destinations which are contacted by a large number of sources – and such destinations could be victims of DDoS attacks. It can be used to identify which port has a high number of distinct destinations or distinct source-destination pairs without keeping per-port information and thus aid in detection of attacks such as worm propagation. Such a port is a heavy distinct-hitter in our setting (x is the port and y is the destination or source-destination pair). Such an algorithm can also be used to identify which port has high ICMP traffic, which often indicates high scanning activity and scanning worm propagation, without keeping per-port information. For example, spammers often send the same emails to many distinct destinations within a short period, and we could identify potential spammers without keeping information for every sender. An algorithm for the heavy distinct-hitter problem may also be useful in peer-to-peer networks, where it could be used to find nodes that talk to a lot of other nodes without keeping per-node information. For simplicity, in the rest of this chapter, we will describe our algorithms for identifying superspreaders. The algorithms can be easily applied to the other applications mentioned above.

To summarize, the contributions of our work are the following:

- We propose new streaming algorithms for identifying *superspreaders*. Our algorithms are the first to address this problem efficiently and provide proven accuracy and performance bounds. The best previous approaches [47, 124] require a certain amount of memory to be allocated for each source [47] or each flow [124] within the time window; we do not keep state for every source, and thus our algorithms scale very well. We present two algorithms: the first, a simpler one, which is already much better than existing approaches, and which we use for base comparison; and the second, a more complex two-level filtering scheme, that is more space-efficient on commonly-seen distributions. In addition, the two-level filtering scheme may have other applications and be of independent interest.

- We also propose several extensions to enhance our algorithms – we extend our algorithms to scenarios when deletion is allowed in the stream (Section 4.4.1), to the sliding window scenario (Section 4.4.2), and we propose efficient distributed versions of our algorithms (Section 4.4.3). The deletion scenario is especially well-motivated – it can be used to find sources that have a large number of distinct connection failures (this may be an indication of scanning behavior), rather than just sources that contact a large number of distinct destinations. That is, once the network monitoring point sees a response from a destination for a connection from a source, that source-destination pair gets deleted from the count of the number of distinct connections a source makes.

- Our experimental results on traces with up to 10 million flows confirm our theoretical results. Further, they show that the memory usage of our algorithms is substantially smaller than alternative approaches. Finally, we study the effect of different superspreader thresholds on the performance of the algorithms, again confirming the theoretical analysis.

Note that our contribution is in the proposal of new streaming algorithms to enable efficient network monitoring for attack detection and defense, when *given* certain parameters. Selecting and testing the correct parameters, however, is application-dependent and outside of the scope of this thesis.

Note that we cannot detect a malicious host that spoofs IP addresses and contacts many destinations, since the algorithms will only operate on the input (src, dst) pairs. It is, however, difficult to engage in TCP-based attacks with IP spoofing. Also, we may need special care when identifying the connection direction. We can handle this issue in TCP traffic by checking for superspreaders only in the SYN packets. In UDP traffic, though, this may not be possible, because we may not be able to distinguish which of the two hosts sent the first packet without extra storage. Thus, in UDP traffic, we may not be able to distinguish between a superspreader and a source that simply *responds* to many clients. (In our abstraction, the latter is also a superspreader in case of UDP). In practice, though, we expect that most sources that typically need to respond to many clients will remain more or less constant over brief periods of time (e.g. web servers over a few days' time), and that it will be easy to identify these sources early, and keep them on a separate list, so that they do not interfere in network anomaly detection.

The rest of the chapter is organized as follows. Section 4.2 defines the superspreader problem and discusses previous approaches. Section 4.3 presents and compares two novel algorithms for the superspreader problem. Section 4.4 presents our extensions to handle distributed monitoring, deletions, and sliding windows. Section 4.5 presents our experimental results, and Section 4.6 presents conclusions.

4.2 Problem Definition and Previous Approaches

In this section, we present a formal definition of the problem and then discuss the deficiencies of previous techniques in addressing the problem.

4.2.1 Problem Definition

We define a k-*superspreader* as a host which contacts more than k unique destinations within a given window of N source-destination pairs. In Figure 4.1, for example, with $k = 2$, source $s2$ is the only k-superspreader. Note that there may be as many as N/k k-superspreaders in a given set of N packets, and reporting them would need $\Omega(N/k)$ space. Thus, this gives us a lower bound on the space bounds needed to find superspreaders. It also follows from a lower bound in [7] that any deterministic algorithm that accurately estimates (e.g., within 10%) the number of unique destinations for a source needs $\Omega(k)$ space. Because we are interested in small space algorithms, we must consider instead randomized algorithms.

More formally, given a user-specified $b > 1$ and confidence level $0 < \delta < 1$, we seek to report source IPs such that a source IP which contacts more than k unique destination IPs is reported with probability at least $1 - \delta$ while a source IP with less than k/b distinct destinations is (falsely) reported with probability at most δ. For example, when $k = 500$, $b = 2$ and $\delta = 0.05$, we want to report any source that contacts at least 500 distinct destinations and report no source that contacts less than 250 distinct destinations with probability 0.95.

We envision our algorithms to be useful in applications where it is acceptable to report sources whose distinct destination count is within a factor of 2 (or a factor of 5, 10, etc.) of a superspreader. For example, if we wish to identify sources involved in fast worm propagation and choose $k = 500$, it suffices to set $b = 2$, as we do not expect to find many sources (in normal traffic) that contact over 250 destinations within a short period. When a much finer distinction needs to be made (when b approaches 1), we will require a very high sampling rate, and there will not be a substantial

N	Total no. of packets in a given time interval
k	A superspreader sends to more than k distinct destinations
b	A false positive is a source that contacts less than k/b distinct destinations but is reported as a superspreader
δ	Probability that a given source becomes a false negative or a false positive
W	Sliding window size
s	Source IP address
d	Destination IP address

Table 4.1: Summary of notation

reduction in memory usage or computational time.

Also, note that by our problem statement, a source will be identified as a superspreader with high probability when it has contacted between $\frac{k}{b}$ and k destinations. Thus, we will expect to report the (potential) k-superspreader *before* it has contacted k destinations, and so our approach will not delay the identification of the superspreader.

Table 4.1 summarizes the notation used in this chapter.

4.2.2 Related Work and Previous Approaches

There has been a volume of work done in the area of streaming algorithms (see the surveys in [11, 90]). However, none of this work addresses the problem of identifying superspreaders efficiently. Perhaps most closely related is the problem of counting the number of distinct values in a stream. It has been studied by a number of papers (*e.g.*, [7, 12, 13, 30, 40, 49, 56, 57, 47]). The seminal algorithm by Flajolet and Martin [49] and its variant due to Alon, Matias and Szegedy [7] estimate the number of distinct values in a stream up to a relative error of $\epsilon > 1$. Cohen [28], Gibbons and Tirthapura [56], and Bar-Yossef et al. [13] give distinct counting algorithms that work for arbitrary relative error. More recently, Bar-Yossef *et al.* [12] improve the space complexity of distinct values counting on a single stream, and Cormode *et al.* [30] show how to compute the number of distinct values in a single stream in the presence of additions *and deletions* of items in the stream. Gibbons and Tirthapura [57] give an (ϵ, δ)-approximation scheme for distinct values counting over a *sliding window* of the last N items, using $B = O(\frac{1}{\epsilon^2} \log(1/\delta) \log N \log R)$ memory bits. The algorithm extends to handle distributed streams, where B bits are used for each stream.

Previous Approaches: We now discuss existing approaches that may be applied to find superspreaders and their deficiencies.

- *Approach 1:* As a first approach, Snort [100] simply keeps track of each source and the set of distinct destinations it contacts within a specified time window. Thus, the memory required by Snort will be at least the total number of distinct source-destination pairs within the time window, which is impractical for high-speed networks.
- *Approach 2:* Instead of keeping a list of distinct destinations that a source contacts for each source, an improved approach may be to use a (randomized) distinct counting algorithm to keep an approximate count of distinct destinations a source contacts for each source [47]. Along these lines, Estan et al. [47] propose using bitmaps to identify port-scans. The triggered bitmap construction that they propose keeps a small bitmap for each distinct source, and once the source contacts more than 4 distinct destinations, expands the size of the bitmap. Such an approach requires $n \cdot S$ space where n is the total number of distinct sources (which can be $\Omega(N)$) and S is the amount of space required for the distinct counting algorithm to estimate its count. These approaches are particularly inefficient when the number of k-superspreaders is small and many sources contact far fewer than k destinations.
- *Approach 3:* The recent work by Weaver et al. [124] proposes an interesting data structure for finding scanning worms using Threshold Random Walk [65]. This data structure may be adapted to find superspreaders by tracking the number of distinct destinations contacted in the address cache. [1] However, it may not scale well to high-speed links, as it needs to keep some state for every flow for a period of time (and thus, the memory usage could be $\Omega(N)$). We present a concrete example for the parameters in [124]. The 1 MB connection cache keeps per-flow details, and after seeing 1 million flows (in one direction), fewer than 37% of new flows (in the same direction) are expected to map to an unused entry in the cache.[2] When new flows map into an existing entry in the connection cache, the counter for the source does not get updated. Thus, roughly 63% of superspreaders that appear after these million flows will not be identified (in expectation). With a time-out of 10 minutes, a rate of 1700 flows a second will saturate the 1 MB connection cache to this point. If we assume that these million flows come from distinct sources, and we need to find 1000-superspreaders with $b = 2$, and error probability $\delta = 0.05$, our two-level filtering algorithm needs only an expected 24KB of space. Thus, in this scenario, our algorithm is more accurate and requires much less space. However, their data structure is designed to find small scans quickly, and in this case performance of our algorithms will degrade.
- *Approach 4:* Another approach that has not been previously considered is using a heavy-hitter algorithm in conjunction with a distinct-counting algorithm. We use a modified version of a heavy-hitter algorithm to identify sources that send to many destinations. Specifically, whereas heavy-hitters count the number of destinations, we count (approximately) the number of distinct destinations. This is done using a distinct counting algorithm. In our experiments we compare with this approach, with LossyCounting [84] as the heavy-hitter

[1]We refer the reader to [124] for an understanding of the data structure. Here, we just describe how to use it for detecting superspreaders and what the issues with doing so are.

[2]Theorems on occupancy problems give these numbers. For details, see [89].

algorithm, and the first algorithm from [12] as the distinct-counting algorithm. The results show that our algorithms use much less memory than this approach; the details are in Section 4.5.

Other Related Work: A number of papers have proposed algorithms for related problems in network traffic analysis. Estan and Varghese [46] propose two algorithms to identify the large flows in network traffic, and give an accurate estimate of their sizes. Estan et al. [45] present an offline algorithm that computes the multidimensional traffic clusters reflecting network usage patterns. Duffield et al. [44] show that the number and average length of flows may be inferred even when some flows are not sampled, and compute the distribution of flow lengths. Golab et al. [58] present a deterministic single-pass algorithm to identify frequent items over sliding windows. Cormode and Muthukrishnan [31] present sketch-based algorithms to identify large changes in network traffic.

4.3 Algorithms for Finding Superspreaders

We now propose two efficient algorithms to find superspreaders. We first propose a one-level filtering algorithm, based on sampling from the set of *distinct* source-destination pairs. We then present a more complex algorithm based on a novel two-level filtering scheme, which will be more space-efficient than the one-level filtering algorithm for the distributions that (we expect) will be more common.

4.3.1 One-level Filtering Algorithm

The intuition for our one-level filtering algorithm for identifying k-superspreaders over a given interval of N source-destination pairs is as follows.

We observe that if we sample the *distinct* source-destination pairs in the packets such that each distinct pair is included in the sample with probability p, then any source with m distinct destinations is expected to occur pm times in the sample. If p were $\frac{1}{k}$, then any k-superspreader (with its $m \geq k$ distinct destinations) would be expected to occur at least once in the sample, whereas sources that are not k-superspreaders would be expected *not* to occur in the sample. In this way, we may hope to use the sample to identify k-superspreaders[3].

There are several difficulties with this approach. First, the resulting sample would be a mixture of k-superspreaders and other sources that got "lucky" to be included in the sample. If there are no k-superspreaders, for example, the sample will consist only of lucky sources. To overcome this, we set p to be a constant factor c_1 larger than $\frac{1}{k}$. Then, any k-superspreader is expected to occur

[3]Note that we are sampling from the set of distinct source-destination pairs, not the set of packets we see; we perform a computation on every element in the stream – the "sampling" is at a conceptual level. The lower bounds of sampling approaches on counting distinct values [25] thus do not apply to our approach.

$$c_1 = \begin{cases} \ln(1/\delta) \cdot \left(\frac{3b+2b\sqrt{6b}+2b^2}{(b-1)^2}\right) \\ \qquad\qquad \text{if } b \leq 3 \\ \ln(1/\delta)\max(b, 2/(1-\frac{e}{b})^2) \\ \qquad\qquad \text{if } 3 < b < 2e^2 \\ 8\ln(1/\delta) \text{ if } b \geq 2e^2 \end{cases} \tag{4.1}$$

$$r = \begin{cases} \frac{c_1}{b} + \sqrt{\frac{3c_1}{b}\ln(1/\delta)} & \text{if } b \leq 3 \\ \frac{ec_1}{b} & \text{if } 3 < b < 2e^2 \\ \frac{c_1}{2} & \text{if } b \geq 2e^2 \end{cases} \tag{4.2}$$

Figure 4.2: The parameters c_1 and r for the one-level filtering scheme.

at least c_1 times in the sample, whereas lucky sources may occur a few times in the sample but nowhere near c_1 times. To minimize the space used by the algorithm, we seek to make c_1 as small as possible while being sufficiently large to distinguish k-superspreaders from lucky sources. A second, related difficulty is that there may be "unlucky" k-superspreaders that fail to appear in the sample as many times as expected. To overcome this, we have a second parameter $r < c_1$ and report a source as a k-superspreader as long as it occurs at least r times in the sample. A careful choice of c_1 and r is required.

Finally, we need an approach for uniform sampling from the *distinct* source-destination pairs. To accomplish this, we use a random hash function that maps source-destination pairs to $[0, 1)$ and include in the sample all distinct pairs that hash to $[0, p)$. Thus each distinct pair has probability p of being included in the sample. Using a hash function ensures that the probability of being included in the sample is not influenced by how many times a particular pair occurs. On the other hand, if a pair is selected for the sample, then all its duplicate occurrences will also be selected. To fix this, our algorithm checks for these subsequent duplicates and discards them.

Algorithm Description: Let *srcIP* and *dstIP* be the source and destination IP addresses, respectively, in a packet. Let h_1 be a uniform random hash function that maps (srcIP, dstIP) pairs to $[0, 1)$, (that is, each input is equally likely to map to any value in $[0, 1)$ independently of other inputs). At a high level, the algorithm is as follows:

- Retain all distinct (srcIP, dstIP) pairs such that $h_1(\text{srcIP, dstIP}) < \frac{c_1}{k}$, where c_1 is given in Figure 4.2.
- Report all srcIPs with more than r retained, where r is given by the equations in Figure 4.2(b).

We can implement the algorithm above using two hash-tables (with $\frac{c_1 N}{k}$ buckets each): the first one to detect and discard duplicate pairs from the sample, and the second one to count the number of distinct destinations for each source in the sample.

In more detail, the above steps can be implemented as follows. Our implementation has the desirable property that each k-superspreader is reported as soon as it is detected. We use two hash tables: one to detect and discard duplicate pairs from the sample, and the other to count the number of distinct destinations for each source in the sample. This latter hash table uses a second uniform random hash function h_2 that maps srcIPs to $[0, 1)$.

- *Initially:* Let T_1 be a hash table with $c_1 N/k$ entries, where each entry contains an initially empty linked list of (srcIP, dstIP) pairs. Let T_2 be a hash table with $c_1 N/k$ entries, where each entry contains an initially empty linked list of (srcIP, count) pairs.

- *On arrival of a packet with srcIP s and dstIP d:* If $h_1(s, d) \geq c_1/k$ then ignore the packet. Otherwise:

  1. Check entry $\frac{c_1 N}{k} \cdot h_1(s, d)$ of T_1, and insert (s, d) into the list for this entry if it is not present. Otherwise, it is a duplicate pair and we ignore the packet.
  2. At this point we know that d is a new destination for s, i.e., this is the first time (s, d) has appeared in the interval. We use $\frac{c_1 N}{k} \cdot h_2(s)$ to look-up s in T_2. If s is not found, insert $(s, 1)$ into the list for this entry, as this is the first destination for s in the sample. On the other hand, if s is found, then we increment its count, i.e., we replace the pair (s, m) with $(s, m + 1)$. If the new count equals $r + 1$, we report s. In this way, each declared k-superspreader is reported exactly once.

Note that at the end of the interval, the counts in T_2 can be used to provide a good estimate on the number of distinct dstIPs for each reported srcIP (by scaling them up by the inverse of the sampling rate, i.e., by a factor of k/c_1).

Accuracy Analysis: Our analysis yields the following theorem for precision:

Theorem 4.1. *For any given $b > 1$, positive $\delta < 1$, and t such that $b < k < 1$, the above algorithm reports srcIPs such that any k-superspreader is reported with probability at least $1 - \delta$, while a srcIP with at most k/b distinct destinations is (falsely) reported with probability at most δ.*

We defer the detailed proof to the full thesis.

Overhead Analysis: The total space is an expected $O(c_1 N/k)$ memory words. The choice of c_1 depends on b. By equation 4.2, we have that $c_1 = O(\ln(1/\delta)(\frac{b}{b-1})^2) = O((1 + \frac{1}{(b-1)^2})\ln(1/\delta))$ for $b \leq 3$. For $3 < b < 2e^2$, b is a constant, so $c_1 = O(\ln(1/\delta))$. For larger b, $c_1 = O(\ln(1/\delta))$. Thus across the entire range for b, we have $c_1 = O((1 + \frac{1}{(b-1)^2})\ln(1/\delta))$. This implies that the total space is an expected $O\left(\frac{N}{k}\ln\frac{1}{\delta}(1+\frac{1}{(b-1)^2})\right)$ bits. For the typical case where δ is a constant and $b \geq 2$, the algorithm requires space for only $O(N/k)$ memory words.

As for the per-packet processing time, note that each hash table is expected to hold $O(c_1 N/k)$ entries throughout the course of the algorithm. Thus each hash table look-up takes constant expected time, and hence each packet is processed in constant expected time.

We now give some examples to illustrate the one-level filtering algorithm.

Example: In this example, we set $k = 1000$ and $b = 2$, which means we are interested in reporting all sources that contact 1000 or more destinations within a given time period, without reporting any source that contacts less than 500 destinations within that time. Let N be the total number of packets seen in this time period.[4] For this, we find numerically that $c_1/k = 0.052$, and $r = 39$ suffice, when $\delta = 0.05$. Note that this sampling rate implies that in expectation, 94.8% of the packets will simply require one computation (hashing to see if the source-destination pair falls below c_1/k), and 5.2% of the packets will be selected for more processing. To store the source-destination pairs with a hash-table of $0.052N$, each of these selected packets will require (in expectation) no more than a read and a write of two IP addresses, which is a small computational overhead. To count the number of distinct destinations for any source in the first hash-table, we could use another hash-table and have an additional overhead of (at most) 2 reads and 2 writes (an IP address and a counter) per stored packet.

Note that these quantities do *not* depend on the distribution of the number of distinct destinations by source. That is, even if nearly every source sent to exactly one destination, the basic algorithm would have us store 5.2% of these sources, where the number 0.052 depends on k, b, and N. We would like to reduce the memory used in storing these non-superspreader sources. Further, we expect that most traces will have a very large number of sources that contact only a few distinct destinations, and *very* few superspreaders. Can we track the superspreaders accurately without tracking so many non-superspreaders?

The difficulty here is that the one-level filtering algorithm needs a certain minimum sampling rate in order to distinguish between sources that send to k destinations and $\frac{k}{b}$ destinations. But sources that contact only a few destinations also get sampled *at this rate*. In the next section, we will effectively reduce the sampling rate of these non-superspreader sources without compromising on the accuracy of the algorithm for detecting superspreaders.

4.3.2 Two-Level Filtering Algorithm

We now present another algorithm that uses *two* levels of filters and is more memory-efficient than one-level filtering in most cases. At a high level, the algorithm uses two levels of filtering

[4]In a real setting, N could be determined historically.

```
function Two-Level Filtering$(s, d)$
    Level2$(s, d)$; Level1$(s, d)$;

function Level1$(s, d)$
    if($h_1$(s,d) $< r_1$) insert $s$ into $T_1$

function Level2$(s, d)$
    if $(h_2(s, d) > r_2)$ return;
    if $(s \notin T_1)$ return;
    else compute $p = \frac{h_2(s,d)}{r_2} \cdot \gamma$ and insert $s$ into $T_{2,p}$.

function Output
    output all sources that appear in more than $\omega$ of the hash-tables $T_{2,i}$.
```

Figure 4.3: Two-level filtering pseudocode, where (s, d) represents a source-destination pair.

in the following manner: the first-level filter effectively decides whether we should keep more information about a particular source, while the second-level filter effectively keeps a small digest that can then be used to identify superspreaders. The first level has a lower sampling rate than the second level. Thus intuitively, the first level is a coarse filter that filters out sources that contact only a small number of distinct destinations, so that we do not need to allocate any memory space for them. The second level is a more precise filter which uses more memory space, and we only use it for sources that pass the first filter.

Intuitively, the reason why the two-level filtering algorithm is more space-efficient than the one-level filtering algorithm is because the sampling rate for the *first level* of two-level filtering algorithm is lower than the sampling rate of the one-level filtering algorithm. (To compensate, the sampling rate for the *second level* will need to be a bit higher.) If a source contacts sufficiently many destinations, it will be sampled (and thus, stored) in both the one-level filtering algorithm and the two-level filtering algorithm. But if a source contacts only a few destinations, the probability that it is sampled (and tracked) in the two-level filtering algorithm is much lower than the probability that it is sampled (and tracked) in the one-level filtering algorithm. Thus, the two-level filtering algorithm will store fewer sources that contact very few distinct destinations. It is therefore more space-efficient when there are many sources that contact only a few distinct destinations.

This type of sampling at multiple levels is a new approach that may be of independent interest.

Algorithm Description: The algorithm takes r_1, r_2, γ, ω as parameters, where r_1 and r_2 represent the sampling rate in the first and second level respectively, and ω is a threshold. Given the required values for k and b, the values of r_1, r_2, γ, ω may be determined as in the analysis of Theorem 4.2.

We keep one hash-table T_1 at the first level, and γ hash-tables denoted $T_{2,i}$ at the second level.

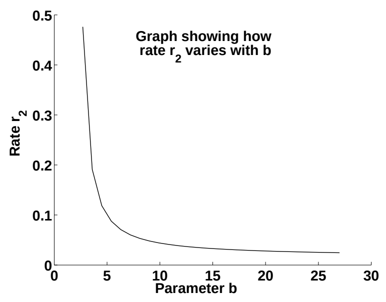


Figure 4.4: The rate r_2 required for $k = 1000$, $\delta = 0.01$, with varying b.

Let h_1 and h_2 be uniform random hash functions that take a source-destination pair and return a value in $[0, 1)$ as described in the previous section.

For each packet (s, d), the network monitor performs the following operations as given in pseudocode in Figure 4.3:

- Step 1: First, we compute $h_2(s, d)$. If $h_2(s, d)$ is greater than rate r_2, we skip to step 2. Otherwise, we check to see if the source s is present in the hash-table T_1. If s is not present in T_1, then again, we skip to step 2. Otherwise, we insert s into level 2 as follows: If $h_2(s, d) < r_2$ and s is present in T_1, we insert s into the level-2 hash-table $T_{2,p}$, where $p = \frac{h_2(s,d)}{r_2} \cdot \gamma$. Thus, we insert s into level 2 with at most probability r_2, and every source appearing in level 2 appears in level 1.
- Step 2: If $h_1(s, d)$ is less than rate r_1, we insert s into T_1.

Finally, we output all sources that appear in more than ω of the tables $T_{2,i}$.

Optimizations: Note that in the above description, we use hash-tables to store the sampled elements for ease of explanation. We can easily optimize the storage space in the two-level sampling further by using Bloom filters instead of hash-tables to store the sampled elements. A discussion of Bloom filters may be found in [16, 20]. In addition, in the above description, we chose the probability of inserting a sampled packet into any level-2 hash-table $T_{2,i}$ to be equal to $1/\gamma$, for simpler description and analysis. We can easily generalize this to alter the probability of inserting a sampled packet into any level-2 hash-table to be non-uniform, e.g., an exponential distribution.

Accuracy Analysis Our analysis yields the following theorem for precision:

Theorem 4.2. *Given k, $N, b > 1$ such that $k/b > 1$ and $0 < \delta < 1$. Let $z = \max(\frac{2b}{b-1}, 5)$. Let $r_1 = \frac{z}{k} \log \frac{2}{\delta}$, and r_2 be minimal value that satisfies the following constraints:*

(1) $r_2 \geq \frac{2 \ln 2/\delta}{k(1-e^{-(z-1)/z})\epsilon_1^2}$*, and (2)* $r_2 \geq \frac{\ln 1/\delta}{k(1-e^{-3/2b})((1+\epsilon_2)\ln(1+\epsilon_2)-\epsilon_2)}$*, where* $\epsilon_1 = 1 - \frac{1-e^{-3/2b}}{1-1/e}(1+\epsilon_2), \epsilon_2 > 0,$ *and* $0 < \epsilon_1 < 1$.

Let ϵ_1' *be the value of* ϵ_1 *when* r_2 *is minimized in the above constraints. Let* $\gamma = r_2 k$*, and* $\omega = (1-\epsilon_1')\gamma(1-\frac{1}{e})$.

Then, for any given $b > 1$*, positive* $\delta < 1$*, and* k *such that* $k/b > 1$*, Algorithm II reports srcIPs such that any* k*-superspreader is reported with probability at least* $1-\delta$*, while a srcIP with at most* k/b *distinct destinations is (falsely) reported with probability at most* δ*.*

We defer the complete proof to the thesis.

Figure 4.4(b) shows how the required rate r_2 varies with b. The threshold ω and the number of hash-tables γ vary similarly.

The expected space required is $O(r_1 N + r_2 N)$. Note that, for a fixed b, both r_1 and r_2 are $O(\frac{1}{k}\ln\frac{1}{\delta})$, and thus the space required is $O(\frac{N}{k}\ln\frac{1}{\delta})$.

We may make a similar statement when we use Bloom filters rather than hash-tables to store sampled elements as described in the optimization above. Using Bloom filters does not affect the false negative rate, but only the false positive rate. We can easily reduce the additional false positive rate caused by the Bloom filter collision by setting the correct parameters of the Bloom filters using the theorems in [16].

We observe also that the accuracy of both algorithms is independent of the input distribution of source-destinations pairs, as long as the assumption of uniform random hash function is obeyed. In addition, note that it is important to pick secret hash functions at run-time each time so that the attacker cannot generate an input sequence that avoid certain hash values. Also, in practice, we optimize our choice of the parameters numerically for both algorithms, since the bounds given by the theorems may have larger constant factors than are strictly necessary.

Example: In this example, we set $k = 1000$ and $b = 2$, which means we are interested in reporting all sources that contact 1000 or more destinations, without reporting any source that contacts less than 500 destinations. For this, we find numerically that $r_1 = 0.006$, $r_2 = 0.15$ and $\gamma = 100$ suffice, when $\delta = 0.05$. Note that this sampling rate implies that 85% of the flows will need only to be hashed once and incur no memory accesses, and 15% of the flows will have to be additionally processed. The amount of computational overhead that these selected flows incur will depend on the number of distinct destinations that their respective source contacts, so we will examine two specific cases:
Case 1: For the sources that contact exactly one distinct destination each, in expectation, 0.6% of the packets will be entered into the first level, and require 1 read and 1 write (of one IP address), and 15% of the packets will require exactly 1 read.
Case 2: For any particular superspreader, at most 15% of the distinct flows (corresponding to that superspreader) will require 2 distinct memory locations (1 in level-1, 1 in level-2, of 1 IP address

each, with 2 reads and 1 write).
Note that if the trace contains only sources that contact one destination each (the first case), the two-level filtering algorithm has much less overhead than the one-level filtering algorithm, and if the trace contains only superspreaders (the second case), two-level filtering algorithm has about three times as much overhead as one-level filtering algorithm. This gives an idea of the trade-off between the algorithms; we expect that most sources only contact a few distinct destinations, and thus the traces will resemble the first case far more than the second case.

Using a Bloom filter in both levels will reduce the memory storage required, but increase the number of accesses that need to be made. If we use a Bloom filter with 8 independent hash functions at each of the two levels, our memory storage will drop by a constant factor of at least 2.5 (estimating conservatively to account for additional false positives), and our computational overhead will increase by a factor of 8 – since we will need to make 8 memory accesses for every memory access of the hash-table implementation.

Note that there could exist as many as $\frac{N}{k}$ superspreaders; thus, for constant b, all our bounds are within a $\log \frac{1}{\delta}$ factor of the asymptotically optimal values.

4.4 Extensions

In this section we show how to extend our algorithms to handle deletions, and sliding windows and distributed monitoring.

4.4.1 Superspreaders with Deletion

We can extend our algorithms to support streams that include both newly arriving (srcIP, dstIP) pairs and the *deletion* of earlier (srcIP, dstIP) pairs. Recall from Section 4.1 that a motivating scenario for supporting such deletions is finding source IP addresses that contact a high number of distinct destinations and do not get legitimate replies from a high number of these destinations. Each in-bound legitimate reply packet with source IP x and destination IP y is viewed as a deletion of an earlier request packet with source IP y and destination IP x from the corresponding flow, so that the source y is charged for only distinct destinations without legitimate replies.

For the one-level filtering algorithm (Section 4.3.1), a deletion of (s, d) is handled by first checking to see if (s, d) is in the hash-table. If it is not, then d is already not being accounted for in s's distinct destination count, so we can ignore the deletion. Otherwise, we delete (s, d) from the hash-table. The precision, space, and time bounds are the same as in the case without deletions. Similarly, we can extend the hash-table implementation of the two-level filtering algorithm to handle deletions as well.[5]

[5]Technically, we need a slight modification of the algorithm described earlier; we need to store the destinations as well at each level-2 hash-table; this may increase the memory required by at most a factor of 2.

We can also use this approach to find those sources which have more than k failures and fewer than k/b successes. We could find these sources by computing separately the sources that have at least k successes and those that have at least k failures, and return the appropriate difference.

Note that the definition of a k-superspreader under deletions is not a stable one. At any point in time, the monitor may have just processed a packet, and have no idea whether this pair will be subsequently deleted. There may be a source right at the k-superspreader threshold that exceeds the threshold unless the pair is subsequently deleted. Our algorithms can be readily adapted to handle a variety of ways of treating this issue. For example, the one-level filtering algorithm can report a source as a *tentative* k-superspreader when its count in T_2 reaches $r + 1$, and then report at the end of the interval which sources (still) have counts greater than r.

4.4.2 Superspreaders over Sliding Windows

In this section, we show how to extend our algorithms to handle sliding windows. Our goal is to identify k-superspreaders with respect to the most recent W packets, i.e., hosts which contact more than k unique destinations in the last W packets. Our goal is to use far less space than the space needed to hold all the pairs in the current window.

Figure 4.5 gives an example of a stream subject to a sliding window of size $W = 4$. The top row shows the packets in the sliding window after the arrival of $(s1, d3)$. The middle row shows that on the arrival of $(s2, d3)$, the window includes this pair but drops $(s1, d1)$. The bottom row shows that on the arrival of $(s2, d1)$, the window adds this pair but drops $(s1, d2)$.

What makes the sliding window setting more difficult than the standard setting is that a packet is dropped from the window at each step, but we do not have the space to hold on to the packet until it is time to drop it. This is in contrast to the deletions setting described in Section 4.4.1 where we are given at the time of deletion the source-destination pair to delete.

In the sliding window setting, a source may transition between being a k-superspreader and not, as the window slides. In Figure 4.5, for example, suppose that the threshold for being a k-superspreader is having at least 3 distinct destinations (e.g., $k = 3$). Then source $s1$ is a superspreader in the first window, but not the second or third windows.

We show how to adapt one-level filtering algorithm to handle sliding windows. the approach for two-level filtering algorithm is similar. We keep a running counter of packets that is used to associate each packet with its stream sequence number (seqNum). Thus if the counter is currently x, the sliding window contains packets with sequence numbers $x - W + 1, \ldots, x$. At a high level, the algorithm works by (1) maintaining the pairs in our sample sorted by sequence number, in order to find in $O(1)$ time sample points that drop out of the sliding window, and (2) keeping track of the largest sequence number for each pair in our sample, in order to determine in $O(1)$ time whether there is at least one occurrence of the pair still in the window.

In further detail, the steps of the algorithm are as follows.

$(s1, d1)$, $(s1, d2)$, $(s2, d2)$, $(s1, d3)$
$\quad\quad\quad$ $(s1, d2)$, $(s2, d2)$, $(s1, d3)$, $(s2, d3)$
$\quad\quad\quad\quad\quad\quad$ $(s2, d2)$, $(s1, d3)$, $(s2, d3)$, $(s2, d1)$

Figure 4.5: Example stream, showing three steps of a sliding window of size $W = 4$. The top row shows the packets in the sliding window after the arrival of $(s1, d3)$. The middle row shows that on the arrival of $(s2, d3)$, the window includes this pair but drops $(s1, d1)$. The bottom row shows that on the arrival of $(s2, d1)$, the window adds this pair but drops $(s1, d2)$.

- *Initially:* Let L be an initially empty linked list of (srcIP, dstIP, seqNum) triples, sorted by increasing seqNum. Let T_1 and T_2 be as in the original one-level filtering algorithm, except that T_1 now contains (srcIP, dstIP, seqNum) triples.
- *On arrival of a packet with srcIP s and dstIP d:* Let x be its assigned sequence number.
  1. *Account for a pair dropping out of the window, if any:* If the tail of L is a triple (s, d, n) such that $n = x - W$, then remove the triple from L and check to see if the triple exists in entry $\frac{c_1 N}{k} \cdot h_1(s, d)$ of T_1. If the triple exists, then because T_1 holds the latest sequence numbers for each source-destination pair in the sample, we know that (s, d) will not exist in the window after dropping (s, d, n). Accordingly, we perform the following steps:
     (a) Remove the triple from T_1.
     (b) Use $\frac{c_1 N}{k} \cdot h_2(s)$ to look-up s in T_2, and decrement the count of this entry in T_2, i.e., replace the pair (s, m) with $(s, m - 1)$.
     (c) If the new count equals 0, we know that the source no longer appears in the sample and we remove the pair from T_2.

     On the other hand, if the triple does not exist, then there is some other triple (s, d, n') corresponding to a more recent occurrence of (s, d) in the stream ($n < n'$). Thus dropping (s, d, n) changes neither the sampled pairs nor the source counts, so we simply proceed to the next step.
  2. *Account for the new pair being included in the window:* If $h_1(s, d) \geq c_1/k$ ignore the packet. Else:
     (a) Check entry $\frac{c_1 N}{k} \cdot h_1(s, d)$ of T_1 for a triple with s and d. If such an entry exists, replace it with (s, d, x), maintaining the invariant that the entry has the latest sequence number, and return to process the next packet. Otherwise, insert (s, d, x) into the list for this entry.
     (b) At this point we know that d is a new destination for s, i.e., this is the first time (s, d) has appeared in the window. We use $\frac{c_1 N}{k} \cdot h_2(s)$ to look-up s in T_2. If s is not found, insert $(s, 1)$ into the list for this entry, as this is the first destination for

Stream 1:	$(s1, d1), (s2, d2), (s3, d3), (s4, d4)$
Stream 2:	$(s2, d3), (s1, d1), (s1, d1), (s3, d2)$
Stream 3:	$(s4, d2), (s4, d4), (s2, d4), (s4, d3)$

Figure 4.6: Example streams at 3 monitoring points

s in the sample. On the other hand, if s is found, then we increment its count, i.e., we replace the pair (s, m) with $(s, m+1)$. If the new count equals $r+1$, we report s.

The precision, time and space bounds are the same as in one-level filtering algorithm of Section 4.3.1 with W substituted for N.

Note that the algorithm is readily modified to handle sliding windows based on time, e.g., over the last 60 minutes, by using timestamps instead of sequence numbers. The precision, time and space bounds are unchanged, except that the time is now an amortized time bound instead of an expected one. This is because multiple pairs can drop out of the window during the time between consecutive arrivals of new pairs. If more than a constant number of pairs drop out, then the algorithm requires more than a constant amount of time to process them. However, each arriving pair can drop out only once, so the amortized per-arrival cost is constant.

4.4.3 Distributed Superspreaders

In the distributed setting, we would like to identify source IP addresses that contact a large number of unique hosts in the union of the streams monitored by a set of distributed monitoring points. Consider for example, the three streams in Figure 4.6 and $k = 3$. Sources $s1$, $s2$, $s3$, and $s4$ contact 1, 3, 2, and 3 distinct destinations, respectively. Thus for the total of $N = 12$ source-destination pairs, only $s2$ and $s4$ are k-superspreaders.

Note that a source IP address may contact a large number of hosts overall, but only a small number of hosts in any one stream. Source $s2$ in Figure 4.6 is an example of this. A key challenge is to enable this distributed identification while having only limited communication between the monitoring points.

We describe how to modify our one-level filtering algorithm to work in a distributed setting. First, each network monitor runs the algorithm as described in Section 4.3.1 (all using the same hash function, and with appropriately sized hash-tables, say c_1N/kj if each of the j monitors expects to see N/j packets). The monitor reports any locally detected superspreader. Next, at the end of the stream, each monitor sends its hash-table of (s, d) pairs to the central monitor. Finally, the central monitor treats these hash-tables as a stream of (s, d) pairs, and using the same hash function, runs the algorithm on this stream, and reports any superspreader found.

The overall space and time overhead of first step above summed over all the monitors is the same as if one monitor monitors the union of the streams. The second step requires a total amount of communication equal to the sum of the space for the hash-tables, i.e., an expected $O(c_1 N/k)$ memory words. Accounting for the last step increases the total space and time by at most a factor of 2. Note that the algorithm does not require that all streams use an interval of N/j packets. As long as there are exactly N packets in all, the algorithm achieves the precision bounds given in Theorem 4.1. Thus our distributed one-level filtering algorithm uses little memory and little communication. On the other hand, a similar extension to the two-level filtering algorithm results in more communication in the distributed setting – specifically, at each step, the monitors would need to have access to all the (individual) first-level hash-tables, which results in significant increase in communication between monitors.

4.5 Experimental Results

We implemented our algorithms for finding superspreaders, and we evaluated them on network traces taken from the NLANR archive [94], after they were injected with appropriate superspreaders as needed. All of our experiments were run on an Intel Pentium IV, 1.8GHz. We use the *OPENSSL* implementation of the *SHA1* hash function, picking a random key during each run, so that the attacker cannot predict the hashing values. For a real implementation, one can use a more efficient hash function. Both algorithms are implemented so that the superspreaders get output at the end of the run, once all the packets have been processed. We ran our experiments on several traces and obtained similar results. Our results show that our algorithms are fast, have high precision, and use a small amount of memory. On average, the algorithms take on the order of a few seconds for a hundred thousand to a million packets (with a non-optimized implementation).

In this section, we first examine the precision of the algorithms experimentally, then examine the memory used as k, b and N change, and finally compare with the alternate approach proposed in Section 4.2.2.

4.5.1 Experimental evaluation of precision

To illustrate the precision of the algorithms, we show a set of experimental results below. To the base trace 1 (see Figure 4.8), we inserted various attack packets where some sources contacted a high number of distinct destinations. That is, for given parameters k and b, we added 100 sources that send to k destinations each, and 100 sources that send to just under k/b destinations each. This was done in order to test if our algorithms do indeed distinguish between sources that send to more than k destinations and fewer than k/b destinations.

We set $\delta = 0.05$. In Figure 4.7, we show the results of our experiments, with regards to precision of the algorithms. We examine the correctness of our algorithm by comparing it against an exact calculation of the number of distinct destinations each source contacts. We optimize our choice of the other parameters numerically for both algorithms (in a manner suggested by the

k	b	False Positives		False Negatives	
		1-Level	2-Level	1-Level	2-Level
500	2	8.1e-5	6.3e-5	0	0
500	5	1.13e-4	1.13e-4	0	0
500	10	1.35e-4	8.1e-5	0.01	0
1000	2	4.95e-5	1.13e-4	0.02	0
1000	5	1.62e-4	0	0.02	0.02
1000	10	1.13e-4	9.45e-5	0	0.03
5000	2	8.1e-5	0	0	0
5000	5	4.95e-5	1.62e-5	0	0
5000	10	3.19e-5	1.62e-5	0	0.01
10000	2	1.62e-4	0	0.02	0
10000	5	3.2e-5	0	0.01	0
10000	10	1.62e-5	3.2e-5	0.04	0

Figure 4.7: Evaluation of the precision of one-level filtering and two-level filtering algorithms over various settings for k and b, with $\delta = 0.05$.

analysis of the theorems), since the bounds given by the theorems may have larger constant factors than are strictly necessary.

We observe that the accuracy of both algorithms is comparable and bounded by δ, which confirms our theoretical results. Note that using a smaller value of δ would produce a smaller false positive rate and false negative rate. We note that the false positive rate is much lower than the false negative rate. Our sampling rates are chosen to distinguish sources that send to k destinations from sources that send to k/b destinations with error rate δ. When a source sends to a very small number of destinations (much smaller than k/b), the probability that it becomes a false positive is significantly lower than δ. Likewise, when a source sends to a very large number of destinations ($\gg k$), the probability that it becomes a false negative is much less than δ. Through the construction of our traces, there are only a 100 possible sources that may be false negatives, and all of them send to just over k destinations. There are many more sources that could be false positives, and only a 100 of these sources send to nearly k/b destinations. Thus, the false positive rate that is seen is much less than the set δ. Further, *all of the false positives in our experiments come from the sources at the threshold that we added*, not the original trace itself. The false positive rate is typically of much more importance than the false negative rate, since there are usually many more sources that could be false positives than sources that could be false negatives. Thus, it is very useful to verify that the false positive rate is much lower than the stated δ in real traces, and that the false positives observed do come only from the inserted traffic at the threshold.

	Length (in sec)	No. distinct sources	No. distinct src-dst pairs	N (no. of packets)
1	65	59,862	194,060	$2.88e6$
2	154	282,484	416,730	$3.09e6$
3	207	$1.21e6$	$1.35e6$	$4.02e6$
4	269	$2.12e6$	$2.29e6$	$4.49e6$

Figure 4.8: Base traces used for experiments

4.5.2 Memory usage on long traces

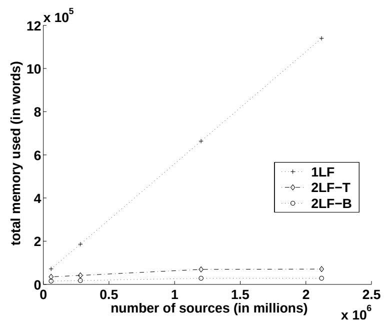
We now examine memory used on very long traces by one-level filtering algorithm (Section 4.3.1) and the hash-table and Bloom-filter implementations of the two-level filtering algorithm (Section 4.3.2). To distinguish the two implementations of the two-level filtering algorithm, we will refer to the hash-table implementation as 2LF-T, and the Bloom-filter implementation as 2LF-B. We will use 1LF to refer to the one-level filtering algorithm. We examine the memory used as the parameters k, b and N are allowed to vary. The memory usage reported is the number of elements actually stored, which is always very close to the size of the hash-tables. (The size of each hash-table set to be the expected number of elements that will be inserted, based on the sampling rates and N.) For the bloom filter implementation, we use 8 independent hash functions.

The traces used for this section are constructed by taking four base traces of varying lengths, and adding to each of them a hundred sources that send to k destinations, and a hundred sources that send to k/b destinations. The details of the base traces are shown in Figure 4.8. We observe that, with the largest of these traces, a source that sends to 200 distinct destinations contributes just about 0.004% to the total traffic analyzed. The memory used is the number of words (or IP addresses) that need to be stored.

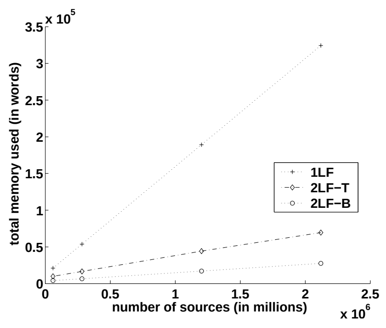
The graphs in Figure 4.9 show the total memory used by each algorithm plotted against the number of distinct sources in the trace, at different values of b. Notice that through our trace construction procedure, the traces in Figure 4.9(a), 4.9(b), and 4.9(c) contain the same number of distinct sources, even though the value of b differs.

We observe that the memory used by the two algorithms is strongly correlated with b, as pointed out by our theoretical analysis. For both algorithms, the memory required decreases sharply as b increases from 2 to 5, and then decreases more slowly. This can also be seen (for 2LF) from Figure 4.4, in section 4.3.2.

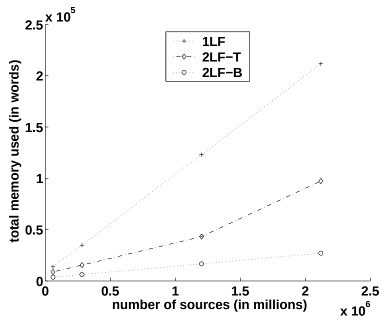
Another observation is that, as expected, the memory used by 1LF eventually exceeds the memory used by 2LF-T & 2LF-B, for every value of b. The number of sources at which the memory used by 1LF exceeds the memory used by 2LF-T & 2LF-B also depends on b. We also note that, as expected, the memory used by 2LF-B is much less than the memory used by 2LF-T and 1LF .



(a) $b = 2$

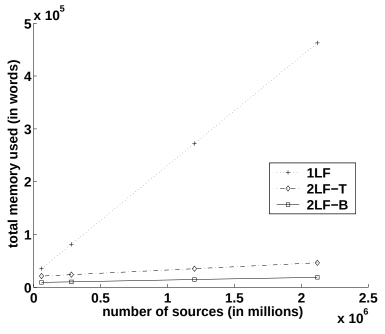


(b) $b = 5$

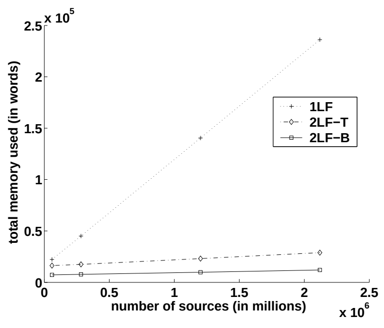


(c) $b = 10$

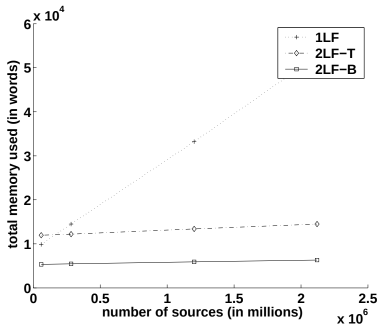
Figure 4.9: Total memory used by the algorithms in words (i.e., IP addresses) vs number of distinct sources, for $b = 2$, 5 and 10, at $k = 200$.



(a) $k = 500$



(b) $k = 1000$

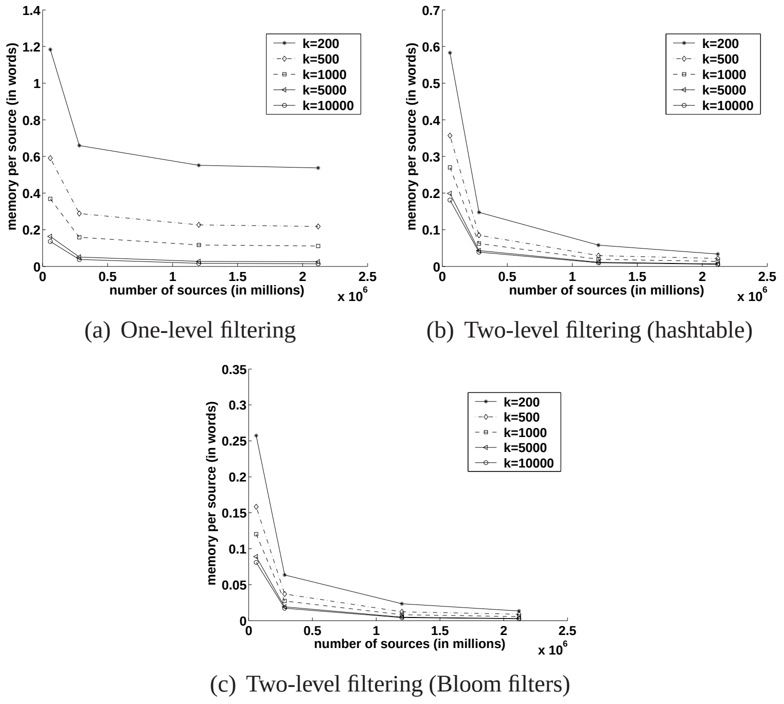


(c) $k = 5000$

Figure 4.10: Total memory used by the algorithms in words (i.e., IP addresses) vs number of distinct sources, for $k = 500$, 1000 and 5000, at $b = 2$.

We next examine the memory usage as k changes, which is shown in Figures 4.10 and 4.11. We observe that the total memory used drops sharply as k increases, as expected: in 4.10(a), at $k = 500$, the memory used ranges from $20,000$ to $200,000$ IP addresses; in 4.10(c), at $k = 5000$, it ranges from $10,000$ to $55,000$ IP addresses. Even though the number of source-destination pairs increases when k increases, we can afford to sample much less frequently. This in turn decreases the number of sources stored that have very few destinations, and thus the total memory used decreases.

Also, for every k, as the number of packets N increases, the memory used by 1LF eventually exceeds the memory used by 2LF-T & 2LF-B. This is because of the two-level sampling scheme. Since the first sampling rate r_1 is much smaller than c_1/k in 1LF , the number of non-superspreader sources stored in 2LF-T & 2LF-B (r_1N in expectation) is much less than in 1LF . The actual number of sources at which this occurs depends on k. As k increases, the number of sources at which the memory used by 1LF exceeds the memory used by 2LF-T (and 2LF-B) also increases, since the sampling rates for both algorithms decrease in the same way. We also observe that, once again, the memory used by 2LF-B is significantly lower than 1LF and 2LF-T.



(a) One-level filtering

(b) Two-level filtering (hashtable)

(c) Two-level filtering (Bloom filters)

Figure 4.11: Memory used per source vs number of distinct sources, for all k, by 1LF , 2LF-T, & 2LF-B at $b = 2$.

The graphs in Figure 4.11 show the memory used per source plotted against the number of distinct sources, for various k – as k increases, the total memory used drops. We observe that

b	1LF	2LF-T	2LF-B	Alt I	Alt II
Trace 1					
2	37610	16234	7223	49063	105589
5	9563	3241	1377	20746	48424
10	5685	2698	1136	16839	36823
Trace 2					
2	71852	17536	7711	133988	273101
5	19298	4543	1865	76543	168256
10	12030	4000	1624	67007	135540

Figure 4.12: Comparisons of total memory used with traces 1 & 2 for $k = 1000$ and varying b.

each algorithm has a similar dependence on k, though the absolute memory usage is different, as discussed.

Lastly, we tested both 1LF and 2LF-T on a trace with 10 million sources that contacted a few destinations each. At $k = 1000$ and $b = 2$, the memory usage of 1LF and 2LF-T were about 1.04 million and 60,100 IP addresses respectively. Thus, we see that our algorithms do indeed scale well as the number of flows increases.

4.5.3 Comparison with an Alternate Approach

We now show results comparing our approach to the Approach 4 described in Section 4.2.2: we count the number of distinct destinations that a source sends to using LossyCounting [84], replacing the regular counter with a distinct-element counter. We do not show experimental comparisons with the other approaches as they all keep per-flow state, and so, it is clear that they need far more space than our algorithms.

We chose the parameters for LossyCounting and the distinct counting algorithm so that (a) the memory usage was minimized for each b and (b) the false positive and false negative rates were similar to (but at least as large as) the results of our algorithm over 10 iterations; we ensured this by slowly reducing the memory used by the alternate algorithms until the error rates were just slightly larger than our algorithm, in order to make a comparison that was favourable to the alternate algorithms. We show experimental results with two variants of the approach: (1) use one distinct counter per source (this is Alt I), and (2) use $\log \frac{1}{\delta}$ distinct counters per source, and use their median for the estimate of the number of distinct elements is used (this is Alt II). The memory used is reported as the maximum of the total number of hash values stored for all the sources at any particular time.

Figure 4.12 shows the result of the comparison of memory usage at $k = 1000$, for $b = 2, 5$ and 10, on Trace 1 & Trace 2. Note that all our algorithms show better performance than Alt I and Alt II on Traces 1 & 2. The results for Trace 3 & Trace 4 are similar, except that Alt I uses less memory than 1LF when $b = 2$.

4.6 Conclusion

In this chapter, we have described new streaming algorithms for identifying superspreaders on high-speed networks. Our algorithms give proven guarantees on the accuracy and the memory requirements. Compared to previous approaches, our algorithms are substantially more efficient, both theoretically and experimentally. We also provide several extensions to our algorithms – we can identify superspreaders in a distributed setting, over the sliding windows, and when deletions are allowed in the stream (which lets us identify sources that make a large number of failed connections to distinct destinations). Our algorithms have many important networking and security applications. We also hope that our algorithms will shed new light on developing new fast streaming algorithms for high-speed network monitoring.

The results presented in this chapter are joint work with Dawn Song and Phillip Gibbons and Avrim Blum, and have previously appeared at the 12^th Annual Network and Distributed Systems Security Symposium, 2005 [116].

Chapter 5

Exploiting IP-based Network Structure for Spam Mitigation

5.1 Introduction

E-mail has emerged as an indispensable and ubiquitous means of communication today. Unfortunately, the ever-growing volume of spam diminishes the efficiency of e-mail, and requires both mail server and human resources to handle.

Great effort has focused on reducing the amount of spam that the end-users receive. Most Internet Service Providers (ISPs) operate various types of spam filters [1, 4, 5, 62] to identify and remove spam e-mails before they are received by the end-user. E-mail software on end-hosts adds an additional layer of filtering to remove this unwanted traffic, based on the typical email patterns of the end-user.

Much less attention has been paid to how the large volume of spam impacts the mail infrastructure within an ISP, which has to receive, filter and deliver them appropriately. Spammers have a strong incentive to send large volumes of spam – the more spam they send, the more likely it is that some of it can evade the spam filters deployed by the ISPs. It is easy for the spammer to achieve this – by sending spam using large botnets, spammers can easily generate far more messages than even the largest mail servers can receive. In such conditions, it is critical to understand how the mail server infrastructure can be made to prioritize legitimate mail, processing it preferentially over spam.

In this context, the requirements for differentiating between spam and non-spam are slightly different from regular spam-filtering. The primary requirement for regular spam-filtering is to be conservative in discarding spam, and for this, computational cost is not usually a consideration. However, when the mail server must prioritize the processing of legitimate mail, it has to use a computationally-efficient technique to do so. In addition, in this situation, even an imperfect distinction criterion would be useful, as long as a significant fraction of the legitimate mail gets classified correctly.

In our work, we explore the potential of using the historical behaviour of IP addresses to predict whether an incoming email is likely to be legitimate or spam. Using IP addresses for classification is computationally substantially more efficient than any content-based techniques. IP address information can also be collected easily and is more difficult for a spammer to obfuscate. Our measurement studies show that IP address information provides a stable discriminator between legitimate mail and spam. We find that good mail servers send mostly legitimate mail and are persistent for significant periods of time. We also find that the bulk of spam comes from IP prefixes that send mostly spam and are also persistent. With these two findings, we can use the properties of *both* legitimate mail and spam together, rather than using the properties of only legitimate mail or only spam, in order to prioritize legitimate mail when needed.

We show that these measurements are valuable in an application where legitimate mail must be prioritized. We focus on the situation when mail servers are overloaded, i.e., they receive far more mail than they can process, even though the legitimate mail received is a tiny fraction of the total received. Since mail typically gets dropped at random when the server is overloaded, and spam can be generated at will, the spammer has an incentive to overload the server. Indeed, the optimal strategy for the spammer is to increase the load on the mail infrastructure to a point where the most spam will be accepted by the server; this kind of behaviour has been observed on the mail servers of large ISPs. In this work, we show an application of our measurement study to design techniques based on the reputations of IP addresses and their aggregates and demonstrate the benefits to the mail server overload problem.

Our contributions are two-fold. We first perform an extensive measurement study in order to understand some IP-based properties of legitimate mail and spam. We then perform a simulation study to evaluate how we can use these properties to prioritize legitimate mail when the mail server is overloaded.

Our main results are the following:

- We find that a significant fraction of legitimate mail comes from IP addresses that last for a long time, even though a very significant fraction of spam comes from IP addresses that are ephemeral. This suggests that the history of “good” IP addresses, that is, IP addresses that send mostly legitimate mail, could be used for prioritizing mail in spam mitigation.

- We explore *network-aware clusters* as a candidate aggregation scheme to exploit structure in IP addresses. Our results suggest that IP addresses responsible for the bulk of the spam are well-clustered, and that the clusters responsible for the bulk of the spam are persistent. This suggests that network-aware clusters may be good candidates to assign reputations to unknown IP addresses.

- Based on our measurement results, we develop a simple reputation scheme that can prioritize

IP addresses when the server is overloaded. Our simulations show that when the server receives many more connection requests than it can process, our policy gives a factor of 3 improvement in the number of legitimate mails accepted over the state-of-the-art.

We note that the server overload problem is just one application that illustrates how IP information could be used for prioritizing email. This information could be used to prioritize e-mail at additional points of the mail server infrastructure as well. However, the kind of structural information that is reflected in the IP addresses may not always be a perfect discriminator between spammers and senders of legitimate mail, and this is, indeed, reflected in the measurements. Such structural IP information could, therefore, be used in combination with other techniques in a general-purpose spam mitigation system, and this information is likely to be useful by itself only when an aggressive and computationally-efficient technique is needed.

The remainder of the chapter is structured as follows. We present our analysis of characteristics of IP addresses and network-aware clusters that distinguish between legitimate mail and spam in Sections 5.2 and 5.3 respectively. We present and evaluate our solution for protecting mail servers under overload in Section 5.4. We review related work in Section 5.5 and conclude in Section 5.6.

5.2 Analysis of IP-Address Characteristics

In this section, we explore the extent to which IP-based identification can be used to distinguish spammers from senders of legitimate e-mail based on differences in patterns of behaviour.

5.2.1 Data

Our data consists of traces from the mail server of a large company serving one of its corporate locations with approximately 700 mailboxes, taken over a period of 166 days from January to June 2006. The location runs a PostFix mail server with extensive logging that records the following: (a) every attempted SMTP connection, with its IP address and time stamp, (b) whether the connection was rejected, along with a reason for rejection, (c) if the connection was accepted, results of additional mail server's local spam-filtering tests, and if accepted for delivery, the results of running SpamAssassin.

Fig. 5.1(a) shows a daily summary of the data for six months. It shows four quantities for each day: (a) the number of SMTP connection requests made (including those that are denied via blacklists), (b) the number of e-mails received by the mail server, (c) the number of e-mails that were sent to SpamAssassin, and (d) the number of e-mails deemed legitimate by SpamAssassin. The relative sizes of these four quantities on every day illustrate the scale of the problem: spam is 20 times larger than the legitimate mail received. (In our data set, there were 1.4 million legitimate messages and 27 million spam messages in total.) Such a sharp imbalance indicates the potential of a significant role for applications like maximizing legitimate mail accepted when the server is overloaded: if there is a way to prioritize legitimate mail, the server could handle it much more quickly, because the volume of legitimate mail is tiny in comparison to spam.

In the following analysis, every message that is considered legitimate by SpamAssassin is counted as a legitimate message; every message that is considered spam by SpamAssassin, the mail server's local spam-filtering tests, or through denial by a blacklist is counted as spam.

5.2.2 Analysis of IP Addresses

We first explore the behaviour of individual IP addresses that send legitimate mail and spam, with the goal of uncovering any significant differences in their behavioral patterns.

Our analysis focuses on the *IP spam-ratio* of an IP address, which we define to be the fraction of mail sent by the IP address that is spam. This is a simple, intuitive metric that captures the spamming behaviour of an IP address: a low spam-ratio indicates that the IP address sends mostly legitimate mail; a high spam-ratio indicates that the IP address sends mostly spam. Our goal is to see whether the historical communication behaviour of IP addresses categorized by their spam-ratios can differentiate between IP addresses of legitimate senders and spammers, for spam mitigation.

As discussed earlier, the differentiation between the legitimate senders and spammers need not be perfect; there are benefits to having even a partial differentiation, especially with a simple, computationally inexpensive feature. For example, in the server overload problem, when all the mail cannot be accepted, a partial separation would still help to increase the amount of legitimate mail that is received.

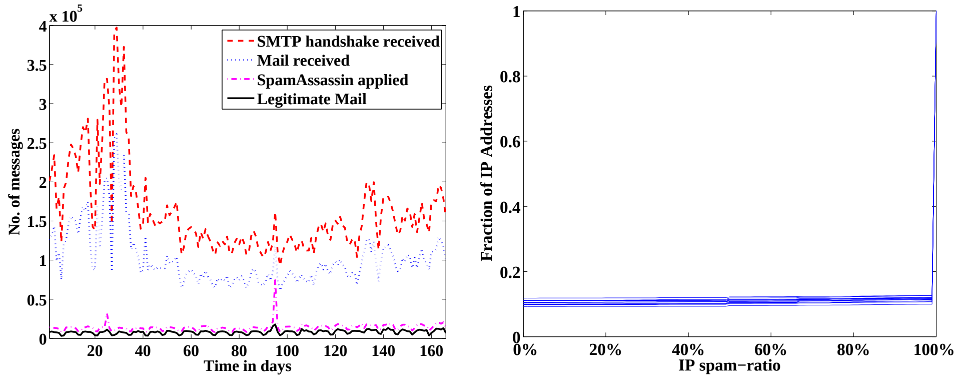
In the IP-based analysis, we will address the following questions:

- *Distribution by IP Spam Ratio*: What is the distribution of IP addresses by their spam-ratio, and what fraction of legitimate mail and spam is contributed by IP addresses with different spam-ratios?
- *Persistence*: Are IP addresses with low/high spam-ratios present across long time periods? If they are, do such IP addresses contribute to a significant fraction of the legitimate mail/spam?
- *Temporal Spam-Ratio Stability*: Do many of the IP addresses that appear to be good on average fluctuate between having very low and very high spam-ratios?

The answers to these three questions, taken together, give us an idea of the benefit we could derive in using the history of IP address behaviour in spam mitigation. We show in Sec. 5.2.2, that most IP addresses have a spam-ratio of 0% or 100%, and also that a significant amount of the legitimate mail comes from IP addresses whose spam-ratio exceeds zero. In Sec. 5.2.2, we show that a very significant fraction of the legitimate mail comes from IP addresses that persist for a long time, but only a tiny fraction of the spam comes from IP addresses that persist for a long time. In

Sec. 5.2.2, we show that most IP addresses have a very high temporal ratio-stability – they do not fluctuate between exhibiting a very low or very high daily spam-ratio over time.

Together, these three observations suggest that *identifying IP addresses with low spam ratios that regularly send legitimate mail* could be useful in spam mitigation and prioritizing legitimate mail. In the rest of this section, we present the analysis that leads to these observations. For concreteness, we focus on how the analysis can help spam mitigation in the server overload problem.



(a) Data characteristics

(b) CDFs of IP spam-ratios for many days: each line is a CDF for a different day.

Figure 5.1: 1(a): Daily summary of the data set over 6 months. 1(b): CDFs of IP spam-ratios for many different days.

Distribution by IP Spam-Ratio

In this section, we explore how the IP addresses and their associated mail volumes are distributed as a function of the IP spam-ratios. We focus here on the spam-ratio computed over a short time period in order to understand the behaviour of IP addresses without being affected by their possible fluctuations in time. Effectively, this analysis shows the limits of the differentiation that could be achieved by using IP spam-ratio, even assuming that IP spam-ratio could be predicted for a given IP address over short periods of time. In this section, we focus on day-long intervals, in order to take into account possible time-of-day variations. We refer to the IP spam-ratio computed over a day-long interval as the *daily spam-ratio*.

Intuitively, we expect that most IP addresses either send mostly legitimate mail, or mostly spam, and that most of the legitimate mail and spam comes from these IP addresses. If this hypothesis holds, then for spam mitigation, it will be sufficient if we can identify the IP addresses as senders of legitimate mail or spammers. To test this hypothesis, we analyze the following two empirical distributions: (a) the distribution of IP addresses as a function of the spam-ratios, and (b) the distribution of legitimate mail/spam as a function of their respective IP addresses' spam-ratio.

We first analyze the distribution of IP addresses by their daily spam-ratios in Fig. 5.1(b). For each day, it shows the empirical cumulative distribution function (CDF) of the daily spam-ratios of individual IP addresses active on that day. Fig. 5.1(b) shows this daily CDF for a large number of randomly selected days across the observation period.

Result 1. Distribution of IP addresses: *Fig. 5.1(b) indicates: (i) Most IP addresses, send either mostly spam or mostly legitimate mail. (ii) Fewer than $1 - 2\%$ of the active IP addresses have a spam-ratio of between $1\% - 99\%$, i.e., there are very few IP addresses that send a non-trivial fraction of both spam and legitimate mail. (iii) Further, the vast majority (nearly 90%) of IP addresses on any given day generate almost exclusively spam, and have spam-ratios between $99\% - 100\%$.*

The above results indicate that identifying IP addresses with low or high spam-ratios could identify most of the legitimate senders and spammers. In addition, for some applications (e.g., the mail server overload problem), it would be valuable to identify the IP addresses that send the bulk of the spam or the bulk of the legitimate mail, in terms of mail volume. To do so, we next explore how the daily legitimate mail or spam volumes are distributed as a function of the IP spam-ratios, and the resulting implications.

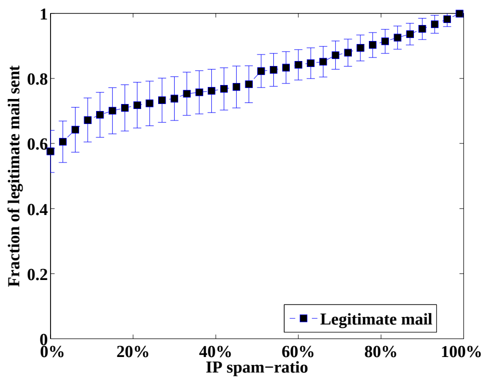
Let I_k denote the set of all IP addresses that have a spam-ratio of at most k. Fig. 5.2 examines how the volume of legitimate mail and spam sent by the set I_k depends on the spam-ratio k. Specifically, let $L_i(k)$ and $S_i(k)$ be the fractions of the total daily legitimate mail and spam that comes from all IPs in the set I_k, on day i. Fig. 5.2(a) plots $L_i(k)$ averaged over all the days, along with confidence intervals. Fig. 5.2(b) shows the corresponding distribution for the spam volume $S_i(k)$.

Result 2. Distribution of legitimate mail volume: *Fig. 5.2(a) shows that the bulk of the legitimate mail (nearly 70% on average) comes from IP addresses with a very low spam-ratio ($k \leq 5\%$). However, a modest fraction (over 7% on average) also comes from IP addresses with a high spam-ratio ($k \geq 80\%$).*

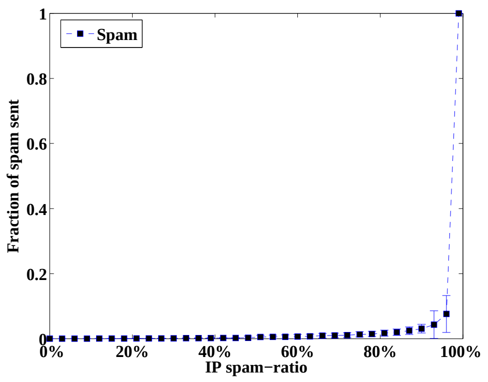
Result 3. Distribution of spam volume: *Fig. 5.2(b) indicates that almost all (over 99% on average) of the spam sent every day comes from IP addresses with an extremely high spam-ratio (when $k \geq 95\%$). Indeed, the contribution of the IP addresses with lower spam-ratios ($k \leq 80\%$) is a tiny fraction of the total.*

We observe that the distribution of legitimate mail volume as a function of the spam-ratio k is more diffused than the distribution of spam volume. There are two possible explanations for such behaviour of the legitimate senders. First, spam-filtering software tends to be conservative, allowing some spam to marked as legitimate mail. Second, a lot of legitimate mail tends to come from large mail servers that cannot do perfect outgoing spam-filtering. These mail servers may, therefore, have a slightly higher IP spam-ratio, and this would cause the distribution of legitimate mail to be more diffused across the spam-ratio.

Together, the above results suggest that the IP spam-ratio may be a useful discriminating feature for spam mitigation As an example, assume that we have a classification function that accepted



(a) Legitimate mail



(b) Spam

Figure 5.2: Legitimate mail and spam contributions as a function of IP spam-ratio.

(or prioritized) all IP addresses with a spam-ratio of at most k and rejected all IP addresses with a higher spam-ratio. Then, if we set $k = 95\%$, we could accept (or prioritize) nearly all the legitimate mail, and no more than 1% of the spam. However, such a classification function requires perfect knowledge of every IP address's daily spam-ratio every single day, and in reality, this knowledge may not be available.

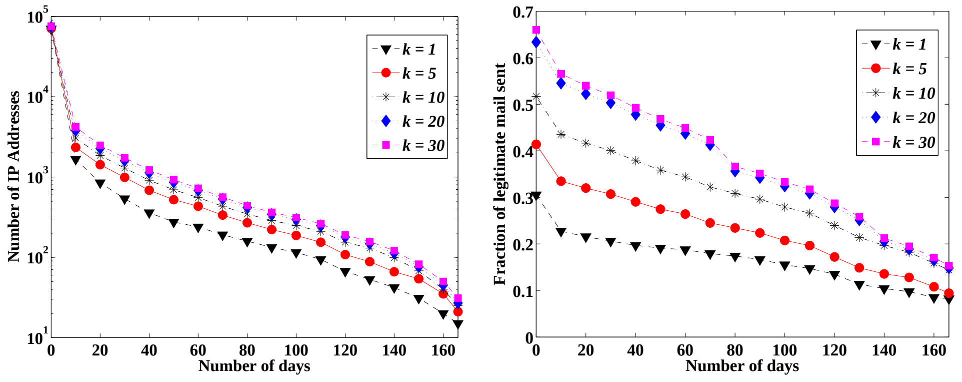
Instead, our approach is to identify properties that occur over longer periods of time, and are useful for predicting the current behaviour of an IP address based on long-term history, and these properties are incorporated into classification functions. The effectiveness of such history-based classification functions for spam mitigation depends on the extent to which IP addresses are long-lived, how much of the legitimate email or spam are contributed by the long-lived IP addresses, and to what extent the spam-ratio of an IP address varies over time. Sec. 5.2.2 and Sec. 5.2.2 explore these questions.

For the following analysis, we focus on the spam-ratio of each individual IP address, computed over the entire data set, since we are interested in its behaviour over its lifetime. We refer to this as the *lifetime spam-ratio* of the IP address. We show the presence of two properties in this analysis: (i) a significant fraction of legitimate mail comes from good IP addresses that last for a long time (*persistence*), and (ii) IP addresses that are good on average tend to have a low spam-ratio each time they appear (*temporal stability*). These two properties directly influence how effective it would be to use historical information for determining the likelihood of spam coming from an individual IP address.

Persistence

Due to the community structure inherent in non-spam communication patterns, it seems reasonable that most of the legitimate mail will originate from IP addresses that recur frequently. Previous studies have also indicated that most of the spam comes from IP addresses that are extremely short-lived. These suggest the existence of a potentially significant difference in the behaviour of senders of legitimate mail and spammers with respect to persistence. We next quantify the extent to which these hypotheses hold, by examining the persistence of individual IP addresses.

Our methodology for understanding the persistence behavior of IP addresses is as follows: we consider the set of all IP addresses with a low lifetime spam-ratio and examine how much legitimate mail they send, as well as how much of the legitimate mail is sent by IP addresses that are present for a long time. Such an understanding can indicate the potential of using a whitelist-based approach for prioritizing legitimate mail. If, for instance, the bulk of the legitimate mail comes from IP addresses that last for a long time, we could use this property to prioritize legitimate mail from long-lasting IP addresses with low spam-ratios.



(a) Number of k-good IP addresses present for x or more days (b) Fraction of legitimate mail sent by k-good IP addresses present for x or more days

Figure 5.3: Persistence of k-good IP addresses.

For this analysis, we use the following two definitions.

Definition 1. *A* k-**good IP address** *is an IP address whose lifetime spam-ratio is at most* k. *A* k-**good set** *is the set of all* k*-good IP addresses. Thus, a* 20*-good set is the set of all IP addresses whose lifetime spam-ratio is no more than* 20%.

We compute (a) the number of k-good IP addresses present for at least x distinct days, and (b) the fraction of legitimate mail contributed by k-good IP addresses that are present in at least x distinct days. [1] Fig. 5.3(a) shows the number of IP addresses that appear in at least x distinct days, for several different values of k.

[1]Our analysis considers persistence of IP addresses only in our data set, i.e., it considers whether the IP address has sent mail for x days to our mail server. These IP addresses may have sent mail to other mail servers on more days, and combining data across multiple different mail servers may give a better picture of stablility of IP addresses sending mail. Nevertheless, in this work, we focus on the persistence in one data set, as it highlights behavioural differences due to community structure present within a single vantage point.

Fig. 5.3(b) shows the fraction of the total legitimate mail that originates from IP addresses that are in the k-good set and appear in at least x days, for each threshold k.

Most of the IP addresses in a k-good set are not present very long, and the number of IP addresses falls quickly, especially in the first few days. However, their contribution to the legitimate mail drops much more slowly as x increases. The result is that the few longer-lived IPs contribute to most of the legitimate mail from a k-good set. For example, only 5% of all IP addresses in the 20-good set appear at least 10 distinct days, but they contribute to almost 87% of all legitimate mail for the 20-good set. If the k-good set contributes to a significant fraction of the legitimate mail, then the few longer-lived IP addresses also contribute significantly to the total legitimate mail. For instance, IP addresses in the 20-good set contribute to 63.5% of the total legitimate mail received. Only 2.1% of those IP addresses are present for at least 30 days, but they contribute to over 50% of the total legitimate mail received.

Result 4. Distribution of legitimate mail from persistent k-good IPs: *Fig. 5.3 indicates that (i) IP addresses with low lifetime spam ratios (small k) tend to contribute a major proportion of the total legitimate email, and (ii) only a small fraction of the IP addresses with a low lifetime spam-ratio addresses appear over many days, but they contribute to a significant fraction of the legitimate mail.*

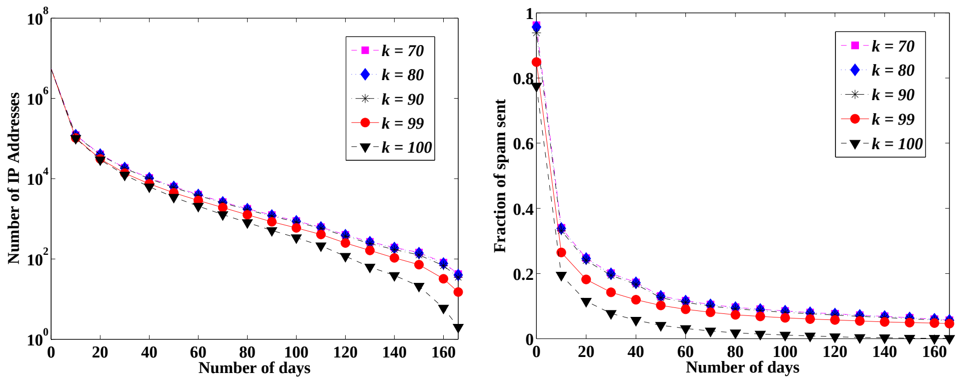
The graphs also reveal another trend: the longer an IP address lasts, the more stable is its contribution to the legitimate mail. For example, 0.09% of the IP addresses in the 20-good set are present for at least 60 days, but they contribute to over 40% of the total legitimate mail received. From this, we can infer that there were an additional 1.2% of IP addresses in the 20-good set that were present for 30-59 days, but they only contributed to 10% of the total legitimate mail received.

Fig. 5.4 presents a similar analysis of persistence for IP addresses with a high lifetime spam-ratio. Like the k-good IP addresses and k-good sets, we define k-bad IP addresses and k-bad sets.

Definition 2. *A* **k-bad IP address** *is an IP address that has a lifetime spam-ratio of at least k. A* **k-bad set** *is the set of all k-bad IP addresses.*

Fig. 5.4(a) presents the number of IP addresses in the k-bad set that are present in at least x days, and Fig. 5.4(b) presents the fraction of the total spam sent by IP addresses in the k-bad set that are present in least x days.

Result 5. Distribution of spam from persistent k-bad IPs: *Fig. 5.4 indicates that (i) IP addresses with high lifetime spam ratios (large k) tend to contribute almost all of the spam, (ii) most of these high spam-ratio IPs are only present for a short time (this is consistent with the finding in [99]) and account for a large proportion of the overall spam, and (iii) the small fraction of these IPs that do last several days contribute a non-trivial fraction of the overall spam; however, a much larger fraction of spam comes from IP addresses that are not present for very long. As in the case of the k-good IP addresses, the spam contribution from the k-bad IP addresses tends to get more stable with time.*



(a) No. of k-bad IP addresses present in x or more days (b) Fraction of spam sent by k-bad IP addresses present in x or more days

Figure 5.4: Persistence of k-bad IP addresses.

So, for instance, we can see from Fig. 5.4 that only 1.5% of the IP addresses in the 80-bad set appear in at least 10 distinct days, and these contribute to 35.4% of the volume of spam from the 80-bad set, and 34% of the total spam. The difference is more pronounced for 100-bad IP addresses: 2% of the 100-bad IP addresses appear for 10 or more distinct days, and contribute to 25% of the total spam volume.

The results of this section have implications in designing spam filters, especially for applications where the goal is to prioritize legitimate mail rather than discard spam. While spamming IP addresses that are present sufficiently long can be blacklisted, the scope of a purely blacklisting approach is limited. On the other hand, a very significant fraction of the legitimate mail can be prioritized by using the history of the senders of legitimate mail.

Temporal Stability

Next, we seek to understand whether IP addresses in the k-good set change their daily spam-ratio dramatically over the course of their lifetime. The question we want to answer is: of the IP addresses that appear in a k-good set (for small values of k), what fraction of them have ever had "high" daily spam-ratios, and how often do they have "high" spam-ratios? Thus, we want to understand the *temporal stability* of the spam-ratio of IP addresses in k-good sets. In this section, we focus on k-good IP addresses; the results for the k-bad IP addresses are similar.

We compute the following metric: for each IP address in a k-good set, we count how often its daily spam-ratio exceeds k (and normalize this count by the number of days it appears). We define this quantity to be the *frequency-fraction excess* of the IP address, for the k-good set. We plot the

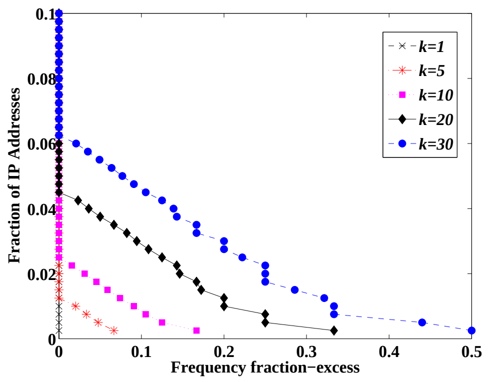


Figure 5.5: Temporal stability of IP addresses in k-good sets, shown by CCDF of frequency-fraction excess.

complementary cdf (CCDF) of the *frequency-fraction excess* of all IP addresses in the k-good set. [2] Intuitively, the distribution of the frequency-fraction excess is a measure of how many IP addresses in the k-good set exceed k, and how often they do so.

Fig. 5.5 shows the CCDF of the frequency-fraction excess for several k-good sets. It shows that the majority of the IP addresses in each k-good set have a frequency-fraction excess of 0, and that 95% of the k-good IP addresses have a frequency-fraction excess of at most 0.1.

We explain the implications of Fig. 5.5 to the temporal stability of the spam-ratio of IP addresses with an example. We focus on the k-good set for $k = 20$: this is the set of IP addresses whose lifetime spam-ratio is bounded by 20%. We note that the frequency-fraction excess is 0 for 95% of the 20-good IP addresses. This implies that 95% of IP addresses in this k-good set do not send more than 20% spam *any* day, i.e., every time they appear, they have a daily spam-ratio of at most 20%. We also note that fewer than 1% of the IP addresses in this k-good set have a frequency-fraction excess larger than 0.2.

Thus, for many k-good sets with small k-values, only a few IP addresses have a significant frequency-fraction excess, i.e., very few IP addresses in those sets exceed the value k often. Since they would need to exceed k often to change their spamming behaviour significantly, it follows that most IP addresses in the k-good set do not change their spamming behaviour significantly.

In addition, the frequency-fraction excess is perhaps too strict a measure, since it is affected even if k is exceeded slightly. We also compute a similar measure that increases only when k is exceeded by 5%. No more than 0.01% of IP address in the k-good set exceed k by 5%, for any $k \leq 30\%$. Since we are especially interested in the temporal stability of IP addresses that appear often, we compute also the frequency-fraction excess distribution for IP addresses that appear for

[2]That is, we plot the fraction of IP addresses in the k-good set whose frequency-fraction excess is at least x. The y-axis of the plot is restricted for readability.

10, 20, 40 and 60 days. In each case, almost no IP address exceeds k by more than 5%, for any $k \leq 30\%$.

We summarize this discussion in the following result.

Result 6. Temporal stability of k-good IPs: *Fig. 5.5 shows that most IP addresses in k-good sets (for low k, e.g., $k \leq 30\%$) do not exceed k often; i.e., most k-good IP addresses have low spam-ratios (at most k) nearly every day.*

With the above result, we can analyze the behaviour of k-good sets of IP addresses, constructed over their entire lifetime, and their behaviour in shorter time intervals.

The analysis of these three properties of IP addresses indicates that a significant fraction of the legitimate mail comes from IP addresses that persistently appear in the traffic. These IP addresses tend to exhibit stable behaviour: they do not fluctuate significantly between sending spam and legitimate mail. These results lend weight to our hypothesis that spam mitigation efforts can benefit by preferentially allocating resources to the stable and persistent senders of legitimate mail. However, there is still a substantial portion of the mail that cannot be accounted for through only IP address-based analysis. In the next section, we focus on how to account for this mail.

5.3 Analysis of Cluster Characteristics

So far, we have analyzed whether the historical behaviour of individual IP addresses can be used to distinguish between senders of legitimate mail and spammers. However, if we only consider the history of individual IP addresses, we cannot determine whether a new, previously unseen, IP address is likely to be a spammer or a sender of legitimate mail. If there are many such IP addresses, then, in order to be useful, any prioritization scheme would need to assign these new IP addresses appropriate reputations as well. Indeed, in Sec. 5.2.2, we found that most IP addresses sending mail are short-lived and that such short-lived IPs account for a significant proportion of both legitimate mail and spam. Any prioritization scheme would thus need to be able to find reputations for these IP addresses as well.

To address this issue, we explore whether coarser aggregations of IP addresses exhibit more persistence and afford more effective discriminatory power for spam mitigation. If such aggregations of IP addresses can be found, the reputation of an unseen IP address could be *derived* from the historical reputation of the aggregation they belong to.

We focus on IP aggregations given by *network-aware clusters* of IP addresses [72]. Network-aware clusters are sets of unique network IP prefixes collected from a wide set of BGP routing table snapshots. In this work, an IP address belongs to a network-aware cluster if the longest prefix match of the IP address matches the prefix associated with the cluster. In the reputation mechanisms we explore in Sec. 5.4, an IP address derives the reputation of the network-aware cluster that it belongs to. We use network-aware clustering because these clusters represent IP addresses that are close in terms of network topology and do, with high probability, represent regions of the IP space that are under the same administrative control and share similar security and spam policies [72].

In this section, we present measurements suggesting that network-aware clusters of IP addresses may provide a good basis for reputation-based classification of IP addresses. We focus on the following questions:

- *Granularity*: Does the mail originating from network-aware clusters consist of mostly spam or mostly legitimate mail, so that these clusters could be useful as a reputation-granting mechanism for IP addresses?

- *Persistence*: Do individual network-aware clusters appear (i.e., do IP addresses belonging to the clusters appear) over long periods of time, so that network-aware clusters could potentially afford us a useful mechanism to distinguish between different kinds of ephemeral IP addresses?

As in the IP-address case, we adopt the spam-ratio of a network-aware cluster as the discriminating feature of clusters and examine whether clusters with low/high spam-ratios are granular and persistent.

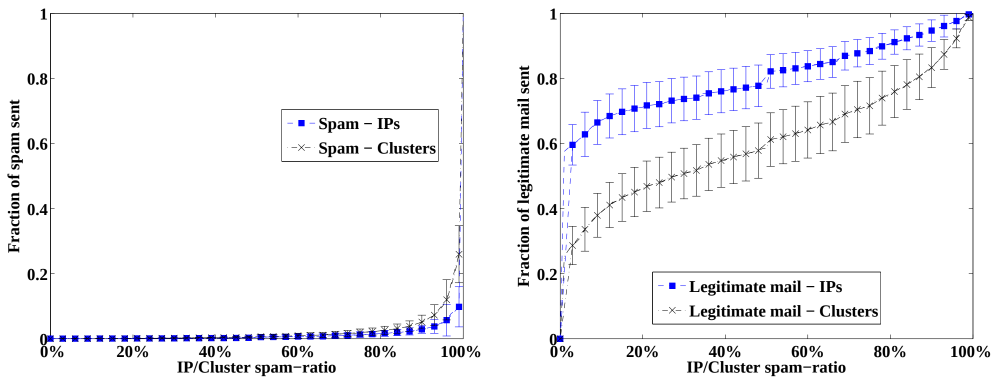
Before examining these two properties in detail, we first summarize our analysis of the properties with respect to which clusters behave as IP addresses do: *clusters turn out to be at least as (and usually more) temporally stable as IP addresses* (similar to the IP address behaviour explored in Sec. 5.2.2), which is the expected behaviour; *the distribution of clusters by daily cluster spam-ratio is similar to the distribution of IP addresses by IP spam-ratio* (similar to the IP address behaviour explored in Sec. 5.2.2).

5.3.1 Cluster Granularity

For network-aware clustering of IP addresses to be useful, the clusters need to be sufficiently homogeneous in terms of their legitimate mail/spam behavior so that the cluster information can be used to separate the bulk of legitimate mail from the bulk of spam. Recall that with the IP addresses, we analyzed the extent to which IP spam-ratios could be used to identify the IP addresses sending the bulk of legitimate mail and spam. Here, we analyze whether, instead of an IP's individual spam-ratio, the spam-ratio of the parent cluster can be used for the same purpose.

To do so, we need to understand how well the cluster spam-ratio approximates the IP spam-ratio. In our context, we focus on the following question: can we still distinguish between the IP addresses that send the bulk of the legitimate mail and the bulk of the spam? If we can, within a margin of error, it would suggest that cluster-level analysis is nearly as good as IP-level analysis.

For the analysis here, we determine the spam-ratio of each cluster by analyzing the mail sent by all IP addresses belonging to that cluster and assign to IP addresses the spam-ratios of their



(a) Fraction of spam sent by clusters & IPs, as a function of cluster & IP spam-ratios.

(b) Legitimate mail sent by clusters & IPs, as a function of cluster & IP spam-ratios.

Figure 5.6: Penalty of using cluster-level analysis.

respective clusters. In the rest of this discussion, we will refer to legitimate mail/spam sent by IP addresses belonging to a cluster as the legitimate mail/spam *sent by* or *coming from* that cluster. As with the IP-based analysis, we examine how the volume of legitimate mail and spam from IP addresses is distributed as a function of their cluster spam-ratios. To understand the additional error imposed by using the cluster spam-ratio, we compare it with how those volumes are distributed as a function of the IP spam-ratio.

Fig. 5.6(a) shows how the spam sent by IP addresses with a cluster or IP spam-ratio of at most k varies with k. Specifically, on day i, let $CS_i(k)$ and $IS_i(k)$ be the fraction of spam sent by the IP addresses with a cluster spam-ratio (and IP spam-ratio, respectively) of at most k. Fig. 5.6(a) plots $CS_i(k)$ and $IS_i(k)$ averaged over all the days in the data set, as a function of k, along with confidence intervals.

Result 7. Distribution of spam with cluster and IP spam-ratios: *Fig. 5.6(a) shows that almost all (over 95%) of the spam every day comes from IPs in clusters with a very high cluster spam-ratio (over 90%). A similar fraction (over 99% on average) of the spam every day comes from IP addresses with a very high IP spam-ratio (over 90%).*

This suggests that spammers responsible for a high volume of the total spam may be closely correlated with the clusters that have a very high spam-ratio. The graph indicates that if we use a spam-ratio threshold of $k \leq 90\%$ for spam mitigation, then using the IP spam-ratio rather than the corresponding cluster spam-ratio as the discriminating feature would increase the amount of spam identified by less than 2%. This suggests that cluster spam-ratios are a good approximation to IP spam-ratios for identifying the bulk of the spam sent.

We next consider how legitimate mail is distributed with the cluster spam-ratios and compare it with IP spam-ratios (Fig. 5.6(b)). We compute the following metric: Let $CL_i(k)$ and $IL_i(k)$ be the fraction of legitimate mail sent by IPs with cluster and IP spam-ratios of at most k on day i. Fig. 5.6(b) plots $CL_i(k)$ and $IL_i(k)$ averaged over all the days in the data set as a function of k, along with confidence intervals.

Result 8. Distribution of legitimate mail with cluster and IP spam-ratios: *Fig. 5.6(b) shows that a significant amount of legitimate mail is contributed by clusters with both low and high spam-ratios. A significant fraction of the legitimate mail (around 45% on average) comes from IP addresses with a low* cluster spam-ratio *($k \leq 20\%$). However, a much larger fraction of the legitimate mail (around 70%, on average) originates from IP addresses with a similarly low* IP spam-ratio.

The picture here, therefore, is much less promising: even when we consider spam-ratios as high as $30 - 40\%$, the cluster spam-ratios can only distinguish, on average, around 50% of the legitimate mail. By contrast, IP spam-ratios can distinguish as much as 70%. This suggests that IP addresses responsible for the bulk of legitimate mail are much less correlated with clusters of low spam-ratio.

We can then make the following conclusion: suppose we use a classification function to accept or reject IP addresses based on their cluster spam-ratio. What additional penalty would we incur over a similar classification function that used the IP address's own spam-ratio? Fig. 5.6(b) suggests that, if the threshold is set to 90% or higher, we incur very little penalty in both legitimate mail acceptance and spam. However, if the threshold is set to $30 - 40\%$, we may incur as much as a 20% penalty in doing so.

However, there are two additional ways in which such a classification function could be enhanced. First, as we have seen, the bulk of the legitimate mail does come from persistent k-good IP addresses. This suggests that we could potentially identify more legitimate mail by considering the persistent k-good IP addresses *in addition* to cluster-level information. Second, for some applications, the correlation between high cluster spam-ratios and the bulk of the spam may be sufficient to justify using cluster-level analysis. For example, under the existing distribution of spam and legitimate mail, even a high cluster spam-ratio threshold would be sufficient to reduce the total volume of the mail accepted by the mail server. This is exactly the situation in the server overload problem and we see the effect in the simulations in Sec. 5.4.

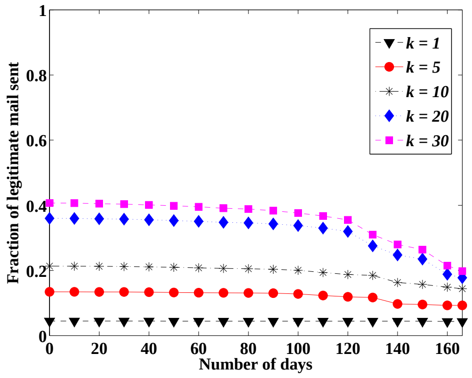
5.3.2 Persistence

Next, we explore how persistent the network-aware clusters are, just as we did for the IP addresses. We define a cluster to be *present* on a day if at least one IP address that belongs to that cluster appears that day. We reported earlier that we found the clusters themselves to be at least as (and usually more) temporally stable as IP addresses. Our next goal is to examine how much of the total legitimate mail/spam the long-lived clusters contribute.

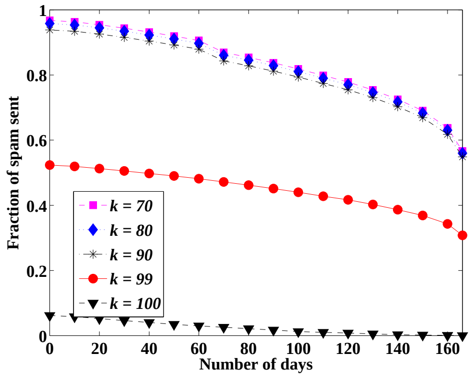
As in Sec. 5.2.2, we will define k-good and k-bad clusters; to do that, we use the *lifetime cluster spam-ratio*: the ratio of the total spam sent by the cluster to the total mail sent by it over its lifetime.

Definition 3. *A k-**good cluster** is a cluster of IP addresses whose lifetime cluster spam-ratio is* at most k. *The k-**good cluster-set** is the set of all k-good clusters. A k-**bad cluster** is a cluster of IP addresses whose lifetime cluster spam-ratio is* at least k. *The k-**bad cluster-set** is the set of all k-bad clusters.*

Fig. 5.7(a) examines the legitimate mail sent by k-good clusters for small values of k. We first note that the k-good clusters (even when k is as large as 30%) contribute less than 40% of the total legitimate mail; this is in contrast to, for instance, 20-good IP addresses that contributed to 63.5% of the total legitimate mail. However, we note the contribution from long-lived clusters is far more than from long-lived individual IPs. The difference from Fig. 5.3(b) is striking: e.g., k-good clusters present for 60 or more days contribute to nearly 99% of the legitimate mail from the k-good cluster set. So, any cluster accounting for a non-trivial volume of legitimate mail is present for at least 60 days. Indeed, the legitimate mail sent by k-good clusters drops to 90% of k-good cluster-set's total only when restricted to clusters present for 120 or more days; by contrast, for individual IP addresses, the legitimate mail contribution dropped to 87% of the 20-good set's total after just 10 days.



(a) Fraction of legitimate mail sent by k-good clusters that appear in at least x days



(b) Fraction of spam sent by k-bad clusters that appear in at least x days

Figure 5.7: Persistence of network-aware clusters.

Fig. 5.7(b) presents the same analysis for k-bad clusters. Again, there are noticeable differences from the k-bad IP addresses, and also from the k-good clusters. A much larger fraction of spam comes from long-lived clusters than from long-lived IPs in Fig. 5.4(b). For example, over 92% of the total spam is contributed by 90-bad clusters present for at least 20 days. This is in sharp contrast with the k-bad IP addresses, where only 20% of the total spam comes from IP addresses that last 20 or more days. We also note that the 90-bad cluster-set contributes to nearly 95% of the total spam. Thus, in contrast to the legitimate mail sent by k-good cluster-sets, the bulk of the spam comes from the k-bad cluster-sets with high k.

Result 9. Distribution of mail from persistent clusters: *Fig. 5.7 shows that the clusters that are present for long periods with high cluster spam-ratios contribute the overwhelming fraction of the spam sent, while those present for long periods with low cluster spam-ratios contribute a smaller, though still significant, fraction of the legitimate mail sent.*

The above result suggests that network-aware clustering can be used to address the problem of transience of IP addresses in developing history-based reputations of IP addresses: even if individual IP addresses are ephemeral, their (possibly collective) history would be useful in assigning reputations to other IP addresses originating from the same cluster.

5.4 Spam Mitigation under Mail Server Overload

In the previous section, we have demonstrated that there are significant differences in the historical behaviour of IP addresses that send a lot of spam, and those that send very little. In this section, we consider how these differences in behaviour could be exploited for spam mitigation.

Our measurements have shown that senders of legitimate mail demonstrate significant stability and persistence, while spammers do not. However, the bulk of the high volume spammers appear to be clustered well within many persistent network-aware clusters. Together, these suggest that we can design techniques based on the historical reputation of an IP address and the cluster to which it belongs. However, because mail rejection mechanisms necessarily need to be conservative, we believe that such a reputation-based mechanism is primarily useful for prioritizing legitimate mail, rather than actively discarding all suspected spammers.

As an application of these measurements, we now consider the mail-server overload problem described in the introduction. In this section, we demonstrate how the problem could be tackled with a reputation-based mechanism that exploits these differences in behaviour. In Sec. 5.4.1, we explain the mail-server overload problem in more detail. In Sec. 5.4.2, we explain our approach, describing the mail server simulation and algorithms that we use, and in Sec. 5.4.3, we present an evaluation showing the performance improvement gained using these differences in behaviour.

We emphasize that this simulation study is intended to demonstrate the potential of using these behavioural differences in the legitimate mail and spam for prioritizing exclusively by IP addresses. However, it is *not* intended to be comparable to content-based spam filtering. We also note that these differences in behaviour could be applied in other ways as well and at other points in the mail processing as well. The quantitative benefits that we achieve may be specific to our application and may be different in other applications.

5.4.1 Server Overload Problem

The problem we consider is the following: When the mail server receives more SMTP connections than it can process in a time interval, how can it selectively accept connections to maximize the acceptance of legitimate mail? That is, the mail server receives a sequence of connection requests from IP addresses every second, and each connection will send mail that is either legitimate or spam. Whether the IP address sends spam or legitimate mail in that connection is not known at the time of the request, but is known after mail is processed by the spam filter. The mail server has a finite capacity of the number of mails that can be processed in each time interval, and may choose the connections it accepts or rejects. The goal of the mail server is to selectively accept connections in order to maximize the legitimate mail accepted.

We note that spammers have strong incentive to cause mail servers to overload, and illustrate this with an example. Assume that a mail server can process 100 emails per second, that it will start dropping new incoming SMTP connections when its load reaches 100 emails per second, and that it crashes if the offered load reaches 200 emails per second. Assume also that 20 legitimate emails are received per second. A spammer could increase the load of the mail server to 100% by sending 80 emails per second which would be all received by the mail server. Alternatively, the spammer could also increase the load to 199%, by sending 179 spam emails per second, and now nearly half the requests would not be served. If the mail server is unable to distinguish between the spam requests and the legitimate mail requests, it drops connections at random, and the spammer will be able to successfully get through 89 spam emails per second to the mail server, as compared to the 80 in the previous case.

Thus, the optimal operation point of a spammer, assuming that he has a large potential sending capacity, is not the maximum capacity of the mail server but the maximum load before the mail server will crash. This observation indicates that the approach of throwing more resources at the problem would only work if the mail server capacity is increased to exceed the largest botnet available to the spammer. This is typically not economically feasible and a different approach is needed.

The results in Sec. 5.2 and Sec. 5.3 suggest that there may be a history-based reputation function R, that relates IP addresses to their likelihood of sending spam. Thus, for example, if $R(i)$ is the probability that an IP address i sends legitimate mail, then maximizing the quantity $\sum R(i)$ would maximize the expected number of accepted legitimate mail. If the reputation function R were known, this problem would be similar to admission control and deadline scheduling; however, in our case, R is not known.

In this work, we choose one *simple* history-based reputation function and demonstrate that it performs well. We reiterate that our goal is *not* to explore the space of the reputation functions or to find the best reputation function. Rather, our goal is to demonstrate that they could potentially be used to increase the legitimate mail accepted when the mail-server is overloaded. In addition, our goal is to preferentially accept e-mails from certain IP addresses *only* when the mail servers are overloaded – we would like to minimize the impact on mail servers when they are not overloaded. A poor choice of R will then not impact the mail server under normal operation.

The techniques and the reputation functions that we choose address concerns that are different from those addressed by standard IP-based classification techniques like blacklisting and greylisting, as neither blacklisting nor greylisting would directly solve the server overload problem. Blacklisting has well-known issues: building a blacklist takes time and effort, most IP addresses that send spam are observed to be ephemeral, appearing very few times, and many of them are not even present in any single blacklist.

While greylisting is an attractive short-term solution that has been observed to work quite well in practice, it is not robust to spammer evasion, since spammers could simply mimick the behaviour of a normal mail server. Greylisting aims to optimize a different goal – its goal is to delay the mail in the hope that a spam signature is generated in the mean time, so that spam can be distinguished from non-spam; however, delaying the mail does not reduce the overall server load, since the spammer can always return to send more mail, and computing a content-based spam signature would continue to be as expensive. Indeed, greylisting gives spammers even more incentive to overload mail servers by re-trying after a specified time period.

Our techniques for the server overload problem provide an additional layer of information when compared to blacklisting and greylisting. It may be possible to use the IP structure information to enhance greylisting, to decide, at finer granularities and with soft thresholding, which IP addresses to deny.

5.4.2 Design and Algorithms

Today, when mail servers experience overload, they drop connections greedily: the server accepts all connections until it is at maximum load, and then refuses all connection requests until its load drops below the maximum. We aim to improve the performance under overload by using information in the structure of IP addresses, as suggested by the results in Sec. 5.2 and Sec. 5.3. At a high-level, our approach is to obtain a history of IP addresses and IP clusters, and use it to select the IP addresses that we prioritize under overload. To explore the potential benefits of this approach, we simulate the mail server operation and allow some additional functionality to handle overload.

To motivate our simulation, we describe briefly the way many mail servers in corporations and ISPs operate. First, the sender's mail server or a mail relay tries to connect to the receiving mail server via TCP. The receiving mail server accepts the connection if capacity is available, and then the mail servers perform the SMTP handshake and transfer the email. The receiving mail server stores the email to disk and adds it to the spam processing queue. For each e-mail on the queue, the receiving mail server then performs content-based spam filtering [3, 1] which is typically the most expensive part of email processing. After this, the spam emails are dropped or delivered to a spam mailbox, and the good emails are delivered to the inbox of the recipient.

In our simulation we simplify the mail server model, while ensuring that it is still sufficiently rich to capture the problem that we explore. We believe that our model is sufficiently representative for a majority of mail server implementations used today; however, we acknowledge that there are mail server architectures in use which are not fully captured in our model. In the next section, we describe the simulation model in more detail.

Mail Server Simulation

We simulate mail-server operation in the following manner:

- *Phase 1:* When the mail server receives an SMTP connection request, it may decide whether or not to accept the connection. If it decides to accept the connection, the incoming mail takes t time units to be transferred to the mail server. Thus, if a server can accept k connection requests simultaneously, it behaves like a k-parallel processor in this phase. We do so because this phase models the SMTP handshake and transfer of mail, and therefore, it needs to model state for each connection separately.
- *Phase 2:* Once the mail has been received, it is added to a queue for spam filtering and delivery to the receiving mailbox if any. At each time-step, the mail server selects mails from this queue and processes them; the number of mails chosen depend on the mail server's capacity and the cost of each individual mail. Here, since we model computation cycles, a sequential processing model suffices. The mail server has a timeout: it discards any mail that has been in the queue for more than m time units. If the load has sufficient fluctuation, a large timeout would be useful, but we want to minimize timeout since email has the expectation of being timely.

We assume that the cost of denying/dropping a request is 0, the cost of processing the SMTP connection is α fraction of its total cost, and the cost of the remainder is $1 - \alpha$ fraction of the total cost. We also allow Phase 1 of the mail server simulator to have α fraction of the server's computational resources, and Phase 2 to have the remainder. Since the content-based analysis is typically the most expensive part of processing a message, we expect that α is likely to be small.

This two-phase simulation model allows for more flexibility in our policy design, since it opens the possibility of dropping emails which have already been received and are awaiting spam filtering without wasting too many resources.

Policies

Next, we present the prioritization/drop policies that we implemented and evaluated on the mail server simulator. In this simulation model, the default mail-server action corresponds to the following: at each time-interval, the server accepts incoming requests in the order of arrival, as long as it is not overloaded. Once mail has been received, the server processes the first mail in the queue, and discards any mail that has exceeded its timeout. We refer to this as the *greedy* policy.[3]

The space of policy options that a mail-server is allowed to operate determine the kinds of benefits it can get. In this problem, one natural option for the mail server is to decide immediately whether to accept or reject a connection request. However, such a policy may be quite sensitive to fluctuation in the workload received at the mail server. Another option may be to reject some e-mails *after* the SMTP connection has been accepted, but *before* any spam-filtering checks or content-based analysis (such as spam-filtering software) has been applied. Note that content-based analysis typically is the most computationally expensive part of receiving mail. Thus, with this option, the mail server may do a small amount of work for some additional emails that eventually get rejected, but is less affected by the fluctuation of mail arrival workload. We restrict the space of policy options to the time before *any* content-based analysis of the incoming mail is done.

To solve the mail-server overload problem, we implement the following policies at the two phases:

- *Phase-1 policy*: The policy in Phase 1 is designed to preferentially accept IP addresses with a good reputation when the server is near maximum load: as the server gets closer to overload, the policy only accepts IP addresses with better and better reputations. In addition, when the load is below some percentage (we choose 75%) of the total capacity, the server accepts all mail: this way, it minimizes impact on normal operation of the mail server. [4]

- *Phase-2 policy*: The scheduling policy here is easier to design, since the queue has some knowledge of what needs to be processed. Even a simple policy that greedily accepts the item with the highest reputation value will do well, as long as the reputation function is reasonably accurate. We use this greedy policy for Phase 2.

Our history-based reputation function R is simple: First, we find a list of persistent senders of legitimate mail from the same time period (we choose all senders that have appeared in at least 10 days), and for these IP addresses, we use their lifetime IP spam-ratio as their reputation value. For the remaining IP addresses, we use their cluster spam-ratio as their reputation value: for each week, we use the history of the preceding four weeks in computing the lifetime spam-ratio (defined over 4 weeks) for each cluster that sends mail. [5] In this way, we combine the results of the IP-based analysis and cluster-based analysis in Sec. 5.2 in designing the reputation function.

[3]To ensure that the current mail server policy is not unfairly modelled under this simulation model, we evaluated greedy policies in another simulation model, in which each connection took z time units to process from start to end. The performance of the greedy policy was similar, therefore we do not describe the model further.

[4]Technically, this is slightly more complex: it examines if the load is below 75% of the server capacity allowed to Phase 1.

[5]One technical detail left to consider are the IP addresses originating from clusters without history. In our reputation function, any IP address that has no history-based reputation value is given a slightly bad reputation.

This reputation function is extremely simple, but it still illustrates the value of using a history-based reputation mechanism to tackle the mail server overload problem. We also note that the historical IP reputations based on network-aware clusters in this manner may not always be perfect predictors of spamming behaviour. While network-aware clusters are an aggregation technique with a basis in network structure, they could serve as a starting point for more complex clustering techniques, and these techniques may also incorporate finer notions of granularity and confidence.

A more sophisticated approach to using the history of IP addresses and network-aware clusters that addresses these concerns is likely to yield an improvement in performance, but is beyond the scope of our work. In the following section, we describe the performance benefits that we gain from using this reputation function in the evaluation.

5.4.3 Evaluation

We evaluate our history-based policies by replaying the traces of our data set on our simulator. Since the traces record each connection request with a time-stamp, we can replay the traces to simulate the exact workload received by the mail server. We do so, with the simplifying assumption that each incoming e-mail incurs the same computational cost. Since our traces are fixed, we simulate overload by decreasing the simulated server's capacity, and replaying the same traces. This way, we do not change the distribution and connection request times of IP addresses in the input traces between the different experiments. At the same time, it allows us to simulate, without changing the traces, how the mail server behaves as a function of the increasing workload.

Simulation Parameters: We now explain the parameters that we choose for our simulation. We choose the time t for the Phase 1 operation to be $4s$.[6] We use 60 seconds for the timeout m, the waiting time in the queue before Phase 2 (it implies that mail will be delivered within 1 minute, or discarded after Phase 1). This appears to be sufficiently small so as to not noticeably affect the delivery of legitimate mail. [7]

To induce overload, we vary the capacity of the simulated mail server to 200, 100, 66, 50, and 40 messages/minute. The greedy policy processed an average of 95.2% of the messages received when the server capacity was set to 200 messages/minute, as seen in Table 2. At capacities larger than 200 messages/minute, the number of messages processed by the greedy policy grows very slowly, indicating that this is likely to be an effect of the distribution of connection requests in the traces. For this reason, we take capacity of 200/minute as the required server capacity. We then refer to the other server capacities in relation to required server capacity for this trace workload: a server with capacity of 100 messages/minute must process the same workload with half the

[6]We vary t for Phase 1 between 2-4s: our traces have a recorded time granularity of 1s, and the maximum seen in the traces before a disconnect was 4s. This does not appear to impact the results presented here, since both kinds of policies receive the same value of t. We present in the results for $t = 4$ seconds

[7]This value also has no noticeable impact on our results when $m \geq 20s$ suggesting that most of the legitimate mail is processed quickly, or not at all.

capacity of the required server, so we define it to have an *overload-factor* of 2. Likewise, the server capacities we test 200, 100, 66, 50 and 40 messages/minute have overload-factors of around 1, 2, 3, 4, and 5 respectively.

Recall that the parameter α is the cost of processing the message at Phase 1. We expect α to impact the performance, so we test two values $\alpha = 0.1, 0.5$ in the evaluation; recall that α is likely to be small, and so $\alpha = 0.5$ is a conservative choice here. The value of α has no effect on the performance of the greedy policy. For this reason, the discussion features only one greedy policy for all values of α. For the history-based policies, α sometimes has an effect on the performance, since these policies allow for a decision to be taken at Phase 2. We therefore refer to the history-based policies as 10-policy, and 50-policy, for $\alpha = 0.1$ and 0.5 respectively.

Impact on Legitimate mail

We first compare the number of legitimate mails accepted by the different policies over many time intervals, where each interval is an hour long. Since our goal is to maximize the amount of legitimate mail accepted, the primary metric we use is the *goodput ratio*: the ratio of legitimate mail accepted by the mail server to the total legitimate mail in the time interval. This is a natural metric to use, since it makes the different time intervals comparable, and so we can see if the policies are consistently better than the greedy policy, rather than being heavily weighted by the number of legitimate mails in a few time intervals. For the performance evaluation, we examine the average goodput ratio, the distribution of the goodput ratios and the goodput improvement factor.

Average Goodput Ratio: Table 1 shows the average goodput ratios for the different policies under different levels of overload. It shows that, on average, for each of these overloads, the goodput of any of the policies is better than the greedy policy. The difference is marginal at overload-factor 1, and increases quickly as the overload-factor increases: at overload-factor 4, the average goodput ratio is $64.3 - 64.5\%$ for any of the history-based policies, in comparison to 26.8% for the greedy policy. We also observe that the history-based policies scale more gracefully with the overload. Thus, we conclude that, on average, the history-based policies gain a significant improvement over the greedy policy.

Distribution of Goodput Ratios: While the average goodput ratio is a useful summarization tool, it does not give a complete picture of the performance. For this reason, we next compare the *distribution* of the server goodput in the different time intervals. Fig. 5.8(a)-(b) shows the CDF of the goodput ratios for the different policies, for two overload-factors: 1 and 4. We observe that the goodput ratio distributions are quite similar for the greedy and history-based policies when the overload-factor is 1 (Fig. 5.8(a)): about 60% of the time, all of the policies accept 100% messages. This changes drastically as the overload-factor increases. Fig. 5.8(b) shows the goodput ratio distributions for overload-factor 4. As much as 50% of the time, the greedy policy has a goodput-ratio of at most 0.25. By contrast, more than 90% of the time, the history-based policies have a goodput ratio of at least 0.5. The results show that the history-based policies have a consistent and significant improvement over the greedy policy when the load is sufficiently high.

Improvement factor of Goodput-Ratios: Finally, we compare the goodput ratios on a per-interval basis. For this analysis, we focus on the 10-policy; our goal is to see *how often* the 10-policy does better than the greedy algorithm. That is, for each time interval, we compute the *goodput-factor*, defined to be $\frac{\text{Goodput of 10-Policy}}{\text{Goodput of Greedy}}$. Fig. 5.8(c) plots how often goodput-factor lies between $90\% - 300\%$ for the different overload-factors. We note that when the overload-factor is 1, the performance impact of our history-based policy on the legitimate mail is marginal: in all the time intervals, the 10-policy has a goodput-factor of at least 90%, and over 95% of the time, it has a goodput factor of at least 99%. As the overload-factor increases, the amount of time intervals in which the 10-policy has a goodput-factor of 100% or more increases, meaning the number of time intervals in which the 10-policy does better than the greedy algorithm increases, as we would expect. When the overload-factor is 4, for example, 66% of the time, the goodput-factor is at least 200%: 10-policy accepts at least twice as many legitimate mails as the greedy algorithm. We conclude that in most time intervals, the history-based policies perform better than the greedy policy, and the factor of their improvement increases as the overload-factor increases.

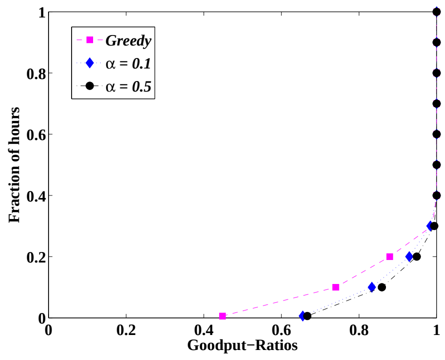
Lastly, we note that the behaviour of the 10-policy and the 50-policy does not appear to differ too much when the overload-factor is sufficiently high or sufficiently low. With intermediate overload-factors, they perform slightly differently, as we see in Table 1: the 50-policy tends to be a little more conservative about accepting messages that may not have a good reputation in comparison to the 10-policy.

Impact on Throughput and Spam

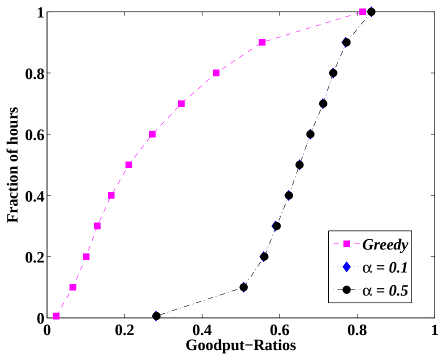
While our primary metric of performance is the goodput, we are still interested in the impact of using the history-based policies on the total messages and spam processed by the mail server. While these are not our primary goals, they are still important since they give a picture of the complete effect of using these history-based policies.

Impact on Server Throughput: The history-based policies obviously gain their improvement by selectively choosing the IP addresses to process: it selectively accepts only good IP addresses in the incoming workload, if it is likely that the whole workload might not be processed. This may result in a decrease in server throughput in comparison to the greedy policy for certain load. For example, if the server receives a little less workload than it could process, the history-based policies may process fewer messages than the greedy policy, because they may reserve capacity for good IP addresses that they expect to see but which never actually appear. We observe this in our simulations and we discuss it now.

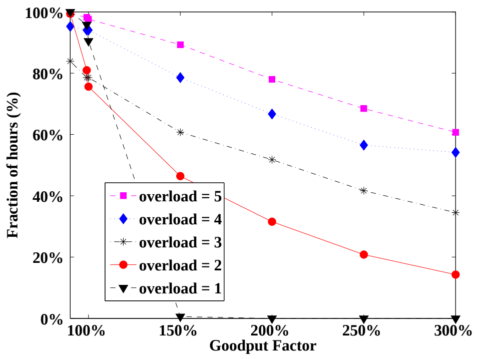
We define *throughput* to be fraction of the total messages processed by the server. Table 2 shows the average throughput achieved by both policies under various capacities of the server. At overload-factor 1, when the greedy algorithm achieves an average throughput of 95%, the history-



(a) Overload-Factor 1: CDF of goodput-ratios for all policies



(b) Overload-Factor 4: CDF of goodput-ratios for all policies



(c) Goodput factor

Figure 5.8: (a) and (b): CDF of the goodput-ratios for two different overload-factors. (c): Performance improvement (goodput-factor) for the 10-policy for various overload factors

Overload Factor	Greedy	α = 0.1	α = 0.5
5	20.3	63	63.6
4	26.8	64.3	64.5
3	39.5	70.7	68.6
2	61.7	84.4	79.6
1	93.7	96	96.7

Table 1: Server Goodput (average, in %).

Overload Factor	Greedy	α = 0.1	α = 0.5
5	31.6	16.6	16.8
4	39.1	17	17
3	51.6	24.1	22.1
2	71.4	65.8	51.3
1	95.2	93.9	95

Table 2: Server throughput (average, in %).

Overload Factor	Greedy	α = 0.1	α = 0.5
5	32	14.8	14.9
4	39.5	15.1	15.1
3	52	20.1	15.2
2	71.7	65.2	50.2
1	95.2	93.8	94.9

Table 3: Spam accepted (average, in %).

based policy algorithm achieves an average throughput of 93%. However, even at this point, the history-based policies accept a little more legitimate mail (on average) than the greedy policy. Note that by design, the history-based policies guarantee that when the server receives no more than 75% of its maximum load capacity, its performance is no different from normal.

Impact on Spam: We also explored the effect of the history-based policies on the number of spam messages accepted. Table 3 shows the average fraction of spam messages accepted by the policies under various overload factors. We see with an overload-factor of 1, the history-based policies accept only $0.3-1\%$ less spam than the greedy algorithm. As the overload-factor increases and the history-based policies grow more and more conservative in accepting suspected spam, the amount of spam accepted will decrease. For example, at a overload-factor of 2, this drops to $50.2\% - 65.5\%$ for the history-based policies. When the overload-factor increases to 4, the history-based policies accept less than 1/2 of the amount of spam accepted by the greedy policy. This suggests that if the server receives much more workload than it can process, the spam is affected much more than the legitimate mail. Therefore, the spammer would not have an incentive to increase the workload significantly, since it is the spam that gets most affected.

Thus, we have shown that our history-based policies achieve a significant and consistent performance improvement over the greedy policy when the server is under overload: we have seen this with multiple metrics of the goodput ratio. We have also seen that the history-based policies do not impact the performance of the server too much when the server is *not* under overload. Finally, we have seen that the the spam is indeed affected when the server is significantly overloaded; this is precisely the behaviour we want to induce.

5.5 Related Work

Since spam is so pervasive, much effort has been expended in developing techniques that mitigate spam, and studies that understand various characteristics of spammers. In this section, we briefly survey some of the most related work. We first describe spam mitigation approaches and how they may relate to our work on the server overload problem. Then we discuss measurement studies that are related and complementary to our measurement work.

Traditionally, the two primary approaches to spam mitigation have used content-based spam-filtering and DNS blacklists. Content-based spam-filtering software [3, 1] is typically applied at the end of the mail processing queue, and there has been a lot of research [102, 86, 8, 83] in techniques for content-based analysis and understanding its limits. Agarwal et al. [6] propose content-based analysis to rate-limit spam at the router; this also reduces the load on the mail server, but is not useful for our situation as it may be too computationally expensive.

DNS blacklists [4, 5] are another popular way to reduce spam. Measurement analyzes on DNS blacklists [66] have shown that over 90% of the spamming IP addresses were present in at least one blacklist at their time of appearance. Our approach is complementary to traditional blacklisting, and the more recent greylisting [62] techniques – we aim to prioritize the legitimate mail, and use the history of IP addresses to identify potential spammers.

Perhaps the closest in spirit to our work in mitigating server overload are those of Twining et al. [112] and Tang et al. [109]. Twining et al. describe a prioritization mechanism that delays spam more than it delays legitimate mail. However, their problem is different, as they eventually accept all email, but just delay the spam. Such an approach would not work when all the mail simply cannot be accepted. While Tang et al. [109] do not consider the problem of server overload, they describe a mechanism to assign trust to and classify IP addresses using SVMs. Our work differs in the way it gets the historical reputations – rather than using a blackbox learning algorithm, it uses the IP addresses and network-aware clusters, thus directly utilizing the structure of the network.

There has also been interest in using reputation mechanisms for identifying spam. There are a few commercial IP-based reputation systems (e.g., SenderBase [2], TrustedSource [111]). A general reputation system for internet defense has been proposed in [22]. There has been work on using social network information for designing reputation-granting mechanisms to mitigate spam [52, 59, 19]. Prakash et al. [98] propose community-based filters trained with classifiers to identify spam. Our work differs from these reputation systems as it demonstrates the potential of using network-aware clusters to assign reputations to IP addresses for prioritizing legitimate mail.

Recently, there have been studies on characterizing spammers, legitimate senders and mail traffic, and we only discuss the most closely related work here. Ramachandran and Feamster [99] present a detailed analysis of the network-level characteristics of spammers. By contrast, our work focuses on the comparison between legitimate mail and spam and explores the stability of legitimate mail. We also use network-aware clusters to probabilistically distinguish the bulk of the legitimate mail from the spam. Gomes et al. [60] study the e-mail arrivals, size distributions and temporal locality that distinguish spam traffic from non-spam traffic; these are interesting features that distinguish spam and legitimate traffic patterns and provide general insights into behaviour. Our measurement study differs as it focuses on understanding the historical behaviour of mail servers at the network level that can be exploited to practical spam mitigation.

5.6 Conclusion

In this chapter, we have focused on using IP addresses as a computationally-efficient tool for spam mitigation in situations when the distinction need not be perfectly accurate. We performed an extensive analysis of IP addresses and network-aware clusters to identify properties that can distinguish the bulk of the legitimate mail and spam. Our analysis of IP addresses indicated that the bulk of the legitimate mail comes from long-lived IP addresses, while the analysis of network-aware clusters indicated that the bulk of the spam comes from clusters that are relatively long-lived. With these insights, we proposed and simulated a history-based reputation mechanism for prioritizing legitimate mail when the mail server is overloaded. Our simulations show that the history and the structure of the IP addresses can be used to substantially reduce the adverse impact of mail server overload on legitimate mail, by up to a factor of 3.

The results presented in this chapter is joint work with Subhabrata Sen, Oliver Spatscheck, Patrick Haffner and Dawn Song, and have previously appeared at Usenix Security 2007 [115].

Chapter 6

Tracking IP Prefixes Originating Malicious Traffic Dynamically

6.1 Introduction

As the Internet traffic grows in volume and complexity, it becomes easier for malicious traffic and malicious entities to remain unidentified. There is a growing trend of attackers using botnets (large armies of compromised hosts) to carry out their attacks, as individual bots are expendable and allow the attacker to preserve his anonymity. Indeed, several studies have shown that tracking individual IP addresses (e.g., spamming bots) has only a limited scope in reducing malicious traffic [66, 99, 115].

There has been recent interest in understanding *regions of the IP space*, rather individual IP addresses, from where malicious traffic originates. This research has indicated that a lot of malicious traffic observed (e.g., spam, scans) is focused on small, specific regions of the IP space. For example, several studies have demonstrated that a significant amount of spam originates from a relatively small number of $/16$ or $/24$ IP prefixes [99, 29]. Our earlier work (chapter 5) also describes additional results in this vein: high-volume spammers are well-correlated with network-aware clusters, and the cluster history be used to distinguish spammers from senders of legitimate mail [115].

This kind of clustered behaviour is unsurprising, as bots originate much of the malicious traffic, and networks that are easily compromised tend to contain many more bots than other networks. Thus, easily compromised networks are likely to originate much more of the malicious traffic than others. However, prior work has focused on finding correlations with a *fixed* set of IP clusters: the analysis typically takes as input a set of IP clusters, and finds clusters among those that originate the most of the malicious traffic. Instead, we are interested in finding the optimal clustering for isolating malicious traffic from legitimate.

In particular, we are interested in the following question: can we partition the IP space into clusters such that can predict IP addresses sending malicious traffic? Such clusters may be useful for many applications e.g. the results in [29] suggest also they could predict future botnet addresses, our earlier work (chapter 4) suggests they may be useful for spam mitigation. They may also be useful for network management, and they may reveal compromised subnets actively employed for malicious purposes. Thus, we want to explicitly find the clusters that have predictive properties: while correlations may be informative, they are often insufficient for detection of attacks and mitigation.

In addition, the regions that originate malicious traffic may change over time for many reasons, attackers may be able to compromise more hosts, some hosts may get patched and no longer send malicious traffic. An attacker may also choose to send malicious traffic from new regions to evade detection. Such evasion has already been observed with individual IP addresses, e.g. spammers query blacklists to identify those IP addresses which are blacklisted, and send spam from the others. For these reasons, we want algorithms that can adapt dynamically when the clusters originating malicious traffic change.

In this chapter, we design online algorithms to track dynamically malicious regions of the IP address space, drawing on ideas from online machine learning, and prove guarantees on its performance. Our performance guarantees ensure that if there is a set of k IP clusters that are good predictors, our algorithm will perform nearly as well as those clusters, on any kind of data. These k IP clusters may even undergo certain kinds of changes over time, and cause only a small impact per change on our algorithm's performance. Our algorithm is also extremely efficient; the space required for the algorithm is $O(klogk)$, and the computation overhead per IP address processed is constant.

We evaluate our algorithm on real data of spam and legitimate mail seen at the enterprise network of a large corporation. We find that our algorithm has excellent empirical results, and is a huge improvement over classifications with network-aware clusters and /24 IP blocks. We also observe that the clusters learnt by the algorithm evolve significantly over the time, illustrating that malicious regions of the IP address may indeed be dynamic.

6.2 Definitions and Preliminaries

In this section, we describe the basic definitions and general framework for the algorithm that we use in the rest of the chapter.

Our high-level goal is to design an algorithm that takes as input IP addresses flagged malicious or non-malicious (e.g., spam logs, labelled with spam-filtering software), and finds a set of IP clusters that predict whether a given IP address is malicious or non-malicious. Often, we may want only a limited number of IP clusters m, to ensure that any application that low overhead on any application that incorporates them.

In machine learning terminology, the IP addresses are the *instances* to be given to the learning algorithm, and their *labels* reflect whether the IP address was flagged to be malicious or non-malicious. We also refer to a non-malicious instance (or IP) as a *positive* instance, and a malicious instance as a *negative* instance. The *classification function* that the algorithm needs to learn is a set of IP clusters and their associated labels. The classification function makes a *mistake* whenever it labels a malicious IP address as non-malicious, or a non-malicious IP address as malicious. The goal of the algorithm is to learn a classification function that minimizes the number of mistakes that are made.

A classification function consisting of IP clusters can also be represented as a binary tree. The IP address space is naturally interpreted as a directed binary tree: the leaves of the tree correspond to individual IP addresses, and the non-leaf nodes correspond to IP prefix ranges. If each leaf node of this binary tree is associated with a label, we get a decision tree over the IP address space. We define *IPTree* to be any pruning of a decision tree over the IP address space. Thus, finding an accurate IPtree is equivalent to finding an accurate set of IP clusters with the labels of individual IP addresses.

Challenges Since our goal is to learn a good classifier for use on network traffic data, we need to address some additional challenges. First, the learning algorithm must have extremely low overhead – the space overhead must be independent of the number of IPs seen, the computation overhead must be constant per IP seen, and the algorithm and must not need to access data it has seen in the past in order to get updated on data that has just arrived. Second, since part of the data may be generated by an adversary, we do not want guarantees that make assumptions about the data. Instead, we require worst-case guarantees on the algorithm's performance – we require guarantees to hold over *every sequence of IP addresses* that may be given to the algorithm. Third, the IP clusters may be dynamic and evolve over time, and we require the algorithm to be able to track dynamically the malicious IP prefix ranges.

We address these challenges by designing algorithms in an *online model of learning* [79]. In this model, the algorithm receives an IP address to be classified, classifies it, and is then given the correct label of the IP address and allowed to update its internal state. The benefits on the overhead in this model are obvious: the algorithm sees only one instance at a time, and does not have to store all the data simultaneously; the algorithm is designed to update with the new data without needing to operate on the earlier data. Second, because the algorithm is allowed to keep updating its classification function, the performance guarantees will not require that assumptions about the data. Lastly, the online model allows the algorithm scope to evolve its classification function as the underlying IP clusters evolve.

The usual kind of guarantees that we get in this framework is a comparison between the mistakes made by the online algorithm and an offline algorithm. A typical point of comparison used in the online learning model is the error of the *optimal offline fixed* algorithm. In this case, the optimal offline fixed algorithm would be the tree of a given size k i.e., the tree of size k that makes the fewest mistakes on the entire sequence. However, if the true underlying IP clusters may change over time, a better point of comparison would allow the offline tree to also change over time. However, to make such a comparison meaningful, the offline tree must pay an additional penalty each time it changes (otherwise the offline tree would not be a meaningful point of comparison – it could change for each IP address in the sequence, and thus make no mistakes).

Instead, we limit the kinds of changes the offline tree can make, and compare the performance of our algorithm to the best IPtree with k leaves that makes only a small number of allowed changes. We define an *adaptive IPtree* with k leaves to be a tree that can (a) contain at most k leaves, (b) grow nodes over time, and (c) change the labels of its leaf nodes. Our goal is perform nearly as well as any adaptive IPtree of size at most k, and in particular, to make a number of mistakes that is not too much larger than the number of mistakes plus the changes of the best such tree.

Problem Statement With this framework and these definitions, we can now present our problem statement: For any given sequence of IP addresses, compute an adaptive IPtree T such that (a) for *every* adaptive tree T' of size k, the mistakes made by T are bounded by a function of the mistakes and the changes made by T', (b) T uses no more than $O(\tilde{k})$ space.

In the next section, we describe an algorithm in which T will need to use $O(k \log k)$ nodes.

6.3 Algorithms and Analysis

In this section, we describe our main algorithm TrackIPTree. We begin with a high-level sketch of the algorithm in Section 6.3.1, and then describe each component of it in detail in Section 6.3.2, and finally summarize the complete algorithm in Section 6.3.3. Then in Section 6.3.4, we give the algorithm's performance guarantees, and in Section 6.3.5, we discuss how it might be adapted to address issues that come up on real data sets.

6.3.1 Overview of *TrackIPTree*

At a high-level, our algorithm operates as follows: when the algorithm receives an IP address, it considers all the nodes on the path of the IP address in the current IPtree. (Recall that the nodes along the path of the IP address are all the prefixes of the IP address in the current IPtree.) The algorithm uses all these prefix nodes together to compute a prediction for the IP address. To do so, it gets the individual predictions of each of the prefix nodes, and combines them togther with the relative importance of the prefixes nodes. Now, in the online model of learning, once the algorithm has made a prediction, it receives the correct label (as described in Section 6.2). It uses this label to "reward" the nodes that predict correctly (e.g., increase their importance), and penalize the nodes that do not.

To flesh out the overview of this algorithm, we still need to address a few issues: (1) determining the relative importance of different nodes; (2) determining whether a node is positive or negative; (3) building an appropriate IPtree that is not too much larger than the optimal tree. We address these issues by formulating each of them as a problem of *combining expert advice* [24] in different situations. Before we fill out the details, we describe briefly the typical problem of combining expert advice, and then adapt different variations of the problem to our setting.

Combining Predictions of Many Experts The typical problem of combining expert advice is the following: we are given several experts who may each make a prediction on a given instance. Our goal is to combine their predictions so that the number of mistakes made is nearly as few as the best *fixed expert* in hindsight. This is a well-studied problem in online learning, with many different variations and many results [79, 80, 81].

The algorithms designed to combine expert advice (*expert algorithms*) have a generic framework. Like other algorithms in the online learning framework, the expert algorithms are also described by a prediction rule and an update rule. Further, each expert carries a weight that describes its importance. The prediction rule combines the weights of different experts' predictions, based on the instance to be classified, while the update rule describes how to penalize incorrect experts and reward the correct experts. In this section, we focus only on describing the prediction and update rules for each of the experts' algorithms that we use; the summary of their guarantees is presented in Section 6.3.4, as part of the analysis of TrackIPTree..

6.3.2 Subproblems of *TrackIPTree*

With this framework, we can now address the three issues mentioned earlier: deciding the relative importance between the different nodes along a path, determining whether a path should predict positive or negative, and building an appropriate tree structure to represent the IPtree.

We disentangle these issues to address them as four separate subproblems – two that manipulate the predictions but do not change the structure of the IPtree, and two that build up the appropriate tree structure in bounded space. In this section, we describe how to address each of these subproblems, and in the next section, we describe how to combine the solutions of all the subproblems into the complete algorithm TrackIPTree.

Before we describe the subproblems of the TrackIPTree algorithm, we introduce some notation. Recall that m is the maximum number of leaves allowed to our algorithm. We use $\epsilon \in (0, 1)$ to denote an online learning rate; it determines how quickly the tree learns from its mistakes a tree with low ϵ will be more robust to noise, but adapt less quickly to changes, and vice versa. Let $P_{i,T}$ denote the set of prefixes of IP address i in the current tree T, i.e., $P_{i,T}$ consists of all nodes in the path of i on T.

```
SLEEPING EXPERTS ALGORITHM
    Initialization:
        Initialize all weights $x_i$ to 1.
    Prediction Rule:
        Denote set of "awake" experts by $A$
        Select expert $j$ with probability $p_j = \frac{x_j}{\sum_{j \in A} x_j}$.
    Update Rule:
        Set costs as follows:
            $cost(j) = 0$ for correct "awake" expert
            $cost(j) = 1$ for incorrect "awake" expert
        For expert $t$:
            $R_t = \frac{[\sum_j p_j \text{cost}(j)]}{(1+\epsilon)} - \text{cost}(t)$
            $x_t = x_t(1+\epsilon)^{R_t}$
```

Figure 6.1: The Sleeping Experts Algorithm

Relative Importance along Path We first consider the problem of computing the relative importance of the different nodes, and we address it with the *sleeping experts* algorithm [51]. The sleeping experts setting is as follows: at every time step, some of the experts are 'awake" and make a prediction on the current instance, and the remaining experts are "asleep", and do not make a prediction on the current instance. The sleeping experts algorithm guarantees that for any expert u, the mistakes made by the algorithm are not much worse than the mistakes made by the expert u on the instances on which u is awake.

In our context, we may consider that the nodes along the path of the IP address in T (i.e., $P_{i,T}$) have the "awake" experts, which we call the *path-node experts*. The leaf of the optimal IPtree has the best path-node expert, and the goal of our algorithm is to predict nearly as well as the best "awake" path-node expert. We use x_n to denote the weight of the path-node expert at node n in T. When TrackIPTree needs to choose a path-node expert for an IP address, it uses the sleeping experts algorithm to make a prediction: it chooses the path-node expert at node n with probability $\frac{x_n}{\sum_{z \in P_{i,T}} x_z}$. Once the algorithm receives the label of the IP address, the path-node experts that predicted incorrectly are penalized, and those that predicted correctly are rewarded, as shown in Fig. 6.1. We integrate this subproblem into the complete algorithm in Sec. 6.3.3.

Deciding Labels of Individual Nodes Next, we need to decide whether the path-node expert at a node n should predict positive or negative. We use a different experts' algorithm to address this subproblem – the *shifting experts* problem. Specifically, we allow each node n to have two additional experts – a positive expert, which always predicts positive, and a negative expert, which always predicts negative, We call these experts *node-label* experts, and now our problem of deciding the label of node n becomes the problem of predicting nearly as well as the best node-label expert at n.

We denote the weights of the positive and negative node-label experts by $y_{n,+}$ and $y_{n,-}$ respectively. The shifting experts algorithm operates as follows: To classify an IP address, the node predicts positive with bias $\frac{y_{n,+}}{y_{n,-}+y_{n,+}}$, and negative with bias $\frac{y_{n,-}}{y_{n,-}+y_{n,+}}$. Every time the node receives a label, it increases the weight of the correct node-label expert by ϵ, and decreases the weight of the incorrect node-label expert by ϵ (up to a maximum of 1, and a minimum of 0).

Thus, the shifting experts algorithm is effectively run on every node on the path of the IP address in T, and the predictions of these nodes are combined with the sleeping experts algorithm described earlier. Note that if the optimal IPtree switches labels, the weights on node-label experts will slowly shift from the incorrect expert to the correct expert, and the weights will reflect the correct expert in $\frac{1}{\epsilon}$ steps. Thus, the shifting experts algorithm applied at each node automatically tracks changing labels on the leaves of the optimal IPtree.

Building Tree Structure We next address the subproblem of building an appropriate structure for the IPtree – our high-level goal here is to grow only those subtrees that are critical to improve the predictions of the IPtree. A natural approach is to target nodes that make a lot of mistakes, and grow their subtrees: when a node in the IPtree makes a lot of mistakes, then either the node has a subtree in the optimal IPtree that separates the positive and negative nodes, or the optimal IPtree must also make the same mistakes. Since TrackIPTree cannot distinguish between these two conditions, it simply splits any node that makes a lot of mistakes, and therefore needs to grow a tree that is somewhat larger than k.

In particular, TrackIPTree operates as follows: it starts with only the root node, and tracks the number of mistakes made at every node. Every time a node makes $\frac{1}{\epsilon}$ mistakes, TrackIPTree splits the node into its children, TrackIPTree waits for $\frac{1}{\epsilon}$ mistakes at each node to add a little resilience with noisy data – otherwise, it would have to split a node every time the optimal tree made a mistake, and the IPtreewould grow very quickly. Note also that if the optimal tree grows nodes over the sequence of IP addresses, this subproblem automatically handles the changes that must occur in the IPtree that our algorithm builds.

Bounding Size of IPtree Since TrackIPTree splits any node that makes a lot of mistakes, it is likely that the IPtree that it builds is split much farther than the optimal IPtree– TrackIPTree does not know when to stop growing a subtree, and it splits even if the same mistakes are made by the optimal IPtree. While this excessive splitting does not impact the predictions of the path-node experts or the node-label experts significantly, we still need to ensure that the IPtree built by our algorithm does not become too large.

We address this issue by recasting it as a paging problem [105]: each node in the IPtree is a page, and the optimal IPtree needs only $2k$ pages. We allow the IPtree built by our algorithm to have m leaves (and thus, $2m$ nodes), and so the size of our *cache* is $2m$ and the optimal cache is $2k$. We can then use a paging algorithm to discard nodes when the IPtree grows beyond $2m$ nodes. In Section 6.3.4, we show that the $m = O(\frac{k}{\epsilon} \log \frac{k}{\epsilon})$ suffices, when the FLUSH-WHEN-FULL (FWF) algorithm is used. In Section 6.3.5, we discuss alternative paging algorithms that may be more useful in a real applications.

6.3.3 The Complete *TrackIPTree* Algorithm

We finally describe the complete TrackIPTree algorithm, putting together the component algorithms that address each of the subproblems. Since it is an online learning algorithm, we summarize its operation by its prediction and update rules as follows:

Prediction Rule: When TrackIPTree needs to classify IP address i, it computes a prediction from the current IPtree T as follows:

- For each node on the path $P_{i,T}$, compute a prediction γ_j using the shifting experts' prediction rule: i.e., flip a coin of bias $\frac{y_{j,+}}{y_{j,+}+y_{j,-}}$ at node j, and return prediction γ_j as positive if the coin turns out heads and negative otherwise. γ_j now becomes the prediction of the path-node expert at node j.
- Using the nodes on the path $P_{i,T}$ as the "awake" nodes, and their respective predictions $\{\gamma_j\}_{j \in P_{i,T}}$, compute the prediction of the entire tree using the sleeping experts' prediction rule: i.e., for each node j on the path $P_{i,T}$, select its path-node expert with probability $\frac{x_j}{\sum\limits_{z \in P_{i,T}} x_z}$. The prediction γ_j of the selected path-node expert is the prediction of the algorithm.

Update Rule: Once TrackIPTree receives the true label, it may update its tree T if necessary. Regardless of whether the algorithm's prediction was correct, the algorithm updates the weights of the path-node experts and the node-label experts of each node on the path $P_{i,T}$, using the sleeping experts' update rule and shifting experts' update rule respectively. In addition, if the algorithm made an incorrect prediction, it does the following:

- Increment the number of mistakes on the leaf of the path $P_{i,T}$ by 1. If the number of mistakes in a leaf node of T exceeds $\frac{1}{\epsilon}$, split the node into its children.
- If the leaf node l on $P_{i,T}$ needs to split:
  - Ensure that the IPtree has fewer than m leaves; if not, discard the appropriate leaves with the chosen paging algorithm.
  - Create the children of l and initialize their weights for the sleeping and shifting experts algorithms.

Thus, we have described the complete TrackIPTree algorithm that puts together the components that address each of the subproblems in the Section 6.3.2. In the next section, we show that the number of mistakes made by TrackIPTree is a small function of the number of changes and the number of mistakes made by any adaptive IPtree of size k.

6.3.4 Analysis

In this section, we present an analysis of our main algorithm TrackIPTree in the online learning model. Our analysis focuses on the case when FWF is used as the paging algorithm for discarding nodes.

Theorem 6.1. *Fix k. Set the maximum number of leaves allowed to the TrackIPTree algorithm m to be $\frac{2k}{\epsilon}(\log k + 2)$. Let T be an adaptive IPtree with at most k leaves. Let $\Delta_{T,s}$ denote the number of times T changed labels on its leaves over s, and $M_{T,s}$ denote the number of mistakes made by T over s.*

The algorithm TrackIPTree ensures that on any sequence of instances s, for all T, the number of mistakes made by TrackIPTree is at most $(1+3\epsilon)M_{T,s} + 2\left(\frac{1}{\epsilon} + 6\right)\Delta_{T,s}$ with probability at least $1 - \left(\frac{1}{k}\right)^{\frac{k}{2\epsilon^2}}$.

Proof:

Our analysis first considers the case when the structure of the tree includes all the nodes in the optimal tree. Here we need to consider the effect of the shifting experts' algorithm for choosing a good node-label expert, of the sleeping experts' algorithm for choosing the a good path-label expert, and of combining them to get a bound on the mistakes made. Let OPT denote any adaptive IPtree of size k; our guarantees hold for all such adaptive IPtrees.

We begin by analyzing the mistakes caused in the shifting experts algorithm that decides the label of an individual node. The optimal node-label expert can look at the entire sequence of instances that arrive at node j, and can decide which label to use to classify instances, and if it chooses, when to switch its label. Let OPT_j be the optimal shifting expert at node j, and Y_j denote the number of mistakes our algorithm makes at node j.

Recall that we shift ϵ from the incorrect node-label expert to the correct node-label expert on every instance that arrives at the node. So the experts at each node n have their weights $y_{n,+} = p$ and $y_{n,-} = (1-p)$ on many different instances in the sequence, for each p. Let $\mathcal{E}_p^+$ denote the event $y_{n,+}$ increases from p to $p+\epsilon$ after seeing an instance and its (positive) label, and let $\mathcal{E}_p^-$ denote the event $y_{n,-}$ decreases from p to $p-\epsilon$ after seeing an instance and its (negative) label. Each instance may cause an event $\mathcal{E}_p^+$ or $\mathcal{E}_p^-$, or the algorithm will predict it correctly. Therefore, to compute the expected error, we simply need to compute (a) the error caused during each event $\mathcal{E}_p^+$ and $\mathcal{E}_p^-$, and (b) the number of such events.

In a pair of events ($\mathcal{E}_p^+$, $\mathcal{E}_{p+\epsilon}^-$), observe that weight $y_{n,+}$ increases from p to $p+\epsilon$, and decreases back from $p + \epsilon$ to p; therefore, we term this pair of events ($\mathcal{E}_p^+$, $\mathcal{E}_{p+\epsilon}^-$) a *cycle*, and we denote the set of all cycles by C. We observe in each cycle, the expected error of our algorithm is $(1+\epsilon)$. We note also that $\sum_p \mathcal{E}_p^+ + \mathcal{E}_p^-$ is at most $|C| + \frac{1}{\epsilon}$. Summing over all events $\mathcal{E}_p^+$ and $\mathcal{E}_p^-$, the expected error is $(1+\epsilon)|C|+\frac{1}{\epsilon}$. Each cycle in C corresponds to either a mistake by OPT_j, or $\frac{1}{\epsilon}$ of 2 changes of OPT_j's node-label experts. Thus, we can get the following bound on the mistakes made by the node-label experts at node j:

$$E[Y_j] = (1+\epsilon)(M_{OPT_j} + \frac{2}{\epsilon}\Delta_{OPT_j} + \frac{1}{\epsilon},$$

$$E[Y_j] = (1+\epsilon)M_{OPT_j} + 2(1+\frac{1}{\epsilon})\Delta_{OPT_j}) + \frac{1}{\epsilon}.$$

Next, we analyze the mistakes caused in choosing the right path-label expert. Consider a node j that is a leaf in the optimal tree. For all the IPs whose prefix is j, the best expert will be node j, and further, node j will not be "awake" for any other IP. By applying the analysis of the sleeping experts' algorithm on the result of the shifting expert algorithm, we get:

$$E[X_j] \leq Y_j(1+\epsilon) + \frac{1}{\epsilon}\log m.$$

Since $m = \frac{8k}{\epsilon^2}\log\frac{k}{\epsilon} < \frac{k^3}{\epsilon^3}$, for $k > 10$, we get:

$$E[X_i] \leq Y_i(1+\epsilon) + \frac{3}{\epsilon}\log\frac{k}{\epsilon}.$$

Since there are k such leaves in OPT, we can apply a similar analysis to sets of the IP addresses classified by each of them. By summing over all the leaves of OPT, we have effectively summed over the entire sequence of IP addresses, and so we can get:

$$E[X] \leq (1+\epsilon)\left(M_{OPT}(1+\epsilon) + 2\left(1+\frac{1}{\epsilon}\right)\Delta_{OPT}\right) + \frac{4k}{\epsilon}\log\frac{k}{\epsilon}.$$

So far, we have analyzed the algorithm when the tree T starts with only the root node, and only grows or changes the labels of its leaves. To complete the proof, we need to analyze the effect of using the paging algorithm to ensure that the size of T is bounded.

Recall that the FWF algorithm discards the entire tree T when the number of leaves in T exceeds m, and restarts with only the root node. Thus, every time T needs to exceed m leaves, TrackIPTree moves back to the case we previously analyzed.

We define an *epoch* to be a sequence of instances such that T starts from the root node and grows to have m leaves. We now bound the number of mistakes made in each epoch. Suppose we set m such that $E[X] \geq \frac{4k}{\epsilon^2}\log\frac{k}{\epsilon}$.

$$
\begin{aligned}
E[X] &\leq (1+\epsilon)\left(M_{OPT}(1+\epsilon) + 2\left(1+\frac{1}{\epsilon}\right)\Delta_{OPT}\right) + \frac{4k}{\epsilon}\log\frac{k}{\epsilon}, \\
or, (1-\epsilon)E[X] &\leq (1+\epsilon)\left(M_{OPT}(1+\epsilon) + 2\left(1+\frac{1}{\epsilon}\right)\Delta_{OPT}\right), \\
or, E[X] &\leq (1+3\epsilon)M_{OPT} + 2\left(6+\frac{1}{\epsilon}\right)\Delta_{OPT}
\end{aligned}
$$

To ensure that the number of mistakes made by the optimal adaptive tree is at least $\frac{4k}{\epsilon^2}\log\frac{k}{\epsilon}$ with high probability, set $m = \frac{8k}{\epsilon^2}\log\frac{k}{\epsilon}$.

■

Thus, we have shown that the mistakes made by TrackIPTree can be bounded as a small function of the mistakes and changes made by any adaptive IPtree of size k. In other words, if there is an adaptive IPtree of size at most k, that makes few changes and few mistakes on the input sequence of IP addresses, then TrackIPTree will also make only a small number of mistakes.

6.3.5 Modifying TrackIPTree for Experimental Issues

In this section, we discuss how the TrackIPTree algorithm can be adapted to resolve a variety of issues that may arise in experiments on data from real applications, but are not considered in the theoretical model used earlier in the chapter.

Optimization Criteria and Cost of Algorithm So far, our algorithms and analysis have assumed that mistakes on positive and negative instances are equally expensive, and therefore, the only quantity to minimize is the number of mistakes. However, in many real applications, one kind of mistake (typically, the false positive) may be much more expensive than the other, or the data may be extremely one-sided. In these situations, simply analyzing the number of mistakes is not necessarily a good performance indicator.

Our algorithm and analysis can be easily adapted to this by assigning different costs for the positive and negative scenarios. For example, if the positive data is 1% the size of the negative data, we can assign the cost of a mistake on a positive instance to be 100 times the cost of a mistake on a negative instance. Our new goal would be to minimize the total cost of the algorithm, rather than the total number of mistakes, and as a consequence, it would ensure that the algorithm learns much more quickly from the more expensive mistake.

Models of Learning and Usage We have described and analyzed our algorithm TrackIPTree in an online learning model, in which it receives IP addresses one at a time, is allowed to classify the

IP address, and then is given the correct label for the IP address. However, we can use the algorithm in other settings as well that occur in real applications.

For example, if we need to quickly classify IP addresses for applications like spam mitigation, we would also use the only prediction rule of TrackIPTree as an inexpensive coarse-grained discriminator. In this situation, TrackIPTree is only guaranteed to predict well on instances similar to the ones that were used to build the IPtree. If there is a change in the data distributions, the algorithm would not be able to adapt until it is given the new labels reflecting those changes.

Further, it is clear we could also use TrackIPTree to run on data collected offline as well – we can feed the algorithm the IP addresses and their corresponding labels one at a time. The performance benefits of using TrackIPTree in the offline setting is again the reduced computational and space overhead; in the experimental section, we demonstrate how TrackIPTree compares to the optimal offline algorithm in terms of accuracy and overhead.

Paging Algorithm Our analysis assumes that when the IPtree has grown to its maximum allowed size, the FLUSH-WHEN-FULL paging algorithm is used to discard nodes. FWF discards all pages in the cache when the cache is full, and simply starts again with an empty cache. In our context, this would correspond to discarding all nodes in the tree T, and starting again with only a newly initialized root node.

In a real implementation, discarding the entire tree is likely a waste of information, and so we instead use a paging algorithm that makes a a less drastic change to the tree. In particular, we use LRU in our implementation of TrackIPTree to select nodes that may be removed from the tree, and for efficiency, we discard a small fraction (e.g, 1%) of the nodes in T (rather than just discarding a single node at a time).

6.4 Evaluation

In this section, we present the results of our experiments in using TrackIPTree to classify IP addresses as malicious or non-malicious, based on whether they send significant quantities of spam or legitimate mail.

6.4.1 Data and Preliminaries

Our data set consists of e-mail logs that are classified into spam and legitimate mail using SpamAssassin, collected over 6 months from January 2006 to June 2006. The logs contain 28 million messages, of which around 1.2 million are legitimate mail and the rest are spam. These are the same logs used in the measurement analysis in Chapter 5.

For all the experimental results that follow, we use LRU as the paging algorithm, when nodes need to be discarded from the tree, and it discards 1% of the nodes in the tree (i.e., 1% of m, the maximum number of leaves allowed in the tree) every time. Our results are similar when LFU is used as the paging algorithm, and the number of nodes discarded is varied between $1-5\%$ of m. Our learning rate ϵ is set to 0.05 and the maximum number of leaves allowed m is set to $10,000$.

Because our data contains mostly spam, evaluating the results in terms of the number of mistakes made is no longer meaningful – it would be trivial for any algorithm to achieve a very small number of mistakes by classifying all the spam correctly, and very little of the legitimate mail correctly. However, an algorithm that classifies a significant fraction of the legitimate mail correctly is likely preferable, even if it makes more mistakes overall. Therefore, we evaluate the performance of the different algorithms by plotting the trade-off on their accuracy on spam and legitimate mail, i.e., we plot the algorithm's accuracy on spam subject to the constraint that it classifies at least $x\%$ of the legitimate mail correctly.

Our experiments study three sets of results. First, we compare the accuracy tradeoff of Track-IPTree with optimized /24 prefixes and network-aware clusters. Then, we examine the accuracy tradeoff as k varies. Finally, we examine the distribution of IP prefix clusters generated over different points of time, which gives us a snapshot of how the IPtree is evolving.

6.4.2 A Baseline Algorithm

In order to use any pre-defined clusters to classify IP addresses, they need to be assigned labels. Because our data is one-sided, we assign labels subject to the constraint that at least a w-fraction of the positive labels are correct. We assign these labels using a basic greedy algorithm with two passes over the data. The greedy algorithm will optimize the labels of the clusters for the data, and minimize the number of mistakes that need to be made on the negative instances, subject to the constraint that at least a w-fraction of the positive labels are correct.

Specifically, we do the following: In the first pass, we store the number of positive and negative labels at each cluster. At the end of the pass, we compute the ratio of positive to negative labels in each leaf cluster, that we term the *positive-ratio* of the leaf cluster. We sort the leaves by their positive-ratio, and compute m, the number of leaves that contain w-fraction of the positive labels. We assign positive labels to the top m clusters and negative labels to the rest. In the second pass, we use these labels to classify the IP addresses.

Next, we examine /24 IP blocks and network-aware clusters perform when assigned labels with this greedy algorithm, compared to TrackIPTree.

6.4.3 Comparisons with Baseline Algorithm

Our first set of experiments compares the accuracy of our algorithm with network-aware clusters and /24 IP prefixes. We assign labels to these two sets of clusters using 2-pass greedy algorithm in

Section 6.4.2. Note that this experimental set-up is favourable to the baseline clusters – since the greedy algorithm gets two passes over the data, the clusters are optimized for the data that they will be classifying.

Fig. 6.4.3(a) plots the accuracy curve of the three sets of clusters. It is clear that the accuracy of TrackIPTree is significantly better than using either of the pre-defined clusters – for any choice of accuracy of legitimate mail, the mistakes made by TrackIPTree is far lower than the mistakes made by the network-aware clusters For instance, when the algorithms are constrained to classify at least 50% of the legitimate mail correctly, TrackIPTree classifies nearly 99% of the spam correctly as well, while the pre-defined clusters classify only $56 - 77\%$ of the spam correctly.

Fig. 6.4.3(b) shows the number of leaves instantiated for each of the three IPtrees. For the pre-defined clusters, we only instantiate leaves that are needed for the classification of our data. Note that the trees for the baseline clusters are significantly *less* space-efficient than TrackIPTree – they require at least a factor of 100 more leaves than TrackIPTree. Thus, TrackIPTree produces a smaller set of clusters that have much better predictive power.

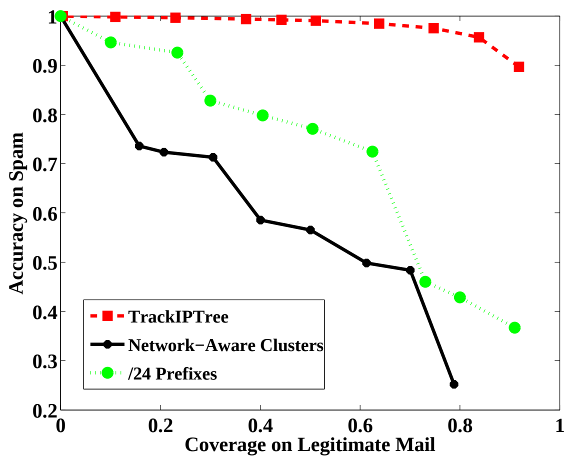
6.4.4 Effect of Changing m

Next, we explore the effect of changing m, the maximum number of leaves allowed to the TrackIPTree. Larger values of m typically imply higher accuracy for the algorithm. However, once m is large enough to capture the most of the distinct subtrees in the underlying optimal IPtree, any further increase will not yield a significant improvement.

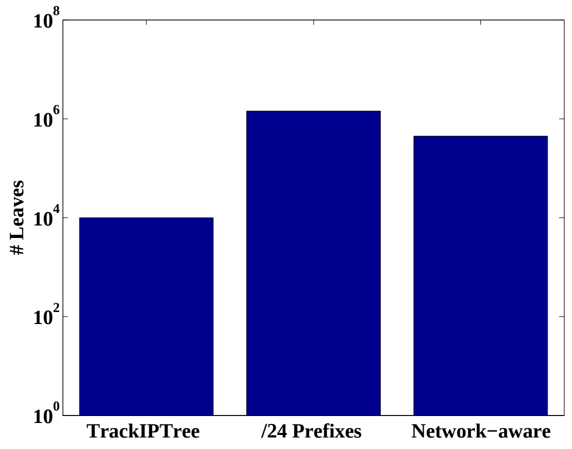
Fig. 6.4.4(a) shows the accuracy curves for the TrackIPTreewhen m ranges from 1,000 to 50,000 leaves. It is clear that when the TrackIPTree is allowed a maximum of only a 1000 leaves, there is a noticeable drop in the classification accuracy. For instance, when $m = 10,000$, the TrackIPTree is able to simultaneously classify 77.2% of the legitimate mail and 97% of the spam correctly. However, for $m = 1000$, the accuracy on the legitimate mail drops to 57% when the algorithm must classify 97% of the spam correctly.

On the other hand, the results of TrackIPTree with 10,000 leaves or with 50,000 leaves are quite similar. When m is increased from 10,000 leaves to 50,000 leaves, the algorithm's accuracy on the legitimate mail increases only marginally, from 77.2% to 77.8%. Thus, for this data set, little improvement seems to be gained by any increase in m beyond 10,000.

While this value of $m = 10,000$ is specific to this data set, the experiment illustrates the tradeoff that appears with choosing different values of m – too large a m causes a performance overhead without an improvement in accuracy (and may also result overfitting), while too small a m could cause a significant loss in accuracy. By choosing an appropriate m for the application and the environment, we can provide an good balance between accuracy and overhead.

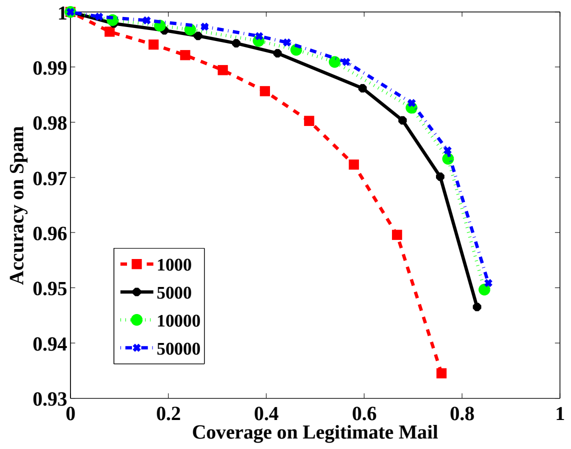


(a) Accuracy Trade-off

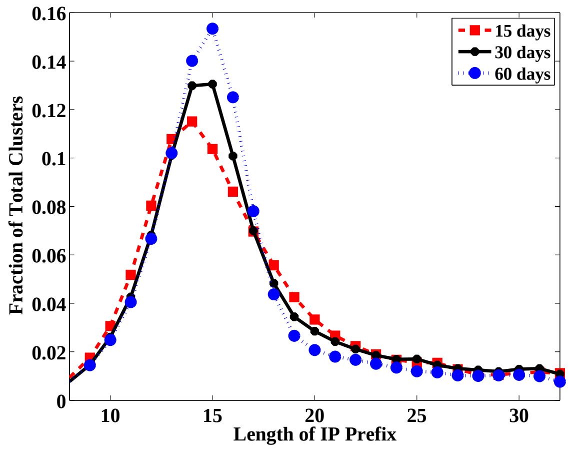


(b) Leaves generated

Figure 6.2: Comparisons with Baseline Algorithms



(a) Accuracy Tradeoff with Changing m



(b) Distribution of Prefix Sizes of Clusters

Figure 6.3: Properties of clusters generated by TrackIPTree

6.4.5 Cluster Distribution over Time

Finally, we examine the prefix sizes of the nodes in the tree generated by TrackIPTree, and how the distribution of the prefix sizes changes over time. Fig. 6.4.4(b) plots the number of clusters (normalized by $2m$) with different prefix sizes after 15, 30, and 60 days of running the algorithm continuously.

Fig. 6.4.4(b) is one way to observe how the tree evolves over time. The fraction of clusters that with prefix lengths between 15-17 increases by as much as a third, between the shortest and longest time periods. On the other hand, the fraction of clusters with prefix lengths between 24-32 drops by as much as a third between the same two periods. Clearly, the tree has lost a number of nodes with prefix lengths between 24-32, and gained a number of them with prefix-lengths 15-17.

The experiment suggests that the IP prefixes from which the spam and legitimate data originate may change over time. While the early IPtree does not need to branch on many of the /15-/16 IP blocks, the later IPtree does branch on them, while dropping many of the smaller IP blocks. This would happen only if the IPtree made sufficient mistakes between the 15th and the 60th days on all of those nodes.

6.5 Related Work

In this section, we review related work that has not been previously mentioned earlier in the chapter.

The primary motivation for our work came from a series of measurement results and analysis, all of which suggested that prefix-based IP address blocks may be useful for predicting malicious and non-malicious network traffic. We have described this work in Section 6.1, and so we do not go into detail again here.

There has been much interest in networking and database literature in efficient algorithms for finding IP clusters; among these are algorithms for the heavy-hitter [46, 47, 45], hierarchical heavy-hitter [129], multidimensional heavy-hitter problems [32]. The goal of these problems is to analyze traffic patterns and detect new IP prefix ranges with significant activity. To our knowledge, none of the work here has focused on the prediction problem – separating the regions of the IP address space that send malicious and benign traffic.

There has also been work on related problems in the machine learning literature and algorithms literature. Perhaps most closely related in the online learning literature is the work of Helmbold and Schapire [63] that presents an algorithm to predict nearly as well as the best pruning of a decision tree. Our problem has a few constraints that they do not: first, our algorithm needs to have low space and computational overhead in order to scale to data seen even in enterprise networks today. In addition, the underlying tree may change over time, and our algorithm needs to be able to adapt to dynamic evolving data.

A second line of closely related work considers the problem of finding decision trees over streaming data [64, 42]. Our problem is a little different as we do not actually need to build a decision tree from scratch – at every node, we already know the how to branch further; the structure of the tree is fixed. On the other hand, we also have two additional sources of complexity in comparison to their setting: (1) our data may be generated by an adversary, and thus our bounds need to hold in adversarial environments, and (2) the underlying tree may change over time and algorithm must adapt to the changes.

6.6 Conclusion

In this chapter, we design online algorithms to track dynamically malicious regions of the IP address space, drawing on ideas from online machine learning. We prove guarantees on its performance, and these guarantees ensure that as long as there is *one* set of k IP clusters that are good predictors, our algorithm will perform nearly as well as those clusters, on any sequence data, when allowed $O(k \log k)$ space. These k IP clusters may even undergo certain kinds of changes over time, and cause only a small impact per change on our algorithm's performance. We evaluated our algorithm on real data of spam and legitimate mail seen at the enterprise network of a large corporation. We found that our algorithm has excellent empirical results, and its predictions are orders of magnitude better than the predictions of two fixed clustering schemes – network-aware clusters and /24 IP prefix blocks.

Chapter 7

Conclusion

A recurring problem in network traffic analysis is to automatically distinguish legitimate traffic from malicious or spurious traffic. This problem arises in several guises, e.g., for spam mitigation at the network-level, we need to distinguish between the connections that originate from legitimate mail servers and spammers; for worm detection, we need to distinguish between packets that contain exploits and those that do not. Many of these problems are, at core, machine learning or data mining problems – we need to either learn a classifier that identifies legitimate traffic patterns, or to find a known traffic pattern of interest efficiently. However, it is often infeasible to directly apply classical techniques from machine learning and data mining to these problems, because of the presence of intelligent adversaries and the scale of traffic on the Internet.

In this thesis, we examined four different instances of this problem in network traffic analysis, and we used tools from computational learning theory and streaming algorithms to address them. These four problems are characterized by different kinds of structure in network traffic, have different kinds of adversaries, and different kinds of scaling requirements. In each of these problems, the approach we take is to formally specify the structure of the problem and the adversary, and use this structure to reason about the kind of performance guarantees we can get as a function of the adversary's power.

Chapter 2 examined the stepping-stones problem. Here, we used the temporal structure of the traffic – in particular, the inter-packet timing delays – to identify pairs of streams that are likely to be stepping-stones. We provided algorithms with strong upper bounds on the number of packets they need to observe, to detect attacks with given false positive and false negative rates. We also presented lower bounds showing how an adversary, with sufficient chaff, could evade any detection mechanism that is based only on the timing delays between packets.

Chapter 3 explored the hardness of automatic signature generation using *pattern-extraction algorithms*, which are algorithms that use the content structure of network traffic to distinguish worm traffic from normal traffic. A sequence of prior work has alternately developed a variety of systems, each demonstrating good experimental performance on the problem, and a variety of attacks, demonstrating how each system could be evaded. We presented lower bounds showing how *any* pattern-extraction algorithm could be misled, in the presence of an adversary with sufficient control over the malicious data.

Chapter 4 presented efficient streaming algorithms to identify *superspreaders*, which are sources that contact many distinct destinations in a short time period. The communication structure of most hosts on the Internet makes finding superspreaders of interest to security applications, as they are likely indicators of worms, scanning, or other malicious activity. Our experimental results on real network traces showed that our algorithms are substantially more efficient than earlier approaches.

Chapters 5 and 6 focused on network-level spam mitigation. In Chapter 5, we performed a systematic characterization of the discriminatory power of IP addresses for this problem, and our analysis showed that the network-level characteristics of spammers differ significantly from those of legitimate mail senders. In Chapter 6, we examined the problem of leveraging this structure as effectively as possible by tracking malicious regions of the IP address space. We developed online algorithms to leverage this network structure with provable optimality guarantees. Our experimental results demonstrated that our algorithm finds IP prefixes with predictive power that is orders of magnitude more accurate than commonly-used IP prefixes.

The results in this thesis illustrate how the unusual challenges in traffic analysis for network security may be tackled by using ideas – rather than classical blackbox techniques – from machine learning and data mining. Chapters 2 & 3 illustrate how mathematical guarantees can add significantly to experimental analysis in an adversarial setting. For example, our results on learning-based signature showed that by abstracting away the specific algorithms and attacks, we could analyze when the approach may or may not work, and how long an adversary could fool the approach. Chapters 4, 5 & 6 illustrate how problem-specific structure may be used to design efficient algorithms. For example, in our work on detecting superspreaders, we used insights on traffic distributions to design a novel (and substantially more efficient) algorithm than a straightforward combination of sampling primitives. Likewise, our data analysis of spam empirically revealed a key problem-specific structure inherent in spamming IP addresses, which we then used to enhance spam mitigation at the network-level.

Finally, while this thesis has illustrated how ideas from machine learning provide great leverage to address the challenges of traffic analysis, it has still focused primarily on drawing ideas from computational learning theory and streaming algorithm design. As applications grow more complex and sophisticated, it is likely that the challenges will need to draw on ideas from many other specialized sub-fields of machine learning and data mining leading to closer connections between systems security and machine learning.

Bibliography

[1] Brightmail. http://www.brightmail.com.

[2] SenderBase. http://www.senderbase.org.

[3] SpamAssassin. http://www.spamassassin.org.

[4] SpamCop. http://www.spamcop.net.

[5] SpamHaus. http://www.spamhaus.net.

[6] Banit Agarwal, Nitin Kumar, and M. Molle. Controlling spam e-mails at the routers. In *IEEE International Conference on Communications (ICC)*, 2005.

[7] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. of Computer and System Sciences*, 58:137–147, 1999.

[8] I. Androutsopoulos, J. Koutsias, K. Chandrinos, G. Paliouras, and C. Spyropoulos. Spam filtering with Naive Bayes - which Naive Bayes? In *Third Conference on Email and Anti-Spam*, 2006.

[9] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[10] Peter Auer. Learning nested differences in the presence of malicious noise. *Theoretical Computer Science*, 185(1):159–175, 1997.

[11] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issue in data stream systems. In *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems (PODS)*, pages 1–16, June 2002.

[12] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proc. 6th International Workshop on Randomization and Approximation Techniques (RANDOM)*, pages 1–10, September 2002. Lecture Notes in Computer Science, vol. 2483, Springer.

[13] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2002.

[14] D. Barbara, J. Couto, S. Jajodia, L. Popyack, and N. Wu. Adam: detecting intrusions by data mining. In *IEEE Workshop on Information Assurance and Security 2001*, 2001.

[15] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D. Joseph, and J. D. Tygar. Can machine learning be secure? In *ASIA CCS*, March 2006.

[16] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[17] Avrim Blum, Dawn Song, and Shobha Venkataraman. Detection of interactive stepping stones: Algorithms and confidence bounds. In *Proceedings of the* 7*th* *International Symposium on Recent Advances in Intrusion Detection, RAID '04*, September 2004.

[18] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Information Processing Letters*, 24:377–380, April 1987.

[19] P. Oscar Boykin and Vwani P. Roychowdhury. Leveraging social networks to fight spam. *Computer*, 38(4):61–68, 2005.

[20] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Allerton*, 2002.

[21] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 2–16, 2006.

[22] David Brumley and Dawn Song. Towards attack-agnostic defenses. In *Proceedings of the First Workshop on Hot Topics in Security (HOTSEC)*, 2006.

[23] N. H. Bshouty, N. Eiron, and E. Kushilevitz. PAC learning with nasty noise. In *Algorithmic Learning Theory, ALT'99*, 1999.

[24] Nicolò Cesa-Bianchi, Yoav Freund, David Haussler, David P. Helmbold, Robert E. Schapire, and Manfred K. Warmuth. How to use expert advice. *J. ACM*, 44(3):427–485, 1997.

[25] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proc. 19th ACM Symp. on Principles of Database Systems*, pages 268–279, May 2000.

[26] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. 29th Intl. Colloq. on Automata, Languages, and Programming*, 2002.

[27] Simon P. Chung and Alyosius K. Mok. Advanced allergy attacks: Does a corpus really help? In *Proceedings of Recent Advances in Intrusion Detection, 10th International Symposium*, 2007.

[28] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. of Computer and System Sciences*, 55(3):441–453, 1997.

[29] M. Patrick Collins, Timothy J. Shimeall, Sidney Faber, Jeff Naies, Rhiannon Weaver, and Markus De Shon. Using uncleanliness to predict future botnet addresses. In *Proceedings of the Internet Measurement Conference*, 2007.

[30] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using hamming norms (how to zero in). In *Proc. 28th International Conf. on Very Large Data Bases (VLDB)*, pages 335–345, August 2002.

[31] G. Cormode and S. Muthukrishnan. What's new: Finding significant differences in network data streams. In *Proceedings of IEEE Infocom*, 2004.

[32] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Diamond in the rough: finding hierarchical heavy hitters in multi-dimensional data. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 155–166, New York, NY, USA, 2004. ACM.

[33] Manuel Cost, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In 20^{th} *ACM Symposium on Operating System Principles (SOSP 2005)*, 2005.

[34] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP '07)*, 2007.

[35] Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Can we contain internet worms? In *HotNets 2004*, 2004.

[36] Jedidiah Crandall, Zhendong Su, S. Felix Wu, and Frederic Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proc. 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[37] Jedidiah R. Crandall and Fred Chong. Minos: Architectural support for software security through control data integrity. In *To appear in International Symposium on Microarchitecture*, December 2004.

[38] W. Cui, M. Peinado, H. Wang, and M. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.

[39] Nilesh Dalvi, Pedro Domingos, Mausam, Sumit Sanghai, and Deepak Verma. Adversarial classification. In *Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2004.

[40] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.

[41] E. Demaine, A. Lopez-Ortiz, and J. Ian-Munro. Frequency estimation of internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium on Algorithms*, pages 348–360, September 2002.

[42] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of ACM SIGKDD*, pages 71–80, 2000.

[43] D. Donoho, A. G. Flesia, U. Shankar, V. Paxson, J. Coit, and S. Staniford. Multiscale stepping-stone detection: Detecting pairs of jittered interactive streams by exploiting maximum tolerable delay. In *Fifth International Symposium on Recent Advances in Intrusion Detection, Lecture Notes in Computer Science 2516*, New York, 2002. Springer.

[44] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *Proceedings of ACM SIGCOMM*, 2003.

[45] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proceedings of SIGCOMM'03*, 2003.

[46] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of SIGCOMM'02*, 2002.

[47] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *ACM SIGCOMM Internet Measurement Workshop*, 2003.

[48] W. Feller. *Probability Theory and its Applications*, volume 1. John Wiley and Sons, Inc., 1968.

[49] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Computer and System Sciences*, 31:182–209, 1985.

[50] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic blending attacks. In *Proceedings of the 15th Usenix Security Symposium (Security '06)*, 2006.

[51] Yoav Freund, Robert E. Schapire, Yoram Singer, and Manfred K. Warmuth. Using and combining predictors that specialize. In *Proceedings of the Twenty-Ninth Annual Symposium on the Theory of Computing (STOC)*, pages 334–343, 1997.

[52] Scott Gariss, Michael Kamisky, Michael Freedman, Brad Karp, David Mazieres, and Haifeng Yu. Re: Reliable email. In *Proceedings of NSDI*, 2006.

[53] A.K. Ghosh and A. Schwartzband. A study in using neural networks for anomaly and misuse detection. In *Proc. of USENIX Security Symposium*, 1999.

[54] G. Giacinto and F. Roli. Intrusion detection in computer networks by multiple classifier systems. In *International Conference on Pattern Recognition 2002*, 2002.

[55] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 331–342, June 1998.

[56] P. B. Gibbons and Srikanta Tirthapura. Estimating simple functions on the union of data streams. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 281–291, June 2001.

[57] P. B. Gibbons and Srikanta Tirthapura. Distributed streams algorithms for sliding windows. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 63–72, August 2002.

[58] L. Golab, D. DeHaan, E. Demaine, A. Lopez-Ortiz, and J. Ian-Munro. Identifying frequent items in sliding windows over online packet streams. In *Proceedings of 2003 ACM SIGCOMM conference on Internet measurement*, pages 173–178. ACM Press, 2003.

[59] Jennifer Golbeck and James Hendler. Reputation network analysis for e-mail filtering. In *First Conference on E-mail and Antispam*, 2004.

[60] Luiz Henrique Gomes, Cristiano Cazita, Jussara M. Almeida, Virglio Almeida, and Jr. Wagner Meira. Characterizing a spam traffic. In *Proceedings of Internet Measurement Conference (IMC)*, 2004.

[61] Matthew Van Gundy, Davide Balzarotti, and Giovanni Vigna. Catch me, if you can: Evading network signatures with web-based polymorphic attacks. In *In Proceedings of WOOT'07*, 2007.

[62] E. Harris. The next step in the spam control war: Greylisting. http://projects.puremagic.com/greylisting/.

[63] David P. Helmbold and Robert E. Schapire. Predicting nearly as well as the best pruning of a decision tree. *Machine Learning*, 27(1):51–68, 1997.

[64] Ruoming Jin and Gagan Agarwal. Efficient and effective decision tree construction on streaming data. In *Proceedings of ACM SIGKDD*, 2003.

[65] J. Jung, V. Paxson, A.W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.

[66] Jaeyeon Jung and Emil Sit. An empirical study of spam traffic and the use of DNS black lists. In *Proceedings of Internet Measurement Conference (IMC)*, 2004.

[67] R. Karp, S. Shenker, and C. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, 2003.

[68] M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[69] Michael Kearns and Ming Li. Learning in the presence of malicious errors. *SIAM Journal on Computing, 22(4):807-837*, 1993.

[70] Hyang-Ah Kim and Brad Karp. Autograph: toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[71] Christian Kreibich and Jon Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[72] Balachander Krishnamurthy and Jia Wang. On network-aware clustering of web clients. In *Proceedings of ACM SIGCOMM*, 2000.

[73] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymophic worm detection using structural information of executables. In *Proceedings of the Recent Advances in Intrusion Detection (RAID)*, 2005.

[74] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. In *ACM SIGCOMM 2005*, 2005.

[75] W. Lee and S. Stolfo. A framework for constructing features and models for intrusion detection. *ACM Transactions on Information and System Security*, 3(4), November 2000.

[76] L.Ertz, E. Eilertson, A. Lazarevic, P. Tan, J. Srivastava, V. Kumar, and P. Dokas. *The MINDS - Minnesota Intrusion Detection System*. MIT Press, 2004.

[77] Zhichun Li, Manan Shanghi, Brian Chavez, Yan Chen, and Ming-Yang Kao. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resiliance. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[78] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proc. of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[79] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear threshold algorithm. *Machine Learning*, 2(285-318), 1988.

[80] N. Littlestone. Redundant noisy attributes, attribute errors, and linear-threshold learning using winnow. In *Fourth Annual Workshop on Computational Learning Theory*, pages 147–156, 1991.

[81] N. Littlestone and M. Warmuth. The weighted majority algorithm. *Information and Computation*, 108:212–251, 1994.

[82] Daniel Lowd and Christopher Meek. Adversarial learning. In *Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2005.

[83] Daniel Lowd and Christopher Meek. Good word attacks on statistical spam filters. In *Second Conference on Email and Anti-Spam*, 2005.

[84] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of VLDB 2002*, 2002.

[85] Z. Morley Mao, Vyas Sekar, Oliver Spatscheck, Jacobus van der Merwe, and Rangarajan Vasudevan. Analyzing large ddos attacks using multiple data sources. In *ACM SIGCOMM Workshop on Large Scale Attack Defense*, 2006.

[86] Ben Medlock. An adaptive, semi-structured language model approach to spam filtering on a new corpus. In *Third Conference on Email and Anti-Spam*, 2006.

[87] A. W. Moore and D. Zuev. Internet traffic classification using bayesian analysis techniques. In *ACM SIGMETRICS 2005*, 2005.

[88] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. Security and Privacy Magazine, July/August 2003.

[89] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, UK, 1995.

[90] S. Muthukrishnan. Data streams: Algorithms and applications. Technical report, Rutgers University, Piscataway, NJ, 2003.

[91] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[92] James Newsome, Brad Karp, and Dawn Song. Paragraph: Thwarting signature learning by training maliciously. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2006.

[93] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, February 2005.

[94] NLANR. National laboratory for applied network research. http://pma.nlanr.net/Traces/, 2003.

[95] Good Word Attacks on Statistical Spam Filters. D. lowd and c. meek. In *Proceedings of Second Conference on E-mail and Anti-Spam*, 2005.

[96] Vern Paxson and Sally Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[97] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul Sharif. Misleading worm signature generators using deliberate noise injection. In *IEEE Symposium on Security and Privacy*, may 2006.

[98] Vipul Ved Prakash and Adam O'Donnell. Fighting spam with reputation systems. *Queue*, 3(9):36–41, 2005.

[99] Anirudh Ramachandran and Nick Feamster. Understanding the network-level behavior of spammers. In *Proceedings of ACM SIGCOMM*, 2006.

[100] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference*. USENIX, 1999.

[101] M. Sabhnani and G. Serpen. Application of machine learning algorithms to kdd intrusion detection dataset within misuse detection context. In *International Conference on Machine Learning, Models, Technologies and Applications 2003*, 2003.

[102] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A Bayesian approach to filtering junk E-mail. In *Learning for Text Categorization: Papers from the 1998 Workshop*. AAAI Technical Report WS-98-05, 1998.

[103] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. The EarlyBird system for real-time detection of unknown worms. Technical Report CS2003-0761, University of California, San Diego, August 2003.

[104] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, December 2004.

[105] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

[106] S. Staniford-Chen and L. T. Heberlein. Holding intruders accountable on the internet. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 39–49, Oakland, CA, 1995.

[107] C. Stoll. *The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage*. Pocket Books, October 2000.

[108] G. Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of ASPLOS*, 2004.

[109] Yuchun Tang, Sven Krasser, and Paul Judge. Fast and effective spam sender detection with granular SVM on highly imbalanced server behavior data. In *2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2006.

[110] M. Thottan and C. Ji. Anomaly detection in ip networks. *IEEE Transactions on Signal Processing*, 51(8), 2003.

[111] TrustedSource. http://www.trustedsource.org.

[112] Dan Twining, Matthew M. Williamson, Miranda Mowbray, and Maher Rahmouni. Email prioritization: Reducing delays on legitimate mail caused by junk mail. In *USENIX Annual Technical Conference*, 2004.

[113] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.

[114] Shobha Venkataraman, Avrim Blum, and Dawn Song. Limits of learning-based signature generation with adversaries. In *Proceedings of the* 15^{th} *Annual Network and Distributed System Security Symposium, NDSS '08*, February 2008.

[115] Shobha Venkataraman, Subhabrata Sen, Oliver Spatscheck, Patrick Haffner, and Dawn Song. Exploiting network structure for proactive spam mitigation. In *Proceedings of Usenix Security'07*, 2007.

[116] Shobha Venkataraman, Dawn Song, Phillip B. Gibbons, and Avrim Blum. New streaming algorithms for fast detection of superspreaders. In *Proceedings of the* 12^{th} *Annual Network and Distributed System Security Symposium, NDSS '05*, February 2005.

[117] Helen J Wang, Chuanxiong Guo, Daniel Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proc. of the 2004 ACM SIGCOMM Conference*, August 2004.

[118] Ke Wang, Janak J. Parekh, and Salvatore J. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[119] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based worm detection and signature generation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[120] X. Wang. The loop fallacy and serialization in tracing intrusion connections through stepping stones. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 404–411, Nicosia, Cyprus, 2004. ACM Press.

[121] X. Wang and D.S. Reeves. Robust correlation of encrypted attack traffic through stepping stones by manipulation of inter-packet delays. In *Proceedings of the 2003 ACM Conference on Computer and Communications Security (CCS 2003)*, pages 20–29. ACM Press, October 2003.

[122] X. Wang, D.S. Reeves, and S.F. Wu. Inter-packet delay-based correlation for tracing encrypted connections through stepping stones. In D.Gollmann, G.Karjoth, and M.Waidner, editors, *7th European Symposium on Research in Computer Security (ESORICS 2002), Lecture Notes in Computer Science 2502*, pages 244–263. Springer, October 2002.

[123] X. Wang, D.S. Reeves, S.F. Wu, and J. Yuill. Sleepy watermark tracing: An active network-based intrusion response framework. In *Proceedings of the 16th International Information Security Conference (IFIP/Sec'01)*, pages 369–384, June 2001.

[124] N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *Proceedings of the 13th Usenix Security Conference*. USENIX, August 2004.

[125] Yinglian Xie, Fang Yu, Kannan Achan, Elliot Gillum, , Moises Goldszmidt, and Ted Wobber. How dynamic are ip addresses? In *Proceedings of ACM SIGCOMM*, 2007.

[126] K. Xu, Z. Zhang, and S. Bhattacharyya. Profiling internet backbone traffic: behavior models and applications. In *ACM SIGCOMM 2005*, 2005.

[127] K. Yoda and H. Etoh. Finding a connection chain for tracing intruders. In *F. Guppens, Y. Deswarte, D. Gollmann and M. Waidner, editors, 6th European Symposium on Research in Computer Security – ESORICS 2000 LNCS-1895*, Toulouse, France, October 2000.

[128] Yin Zhang and Vern Paxson. Detecting stepping stones. In *Proceedings of the 9th USENIX Security Symposium*, pages 171–184, August 2000.

[129] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, and Carsten Lund. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 101–114, New York, NY, USA, 2004. ACM.

[130] Denis Zuev and Andrew Moore. Traffic Classification using a Statistical Approach. In *Proceedings of the Passive & Active Measurement Workshop (PAM2005)*, March/Apri 2005.