A Protocol Graph Based Anomaly Detection System

Michael Collins

April 28, 2008

School of Electrical and Computer Engineering

Carnegie Mellon University

Pittsburgh, PA 15213

Thesis Committee:

Michael Reiter (chair)

John McHugh

Srinivasan Seshan

Hui Zhang

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy.

Contents

1 Introduction **19**

1.1 Anomaly Detection Using Protocol Graphs 23

1.2 Training IDS on Noisy Data 25

1.3 Evaluating IDS Impact 26

1.4 Outline of the Work 28

2 Protocol Graphs **31**

2.1 Previous Work 35

2.2 Preliminaries 37

2.2.1 Protocol Graphs 37

2.2.2 Data Set 39

2.3 Building a Hit-List Worm Detector 41

2.3.1 Graph Behavior Over Time 43

2.3.2 Detection and the False Alarm Rate 45

2.4 Protocol Graph Change During Attack 47

2.4.1 Attack and Defense Model 47

2.4.2 Experiment Construction 49

2.4.3 True Alarms 52
2.5 Bot Identification 57
2.6 Implementation 64
2.7 Conclusion 67

3 Training Anomaly Detectors 69
3.1 Related Work 74
3.2 System Architecture and Log-level Filtering 77
3.2.1 Source Data 77
3.2.2 System Architecture 80
3.2.3 Log-level Filtering 83
3.3 Implementing a State-level Filter 85
3.3.1 State Data 87
3.3.2 Process 89
3.3.3 State Filtering: Sample Size 91
3.4 Evaluation 93
3.5 Impact Evaluation and Notification 100
3.6 Attacking Attack Reduction 101
3.7 Conclusions 103

4 Evaluating Anomaly Detection Systems 105
4.1 Previous Work 109
4.2 Alarm Construction and Training 110
4.2.1 Raw Data 110
4.2.2 Alarm State Variables 112
4.2.3 Alarm Thresholds 114

4.3 Observable Attack Spaces and Detection Probability 118
4.3.1 Observable Attack Spaces and Alarms 119
4.3.2 Estimating the Detection Surface 120
4.3.3 Detection Surface Comparison 123
4.4 Modeling Acquisition 124
4.4.1 Acquisition Payoff Model 127
4.4.2 Calculating Alarm Efficiency 130
4.4.3 Determining a Minimum False Positive Rate 131
4.5 Modeling Reconnaissance 135
4.6 Conclusion 138

5 Conclusions **141**

List of Tables

2.1 Means and standard deviations (to three significant digits on standard deviation) for $\mathcal{V}_{\mathsf{am}}^{\mathsf{dur}}$, $\mathcal{C}_{\mathsf{am}}^{\mathsf{dur}}$, $\mathcal{V}_{\mathsf{pm}}^{\mathsf{dur}}$ and $\mathcal{C}_{\mathsf{pm}}^{\mathsf{dur}}$ for $\mathsf{dur} \in \{30\mathrm{s}, 60\mathrm{s}\}$ on March 12–16, 2007. 50

2.2 Count of servers observed between 12:00GMT and 13:00GMT on each of March 12–16, 2007. 51

2.3 True alarm percentages for combined detector (conditions (2.4) and (2.5)). 52

3.1 Breakdown of activity in recorded data set by flows and bytes (bytes in parentheses); source data is all SSH traffic from September 10th to 24th, 2007. 85

4.1 Summary of Gaussian state variables in SSH training set. . . 115

List of Figures

2.1 Attributes of Oracle traffic on March 5th, 2007; start time of π is on x-axis; $\mathsf{dur} = 60$s. 41

2.2 Atributes and model of Oracle traffic (after filtering) on March 5th, 2007. Start times and dur as in Figure 2.1. 42

2.3 Distributions for Oracle over March 12–16, 2007, fitted to normal distributions. 44

2.4 Graphical representation of attacks, where "C", "S" and "A" denote a client, server and attacker-controlled bot, respectively. The left hand attack affects total graph size ($v(\Lambda_\pi)$), but not largest component size. The attack on the right affects largest component size ($c(\Lambda_\pi)$), but not total graph size. 48

2.5 Contributions of conditions (2.4) and (2.5) to true alarms in Table 2.3. For clarity, only true alarms where $\frac{v(\Lambda_\pi)-\mu_V^{\Pi,\mathsf{dur}}}{\sigma_V^{\Pi,\mathsf{dur}}} \leq 20$ or $\frac{c(\Lambda_\pi)-\mu_C^{\Pi,\mathsf{dur}}}{\sigma_C^{\Pi,\mathsf{dur}}} \leq 20$ are plotted. 54

2.6 Effects of eliminating high degree vertices n from FTP attack traffic logs Λ. 59

2.7 Attacker identification accuracy of (2.8); $\mathsf{hitListPerc} \in \{25\%, 50\%, 75\%\}$, $|\mathsf{hidegree}| = 10$, $|\mathsf{bots}| = 5$. 62

2.8 Accuracy of (2.8) versus (2.9); $\mathsf{hitListPerc} = 25\%$, $|\mathsf{hidegree}| = 10$, $|\mathsf{bots}| = 5$. 65

3.1 Communication of the three highest-degree SSH clients per thirty second period on the monitored network. As the figure shows, multiple clients appear to be communicating with hundreds or thousands of servers at a time, a likely indicator of scanning or failed harvesting. 71

3.2 Information flow diagram for the attack reduction system. . . 81

3.3 Contribution of significant and insignificant flows to traffic. . 86

3.4 Histogram of total graph size with closeup on normally distributed minimal area; this figure supports our hypothesis that the majority of the high-level traffic are simply HCFAs. 88

3.5 Impact of removing observations using shapfilt; note the decided increase in the W-statistic after removing all HCFAs. . 94

3.6 Impact of removing vertices after W exceeds θ_W for different sample sizes of equivalent composition. 96

3.7 Impact of state-level and log-level filtering on data set. 99

4.1 Distribution and normal approximations of h and d in observed data set. 116

4.2 Detection surface ($\mathcal{P}_{\mathsf{det}}^{\mathsf{all}}(a, s)$, as a percentage) for combined alarms. 122

4.3 Detection surfaces ($\mathcal{P}^{x}_{\mathsf{det}}(a, s)$, as a percentage) for individual alarms. 125

4.4 Plot of the payoff for acquisition attacks over the OAS. 129

4.5 False positive rates required to limit expected hosts compromised to 1. 133

4.6 Values of a and s for which $\mathcal{H}^{c}_{\mathsf{acq}}(a, s, k)$ can be limited to at most the specified value, using a threshold $\theta_c = \mu_C + 3.5\sigma_C$. . 135

4.7 Plot of the payoff for reconnaissance attacks over the OAS. . 138

Abstract

Anomaly detection systems offer the potential to identify new attacks before signatures are identified. To do so, these systems build models of normal user activity from historical data and then use these models to identify deviations from normal behavior caused by attacks.

In this thesis, we develop a method of anomaly detection using *protocol graphs*, graph-based representations of network traffic. These protocol graphs model the social relationships between clients and servers, allowing us to identify clever attackers who have a *hit list* of targets, but don't understand the relationships these targets have to each other.

While this method can identify subtle attacks, anomaly detection systems and IDS in general are challenged by the rise of large-scale industrialized attacks conducted by botnets. The attackers who use botnets have an active interest in acquiring new hosts, leading to a general form of attack we refer to as *harvesting*. Harvesting attacks consist of a constant stream of low-success high-volume attempts to take over multiple hosts. Because attackers face relatively little risk of detection, harvesting attacks are conducted continuously. These attacks result in a constant stream of garbage traffic that can mistrain an anomaly detector, if the detector assumes that attacks are rare. Furthermore, since harvesting attacks have such a low success rate, they generally represent minimal risk to a network, treating all attacks as equivalent raises the alarm rate extensively even when the attacks represent little risk to the systems that the anomaly detector monitors.

To that end, we complement our anomaly detection system by developing

a novel training method that can eliminate hostile activity even when it makes up the majority of logged traffic. Using this training method, we are able to increase the sensitivity of our detection method by two orders of magnitude, in order to detect subtle and successful compromises.

Finally, we examine the impact of our anomaly detection system on attacks by developing a novel payoff-based evaluation method. This approach treats alarms as a design specification to the attacker and demonstrates that by using alarms in combination, we can develop a system that caps the attacker's maximum effectiveness. However, we also show that all the systems we examine (ours and otherwise) have specific limits to their detection capabilities which reward a subtle attacker.

Acknowledgements

Even if there's only one name on the cover, a thesis is the product of a team of people who alternately inspire, guide, goad, question and support the person doing the actual writing. I should therefore first thank my inordinately patient advisor, Mike Reiter, for performing all of the above tasks in roughly equal portions. Especially during those periods when it seemed like I was about to start writing Latin poetry than actually work on the problem in front of me.

My thanks also go to my committee: John McHugh, Hui Zhang and Srinivasan Seshan, all of whom provided informative feedback on the development of this thesis and were very patient with a sudden shift over to ssh error correction halfway through.

My thanks also go to the platoons of people who helped create and support the data that made this work possible, both at the CERT and elsewhere. There isn't sufficient space to list all of the people involved, so if you aren't on the list of Tan Dang, Roman Danyliw, Jim Downey, Michael Duggan, Carrie Gates, Jeff Jaime, Marc Kellner, Andrew Kompanek, Sean McAllister, Jim McCurley, Tim Shimeall, Mark Thomas, Brian Trammell, and Greg Virgin, that's only because a list of everyone involved would be longer than the thesis.

Over the years I've spent at CMU, I've had the honor of working with certain faculty who have been both inspiring and generous with their time even when I was bouncing around as a staff developer. My gratitude also therefore to Jay Kadane, Phil Koopman, Jon Peha, Brian Quinn and Eswaran

Subramanian.

Without my parents I'd have never gotten here, and I have to thank them both for their support over the past few years and their unending confidence that I would, at some point, actually wrap up this work and go onto something else.

Dedicated to the memory of Suresh Konda

Chapter 1

Introduction

A *protocol graph* is a graph-based model of network traffic on a particular protocol; in the context of this work "protocol" means a service running on top of TCP or UDP, such as SSH, DNS or SMTP. Protocol graphs are constructed from traffic logs and represent the activity of that protocol over a short time — in these graphs, the vertices are the entities engaged in the protocol (*e.g.*, the clients, servers or peers), and the edges indicate that a communication took place between two entities.

The structure of a protocol graph reflects the social relationships between users of a particular protocol. As an example of this, consider a protocol graph representing SSH traffic: since SSH users must use passwords or keys to log onto servers, an SSH protocol graph will consist of multiple discrete components, each of which represents a group of users with access to a particular server. In comparison, a protocol graph representing HTTP traffic will have far fewer components which are much larger, as HTTP servers generally provide open access.

We hypothesize that the social structures described by protocol graphs provide us with a useful method for detecting anomalies caused by stealthy attackers. An example of such an attacker is a hit-list scanner, who uses a list of IP addresses gained from previous reconnaissance to identify vulnerable hosts. An example of a hit-list scanner includes the SSH attackers observed by Alata *et al.* [3] in their deployments and studies of high-interaction honeypots. In Alata's honeypot studies, there is evidence that attackers would split their efforts into multiple phases of reconnaissance and exploit. By doing so, attackers were able to ensure that the bots used in the actual attacks would come from IP addresses far removed from the scanning bots. IP addresses that blindly scanned the network might be identified and blocked, but the information acquired by the bots at those addresses would be passed on to bots at different locations, who could then apply that hit list to avoid detection.

Protocol graphs provide a method for identifying these attacks because they model the social relationship between hosts. Even if an attacker knows that a particular SSH server is present on a particular network, he is unlikely to know which hosts on the network use that server, and which other SSH servers *those* hosts communicate with.

Protocol graphs therefore provide a potential mechanism for detecting particularly subtle *harvesting attacks*. Harvesting attacks were first identified by Mirkovic and Reiher [39] as part of a general two-phase pattern for DDoS attacks. During the first phase of one of DDoSes, an attacker acquires hosts to build a network of DDoS bots. During the second phase, the attacker uses these bots to actually DDoS the target.

Since Mirkovic and Reiher identified this differentiation between acquisition and assault, botnets have been used for other purposes in electronic crime and network attacks. The key feature of all of these attacks, examples of which are given by Freiling *et al.* [18], is that the attacker uses a large number of bots to implement the attack.

A particularly important use of large botnets is to hide the identity and intent of attackers. The value of anonymous bots to attackers was first demonstrated by Ramachandran *et al.* [48]'s study of blacklists, which provided evidence that attackers have a preference for non-blacklisted hosts. However, with access to hundreds or thousands of addresses, an attacker simply switches to a new host. The most aggressive example of this is the Fast-flux phishing network [41], which uses a large proxy network of occupied hosts that hide the identity of a "mothership" server. Fast-flux is distinctive in that it not only uses bots to hide identity, but that it aggressively switches out bots, expending several hundred a week.

The need for large numbers of expendable bots leads to a new form of attacker that we term an *opportunistic attacker* [12]. An opportunistic attacker has minimal interest in a target except insofar as the target is exploitable — once an attacker has acquired a host, he will reconfigure it for his own purposes. We can therefore characterize opportunistic attackers as more interested in the *quantity* of hosts they control rather than the *qualities* of any particular host.

By switching between bots as described above, an attacker does not need to be subtle with all his bots, just the right ones. An attacker can afford to expend bots in very clumsy attacks and then use other bots for subtle

and stealthy followups. For network traffic analysis, this directly leads to a pollution problem caused by a constant stream of failed takeover attempts.

This problem is particularly challenging for anomaly detection, because the promise of anomaly detection is that it can detect novel attacks by identifying changes in network behavior [15]. However, as Gates and Taylor [23] note, anomaly detection systems require access to clean and labeled data, and this data is generally labeled by assuming that past history is largely free of attacks. In the noise from constant clumsy attacking, an anomaly detection system will be unable to identify subtle or successful attacks.

Furthermore, even if an anomaly detection system can be trained to identify subtle attacks, attackers will still engage in clumsy ones. In order for an anomaly detection system to be a viable IDS, it must be able to distinguish and diagnose different classes of attacks. In particular, an IDS must be at least as effective at identifying events where an attacker has the potential to damage at network as ones where the attacker has no effect.

The contributions of this thesis are an anomaly detection system that uses protocol graphs, a method for training such a system on constantly attacked data, and a method for evaluating the efficacy of such an IDS against different forms of attacks. The presence of automated harvesting leads to novel problems beyond the simple identification of attacks. Constant attacks impact the training and evaluation of IDS, as it is no longer of question of *whether* a network is being attacked, but rather the type and impact of that attack – both on the attacker and defender.

1.1 Anomaly Detection Using Protocol Graphs

Our anomaly detection system uses a novel network traffic representation we call a *protocol graph*. A protocol graph is a graph-based representation of the network traffic observed on a particular protocol over a limited period of time. In protocol graphs, the vertices represent hosts using a particular protocol (*e.g.*, SMTP) and edges represent communications between those hosts.

The advantage of using protocol graphs over other detection methods is that the protocol networks mapped by graphs should have social attributes that are reflected in the graph structure. For example, as shown in this thesis, protocols that require some form of authentication for access (such as SSH or Oracle's database client/server protocol), produce protocol graphs broken into discrete components.

We hypothesize that protocol graphs are a viable method for detecting subtle attacks, specifically hit-list attacks such as the one used for the Witty worm [54]. A hit-list attacker will know which IP addresses point to actual hosts, so they will not produce the type of errors or magnitude of traffic that scan detection systems such as TRW [26] use to identify attackers. However, even if an attacker knows which addresses on a network they can connect to, they should not know how other parties on the network normally interact with those hosts, and consequently risk detection by disrupting measurable attributes of the corresponding protocol graph.

To test our hypothesis, we develop a simple protocol-graph based anomaly detection system which monitors two attributes of a protocol graph: the

graph size, v, and the largest component size c. We demonstrate that in the absence of attack, these qualities can be modeled using a Gaussian distribution and their values predicted using simple statistical techniques. Furthermore, using a sequence of simulated hit-list attack, we demonstrate that an attacker will affect both attributes.

We refer to these attacks as *graph inflation* or *component inflation*. In graph inflation, a bot increases the size of the current protocol graph by communicating with absent or nonexistent hosts. In component inflation, a bot increases the size of the largest component by connecting components which are normally separate. Depending on the protocol involved and the rate at which it hits targets, an attacking bot will trigger alarms based on graph inflation, component inflation, or both.

In addition to demonstrating that we can use protocol graphs to detect anomalies, we introduce a graph based diagnostic mechanism that uses changes in the total number of components to identify attackers. While the total number of components is not a sufficiently stable value to use as an alarm, a bot's impact on the component count is observable. We demonstrate that when a hit-list bot's traffic is removed from a network, the total component count of the graph will increase. This increase is both distinctive and characteristic of hit-list bots — when normal users are removed from the graph, this same increase does not occur.

These results demonstrate that we can identify anomalies and attackers using graph manipulations and provides us a coherent model of anomaly detection and attacker identification using protocol graphs as a model.

1.2 Training IDS on Noisy Data

Recall that the the total graph size and largest component size of a protocol graph can be modeled by a Gaussian distribution *in the absence of attacks*. However, because attackers have an active interest in acquiring additional hosts for their botnets, and because they have little concern about retribution, they regularly scan multiple protocols for viable targets. On a large network, such as the one we observe, protocols such as SSH are scanned literally every minute.

In order to compensate for this, we develop a training mechanism that integrates two stages of filtering to recover a normal distribution. The model is based on the assumption that instead of eliminating *all* attacks from training, we specifically intend to remove *clumsy* attacks, and that these attacks will be common and easily identified. We classify clumsy attacks as HCFAs: High Connection Failure Attacks; in traffic logs, HCFAs appear as long sequences of failed connections from a particular host.

Our attack filtering method combines two forms of filtering: *log-level* and *state-level*. Log-level filtering eliminates individual log records when there is evidence that the record is an HCFA. For example, for flow records, evidence that a flow is not a complete session (*e.g.*, it has less than 3 packets, no ACK flags or no payload beyond TCP options) is a strong indicator that the flow describes an HCFA. Log-level filtering is intended as a coarse first step to reduce the volume of traffic examined by more sophisticated models. Managing the volume of traffic from scanners is a significant performance problem, especially on large networks; an attacker scanning a single /27 will

normally generate more flows than a legitimate SSH session. The goal of log-level filtering is therefore to eliminate indicia of hostile activity without eliminating indicia of normal user activity.

Log-level filtering reduces the amount of data that state-level filtering must process. The intuition motivating state-level filtering is that if a traffic log record was part of a scan, but passed log-level filtering, then that indicates a common mode failure on the part of the log-level filter. In such a case, we will see many traffic records associated with that particular scan. State-level filtering therefore operates by assuming that the state attribute monitored (such as graph size or largest component size) is Gaussian in the absence of attacks, and that the attacker can only *increase* the state attribute. For example, an attacker can increase the total graph size of a protocol graph by scanning, but they cannot decrease that graph because they can't eliminate normal activity. Using a simple normality test and a progressive elimination process, we can recover the body of traffic which *is* Gaussian and then use those results to train an anomaly detector.

1.3 Evaluating IDS Impact

The automated scanning discussed in §1.2 comes from automated harvesting attempts by attackers. In this section of our work, we evaluate the ability of an IDS to deter harvesting attempts.

In order to do so, we make two assumptions about attacker and defender behavior. We assume that the attacker is a rational entity — he is interested in acquiring new hosts without excess risk to himself. Furthermore, we

assume that defenders are vigilant entities — if they find a host on their network has been damaged or occupied, they will attempt to rectify this.

Using these two assumptions, we build a model of IDS evaluation based around what we term an *observable attack space* or OAS. An OAS is the space of attacks that an attacker can conduct as observed by a particular logging mechanism. For example, an attacker observed with NetFlow traffic can conduct many attacks (such as password guessing and buffer overflows) that, from the log data, will appear to be similar. Consequently, attacks such as scanning and automated botnet harvesting, by virtue of the data collected, will appear to be identical and can only be differentiated by the degree to which an attack is carried out. We quantify that degree as an attacker's *aggressiveness*, the number of IP addresses they contact during an attack, and their *success rate*, the number of IP addresses they contact which are actually present.

We contend that the observable attack space can be used as a form of design specification for an attacker. Attacks are represented by points on the OAS, and the probability that a particular attack can be detected is determinable via simulation. Using these results, we calculate a *detection surface* over the OAS, and then can apply a payoff function to the detection surface to determine the attacker's results in each case. We can, at this point, treat the attack as a zero-sum game — the attacker wins by taking over hosts, while the defender wins by reclaiming hosts after they have been taken over.

This methodology allows us to examine the successes and failures of various attack detection mechanisms as a function of attacker behavior, expand-

ing traditional ROC-based analysis into a payoff-based framework. Using the OAS/Detection Surface approach, we are able to compare various forms of IDS, including our own protocol-graph based approaches and Jung *et al.*'s TRW [26] method to evaluate how well they operate against various classes of attacks. We find, for example, that our largest component size protocol graph detection and TRW complement each other.

We evaluate anomaly detection mechanisms individually and in concert. In addition, we develop two models of attacker payoff — one for hit-list attacks and one for scanning. Using these payoff models, we are able to ask questions about the minimum false positive rate (FPR) required to detect attacks and determine what FPR is required to effectively limit attacker behavior.

1.4 Outline of the Work

The core of this thesis is divided into three chapters. Chapter 2 describes the construction and use of protocol graphs for anomaly detection. Chapter 3 describes the problem of training protocol graphs on protocols that are constantly attacked. Chapter 4 compares protocol graph based detection and other anomaly detection methods by examining their potential impact on attacker behavior.

Chapter 2 describes a simple anomaly detection mechanism. In this chapter, we develop an anomaly detector for extremely subtle attacks by using protocol graphs. This analysis studies two particular graph attributes: the total graph size (v) and the largest component size (c). We demonstrate

that these attributes are, in the absence of attacks, predictable. In addition, we demonstrate that protocol-graph based anomaly detection mechanisms can be used to identify attacks which are, on a flow-by-flow basis, extremely subtle. Specifically, since protocol graphs mirror social structure, an attacker with no knowledge of the interrelationships between hosts on a network will damage a protocol graph's structure in a directly observable fashion.

Chapter 3 addresses the problem of training IDS to effectively use a threshold-based anomaly detection system. IDS training is complicated by the presence of a large number of extremely clumsy attackers who conduct blatantly obvious scans. If a threshold-based IDS does not address these common attacks, the resulting thresholds will be so high as to make anomaly detection meaningless. We address this problem by combining filtering on both individual records and the aggregate system. By doing so, we are able to increase the sensitivity of our detection mechanism by two orders of magnitude.

Chapter 4 compares protocol graph based detection methods against other harvesting detection attacks. To do so, we develop a novel IDS evaluation scheme that focuses on the potential impact an alarm can have on an attacker's goals, rather than focusing exclusively on the IDS' capacity to detect attacks. Using this method, we are able to identify systematic weaknesses of several IDS. In addition, we are able to use this method to address the problem of minimum false positive rates: how high an FPR an IDS must have in order to prevent attackers from reaching their goals.

Chapter 2

Protocol Graphs

Large numbers of Internet worms have prompted researchers to develop a variety of anomaly-based approaches to detect these attacks. Examples include monitoring the number of failed connection attempts by a host (*e.g.*, [10, 51, 71]), or the connection rate of a host to new targets (*e.g.*, [64, 52]). These systems are designed to detect abnormally frequent connections and often rely on evidence of connection failure, such as half-open TCP connections. To avoid detection by these systems, an attacker can use a *hit list* [57] generated previous to the attack or generated by another party [4]. An attacker using an accurate hit list contacts only targets known to be running an accessible server, and therefore will not trigger an alarm predicated on connection failure. By constraining the number of attack connections initiated by each attacker-controlled *bot*, the attacker could compromise targets while evading detection by most (if not all) techniques that monitor the behavior of individual hosts or rely on connection failures.

In this chapter, we propose a new detection method, based on moni-

toring *protocol graphs*. A protocol graph is a graph-based representation of a traffic log for a single protocol. In a protocol graph, the vertices represent the IP addresses used as clients or servers for a particular protocol (*e.g.*, FTP), and the edges represent communication between those addresses. We expect that a protocol graph will have properties that derive from its underlying protocol's design and use. For example, we expect that since Oracle communications require password authentication and HTTP interactions do not, a protocol graph representing Oracle will have more connected components than a protocol graph representing HTTP. Conversely, while the HTTP graph will have fewer connected components, its largest connected component will be much larger relative to the total size ofthe graph.

Our detection approach focuses on two graph properties: the number of vertices comprising the graph ("graph size") and the number of vertices in the largest connected component of the graph ("largest component size") for traffic logs collected in a fixed duration. We hypothesize that while an attacker may have a hit list identifying servers within a network, he will not have accurate information about the activity or audience for those servers. As a consequence, a hit-list attack will either artificially inflate the number of vertices in a protocol graph, or it will connect disjoint components, resulting in a larger than expected size for the largest connected component.

To test this, we examine protocol graphs generated from traffic of several common protocols as observed in a large (larger than a /8) network. Specifically, we examine HTTP, SMTP, Oracle and FTP. Using this data, we confirm that protocol graphs for these protocols have predictable graph and largest component sizes. We then inject synthetic hit-list attacks into the

network, launched from one or more attacker-controlled bots, to determine if these attacks detectably modify either graph size or largest component size of the observed protocol graphs. The results of our study indicate that monitoring graph size and particularly largest component size is an effective means of hit-list worm detection for a wide range of attack parameters and protocols. For example, if tuned to yield one false alarm per day, our techniques reliably detect aggressive hit-list attacks and detect even moderate hit-list attacks with regularity, whether from one or many attacker-controlled bots.

Once an alarm is raised, an important component of diagnosis is determining which of the vertices in the graph represent bots. We show how to use protocol graphs to achieve this by measuring the *number* of connected components resulting from the removal of high-degree vertices in the graph. We demonstrate through extensions to our analysis that we can identify bots with a high degree of accuracy for FTP, SMTP and HTTP, and with somewhat less (though still useful) accuracy for Oracle. We also show that our bot identification accuracy exceeds what can be achieved by examining vertex degree alone.

While there are many conceivable measures of a protocol graph that might be useful for detecting worms, any such measure must be efficient to monitor if detection is to occur in near-real-time. The graph size and largest component size are very advantageous in this respect, in that they admit very efficient computation via well-known *union-find* algorithms (see [20]). A union-find algorithm implements a collection of disjoint sets of elements supporting two operations: two sets in the collection can be merged (`union`),

and the set containing a particular element can be located (`find`). In our application, the elements of sets are IP addresses, and the sets are the connected components of the protocol graph. As such, when a new communication record is observed, the set containing each endpoint is located (two `find` operations) and, if these two sets are distinct, they can be merged (a `union` operation). Using well-known techniques, communication records can be processed in amortized time that is effectively a small constant per communication record, and in space proportional to the number of IP addresses observed. By comparison, detection approaches that track connection rates to new targets (*e.g.*, [64, 52]) require space proportional to the number of unique connections observed, which can far exceed the number of unique IP addresses observed. While our attacker identification that is performed following an alarm incurs costs similar to these prior techniques, we emphasize that it can be proceed simultaneously with reactive defenses being deployed and so need not be as time-critical as detection itself.

To summarize, the contributions of this chapter include (i) defining protocol graphs and detailing their use as a hit-list attack detection technique; (ii) demonstrating through trace-driven analysis on a large network that this technique is effective for detecting hit-list attacks; (iii) extending protocol graph analysis to infer the locations of bots when hit-list worms are detected; and (iv) describing efficient algorithms by which worm detection and bot identification can be performed, in particular with detection being even more efficient than techniques that focus on localized behavior of hosts.

Our chapter proceeds as follows. § 2.1 summarizes previous relevant work. § 2.2 describes protocol graphs and the data we use in our analysis.

§ 2.3 examines the size of graphs and their largest components under normal circumstances, and introduces our anomaly detection technique. In § 2.4, we test our technique through simulated hit-list attacks. We extend our approach to identify attackers in § 2.5. § 2.6 addresses implementation issues. § 2.7 summarizes our results and discusses ongoing and future research.

2.1 Previous Work

Several intrusion detection and protocol identification systems have used graph-based communication models. Staniford *et al.*'s GrIDS system [58] generates graphs describing communications between IP addresses or more abstract entities within a network, such as the computers comprising a department. A more recent implementation of this approach by Ellis *et al.* [16] has been used for worm detection. Karagiannis at al. [29] develop a graphical traffic profiling system called BLINC for identifying applications from their traffic. Stolfo *et al.*'s [61] Email Mining Toolkit develops graphical representations of email communications and uses them to detect email viruses and worms.

In all of these cases, the systems detect phenomena of interest based on localized (*e.g.*, per-vertex or vertex neighborhood) properties of the graph. GrIDS generates rules describing how internal departments or organizations communicate, and can develop threshold rules (*e.g.*, "trigger an alarm if the vertex has degree more than 20"). Ellis' approach uses combinations of *link predicates* to identify a host's behavior. Karagiannis' approach expresses these same communications using subgraph models called *graphlets*.

Stolfo *et al.*'s approach identifies *cliques* per user, to whom the user has been observed sending the same email, and flags emails that span multiple cliques as potentially containing a virus or worm. In comparison to these efforts, our work focuses on aggregate graph behavior (graph size and largest component size) as opposed to localized properties of the graph or individual vertices. Moreover, some of these approaches utilize more protocol semantics (*e.g.*, the event of sending an email to multiple users [61], or the expected communication patterns of an application [29]) that we do not consider here in the interest of both generality and efficiency.

Several empirical studies have attempted to map out the structure of application networks. Such studies of which we are aware have been conducted by actively crawling the application network in a depth- or breadth-first manner, starting from some seed set of known participants. For example, Broder *et al.* [8] studied web interconnectivity by characterizing the links between pages. Ripeanu *et al.* [49] and Saroiu *et al.* [50] similarly conducted such studies of Gnutella and BitTorrent, respectively. Pouwelse *et al.* [47] use a similar probe and crawl approach to identify BitTorrent networks over an 8-month period. Our work differs from these in that our techniques are purely passive and are assembled (and evaluated) for the purpose of worm detection.

Our protocol graphs are more closely related to the *call graphs* studied by Aiello *et al.* [2] in the context of the AT&T voice network. In a call graph, each vertex represents a phone number and each (directed) edge denotes a call placed from one vertex to another. Aiello *et al.* observe that the size of the largest connected component of observed call graphs is $\Theta(|V|)$,

where V denotes the vertices of the graph. These call graphs are similar to our protocol graphs, the primary differences being that call graphs are directed (the protocol graphs we study are undirected) and that they are used to characterize a different domain (telephony, versus data networks here). However, Aiello *et al.* studied call graphs to understand their basic structure, but not with attention to worm detection (and in fact we are unaware of worms in that domain).

2.2 Preliminaries

In this section, we investigate the construction and composition of protocol graphs. Protocol graphs are generated from traffic logs; our analyses use CISCO Netflow, but graphs can also be constructed using `tcpdump` data or server logs.

This section is structured as follows. § 2.2.1 describes the construction of protocol graphs and our notation for describing them and their properties. § 2.2.2 describes our source data.

2.2.1 Protocol Graphs

We consider a log file (set) $\Lambda = \{\lambda_1, \ldots, \lambda_n\}$ of traffic records. Each record λ has fields for IP addresses, namely source address $\lambda.\mathsf{sip}$ and destination address $\lambda.\mathsf{dip}$. In addition, $\lambda.\mathsf{server}$ denotes the address of the server in the protocol interaction ($\lambda.\mathsf{server} \in \{\lambda.\mathsf{sip}, \lambda.\mathsf{dip}\}$), though we emphasize that we require $\lambda.\mathsf{server}$ only for evaluation purposes; it is not used in our detection or attacker identification mechanisms.

Given Λ, we define an undirected graph $\mathfrak{G}(\Lambda) = \langle \mathfrak{V}(\Lambda), \mathfrak{E}(\Lambda) \rangle$, where

$$\mathfrak{V}(\Lambda) = \bigcup_{\lambda \in \Lambda} \{\lambda.\mathsf{sip}, \lambda.\mathsf{dip}\} \qquad \mathfrak{E}(\Lambda) = \bigcup_{\lambda \in \Lambda} \{(\lambda.\mathsf{sip}, \lambda.\mathsf{dip})\}$$

In addition to $\mathfrak{E}(\Lambda)$ and $\mathfrak{V}(\Lambda)$, we use $\mathfrak{K}(\Lambda)$ to describe the set of *connected components* of a graph. Each connected component is treated as a discrete set of vertices; the largest connected component of a graph $\mathfrak{G}(\Lambda)$ is denoted $\mathfrak{C}(\Lambda) \subseteq \mathfrak{V}(\Lambda)$. Note that by construction, $\mathfrak{G}(\Lambda)$ has no connected component of size one (*i.e.*, an isolated vertex); all components are of size two or greater.[1]

We denote by Λ_π a log file that is recorded during the interval $\pi \subseteq$ [00:00GMT, 23:59GMT] on some specified date. For each log file Λ_π, we define the *graph size* as:

$$v(\Lambda_\pi) \equiv |\mathfrak{V}(\Lambda_\pi)| \tag{2.1}$$

The *largest component size* as:

$$c(\Lambda_\pi) \equiv |\mathfrak{C}(\Lambda_\pi)| \tag{2.2}$$

And the *component set size* as:

$$k(\Lambda_\pi) \equiv |\mathfrak{K}(\Lambda_\pi)| \tag{2.3}$$

Where the vertical bars indicate the cardinality of the corresponding set.

We model the graph size and largest component size for logs Λ_π collected

[1]It is possible for various logging mechanisms, under specific circumstances, to record a flow from a host to itself. We eliminate those records for this work.

in dur-length time intervals $\pi \subseteq \Pi$, using the random variables $\mathcal{V}_{\Pi}^{\mathsf{dur}}$ and $\mathcal{C}_{\Pi}^{\mathsf{dur}}$. For example, in the following sections we will focus on Π = [00:00GMT, 11:59GMT] (denoted am) and Π = [12:00GMT, 23:59GMT] (denoted pm), and take $v(\Lambda_\pi)$ and $c(\Lambda_\pi)$ with $\pi \subseteq \mathsf{am}$ of length 60 seconds (s) as an observation of $\mathcal{V}_{\mathsf{am}}^{60\mathrm{s}}$ and $\mathcal{C}_{\mathsf{am}}^{60\mathrm{s}}$, respectively.

We calculate the means and standard deviations of these variables from vectors of observed values. A vector V_{Π}^{dur} consists of all the v observations in the interval Π using log files of duration dur. The vector C_{Π}^{dur} is the same vector consisting of c observations. We refer to the arithmetic mean of a vector V_{Π}^{dur} as $\mu_{V}^{\Pi,\mathsf{dur}}$ and the corresponding standard deviation as $\sigma_{V}^{\Pi,\mathsf{dur}}$.

2.2.2 Data Set

The source data for these analyses are CISCO Netflow traffic summaries collected on a large (larger than a /8) ISP network. We use collectors at the border of the network's autonomous intranets in order to record the internal and cross border network activity. Therefore, all protocol participants that communicate between intranets or with the Internet are observed. Netflow reports flow logs, where a flow is a sequence of packets with the same addressing information that are closely related in time. Flow data is a compact summary of network traffic and therefore useful for maintaining records of traffic across large networks.

Flow data does not include payload information, and as a result we identify protocol traffic by using port numbers. Given a flow record, we convert it to a log record λ of the type we need by setting λ.server to the IP address that has the corresponding service port; *e.g.*, in a flow involving

ports 80 and 3946, the protocol is assumed to be HTTP and the server is the IP address using port 80[2]. Protocol graphs constructed using log files with payload could look for banners within the payload to identify services.

The protocols used for analysis are listed below.

- HTTP: The HTTP dataset consists of traffic where the source or destination port is port 80 and the other port is ephemeral ($\geq$ 1024). HTTP is the most active protocol on the monitored network, comprising approximately 50% of the total number of bytes crossing the network during the workday.
- SMTP: SMTP consists of TCP traffic where the source or destination port is port 25 and the other port is ephemeral. After HTTP, SMTP is the most active protocol on the monitored network, comprising approximately 30% of the total number of bytes.
- Oracle: The Oracle dataset consists of traffic where one port is 1521 and the other port is ephemeral. While Oracle traffic is a fraction of HTTP and SMTP traffic, it is a business-critical application. More importantly, Oracle connections are password-protected and we expect that as a consequence any single user will have access to a limited number of Oracle servers.
- FTP: The FTP dataset consists of records where one port is either 20 or 21, and the other port is ephemeral. While FTP provides password-based authentication, public FTP servers are still available.

[2]We note that the construction process assumes that the flow in this context describes a legitimate interection; the difficulties involved in this assumption are addressed in Chapter 3

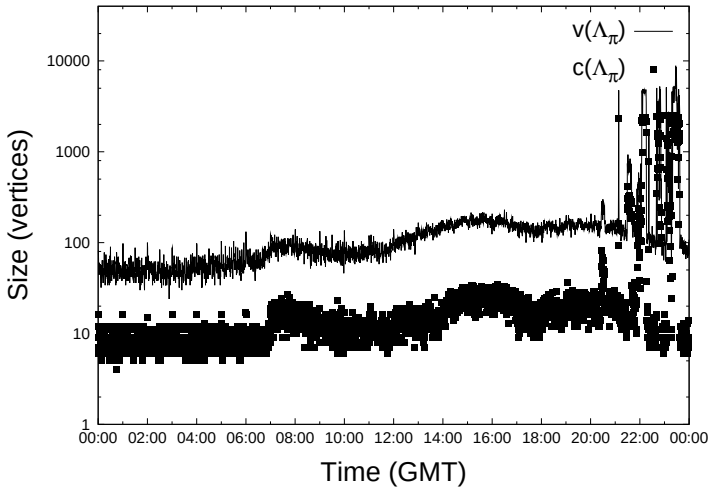
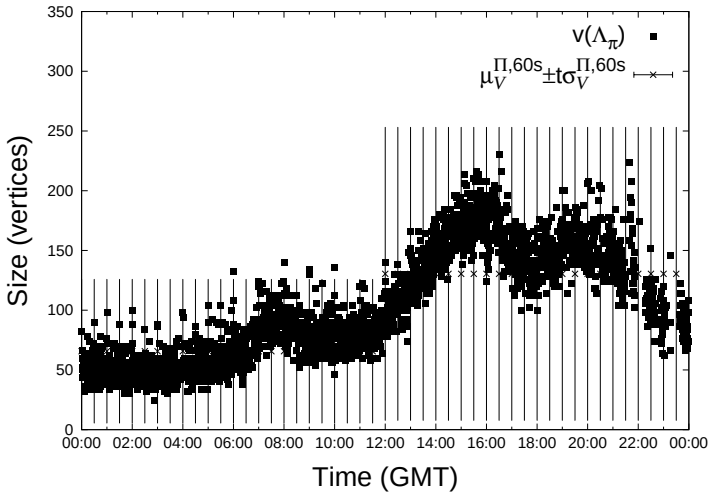


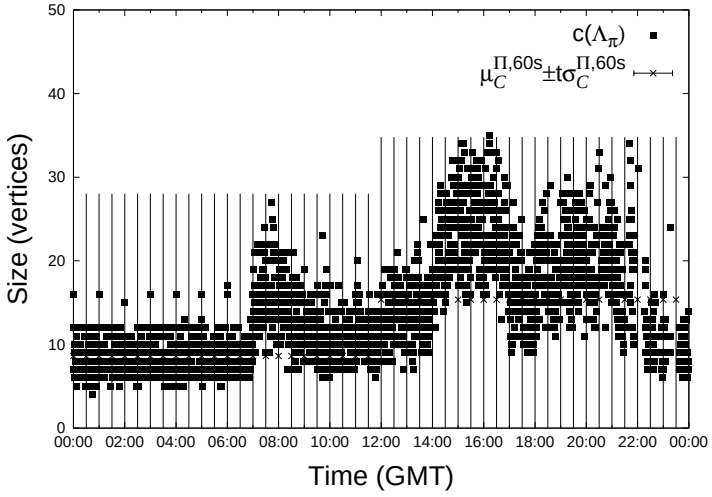
Figure 2.1: Attributes of Oracle traffic on March 5th, 2007; start time of π is on x-axis; $\mathsf{dur} = 60$s.

2.3 Building a Hit-List Worm Detector

In this section we describe the general behavior of protocol graphs over time, and show that the distributions of $\mathcal{V}_\Pi^{\mathsf{dur}}$ and $\mathcal{C}_\Pi^{\mathsf{dur}}$ can be satisfactorily modeled as normal for appropriate choices of Π and dur (§ 2.3.1). The parameters of these distributions change as a function of the protocol (HTTP, SMTP, FTP, Oracle), the interval in which logging occurs (Π), and the duration of log collection (dur). Nevertheless, in all cases the graph and largest component sizes are normally distributed, which enables us to postulate a detection mechanism for hit-list worms and estimate the false alarm rate for any detection threshold (§ 2.3.2).



(a) $v(\Lambda_\pi)$ after filtering



(b) $c(\Lambda_\pi)$ after filtering

Figure 2.2: Atributes and model of Oracle traffic (after filtering) on March 5th, 2007. Start times and dur as in Figure 2.1.

2.3.1 Graph Behavior Over Time

Figure 2.1 is a plot of the observed values of $v(\Lambda_\Pi)$ and $c(\Lambda_\Pi)$ for Oracle traffic on the monitored network for Monday, March 5th, 2007. Each logging interval π begins at a time indicated on the x-axis, and continues for $\mathsf{dur} = 60$s. In this figure, the x-axis represents the starting time for the observation (recorded in GMT) and the y-axis is the total size of the observation expressed as a count of vertices. Traffic between servers internal to the monitored network and their clients (internal or external to the monitored network) was recorded. Plots including external servers show the same business-cycle dependencies and stability. However, we ignore external servers because the vantage point of our monitored network will not allow us to see an external attack on an external server.

Figure 2.1 is plotted logarithmically due to the anomalous activity visible after 18:00GMT. At this time, multiple bots scanned the monitored network for Oracle servers. These types of disruptive events are common to all the training data; we identify and eliminate these scans using Jung *et al.*'s sequential hypothesis testing method [26]. In this method, scanners are identified when they attempt to connect to servers that are not present within the targeted network. This method will not succeed against hit-list attackers, as a hit-list attacker will only communicate with servers that are present on the network.

Figure 2.2(a)–(b) is a plot of the same activity as in Figure 2.1 after the scan events are removed: Figure 2.1(a) plots $v(\Lambda_\pi)$, while Figure 2.1(b) plots $c(\Lambda_\pi)$.

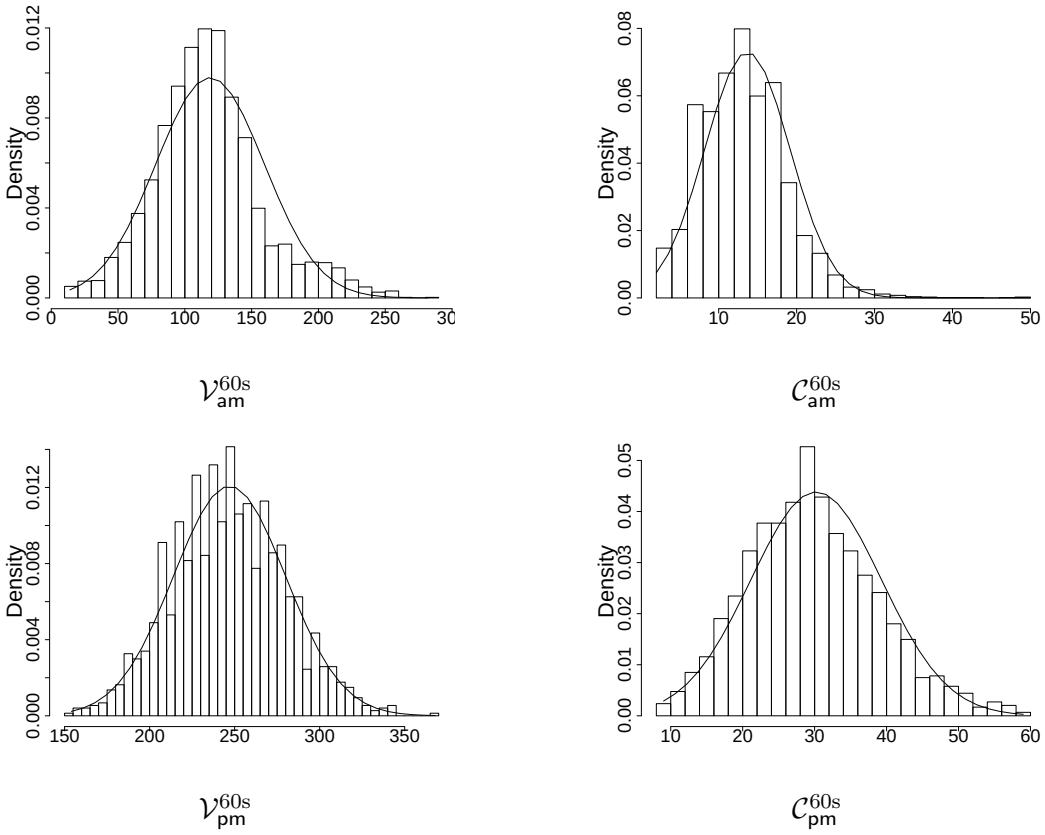


Figure 2.3: Distributions for Oracle over March 12–16, 2007, fitted to normal distributions.

Once scans are removed from the traffic logs, the distribution of traffic can be satisfactorily modeled with normal distributions. More precisely, we divide the day into two intervals, namely am = [00:00GMT, 11:59GMT] and pm = [12:00GMT, 23:59GMT]. For each protocol we consider, we define random variables $\mathcal{V}_{\mathsf{am}}^{60\mathrm{s}}$ and $\mathcal{V}_{\mathsf{pm}}^{60\mathrm{s}}$, of which the points on the left and right halves of Figure 2.2(a) are observations for Oracle, respectively. Similarly, we define random variables $\mathcal{C}_{\mathsf{am}}^{60\mathrm{s}}$ and $\mathcal{C}_{\mathsf{pm}}^{60\mathrm{s}}$, of which the points on the left and

right halves of Figure 2.2(b) are observations, respectively. By taking such observations from all of March 12–16, 2007 for each of $\mathcal{V}_{\mathsf{am}}^{60\mathrm{s}}$, $\mathcal{V}_{\mathsf{pm}}^{60\mathrm{s}}$, $\mathcal{C}_{\mathsf{am}}^{60\mathrm{s}}$ and $\mathcal{C}_{\mathsf{pm}}^{60\mathrm{s}}$, we fit a normal distribution to each effectively; see Figure 2.3.[3] Figure 2.3 are histograms: the x-axis describes the value of an observation, and the y-axis its frequency. The line drawn in each histogram is an estimated normal distribution using the empirically observed mean and standard deviation for that histogram.

On the left half of Figure 2.2(a), we plot $\mu_V^{\mathsf{am},60\mathrm{s}}$ as a horizontal line and $\gamma\ \sigma_V^{\mathsf{am},60\mathrm{s}}$ as error bars with $\gamma = 3.5$. We do similarly with $\mu_V^{\mathsf{pm},60\mathrm{s}}$ and $\gamma\ \sigma_V^{\mathsf{pm},60\mathrm{s}}$ on the right half, and with $\mu_C^{\mathsf{am},60\mathrm{s}}$ and $\gamma\ \sigma_C^{\mathsf{am},60\mathrm{s}}$ and $\mu_C^{\mathsf{pm},60\mathrm{s}}$ and $\gamma\ \sigma_C^{\mathsf{pm},60\mathrm{s}}$ on the left and right halves of Figure 2.2(b), respectively. The choice of $\gamma = 3.5$ will be justified below.

In exactly the same way, we additionally fit normal distributions to $\mathcal{V}_{\mathsf{am}}^{30\mathrm{s}}$, $\mathcal{C}_{\mathsf{am}}^{30\mathrm{s}}$, $\mathcal{V}_{\mathsf{pm}}^{30\mathrm{s}}$, and $\mathcal{C}_{\mathsf{pm}}^{30\mathrm{s}}$ for each protocol, with equally good results. And, of course, we could have selected finer-granularity intervals than half-days (am and pm), resulting in more precise means and standard deviations on, *e.g.*, an hourly basis. Indeed, the tails on the distributions of $\mathcal{V}_{\mathsf{am}}^{60\mathrm{s}}$ and $\mathcal{C}_{\mathsf{am}}^{60\mathrm{s}}$ in Figure 2.3 are a result of the coarse granularity of our chosen intervals, owing to the increase in activity at 07:00GMT (see Figure 2.2(a)). We elect to not refine our am and pm intervals here, however, for presentational convenience.

2.3.2 Detection and the False Alarm Rate

Our detection system is a simple hypothesis testing system; the null hypothesis is that an observed log file Λ does not include a worm propagation.

[3]For all protocols, the observed Shapiro-Wilk statistic is in excess of 0.95.

Recall from § 2.3.1 that for a fixed interval $\Pi \in \{\mathsf{am}, \mathsf{pm}\}$, graph size $\mathcal{V}_{\Pi}^{\mathsf{dur}}$ and largest component size $\mathcal{C}_{\Pi}^{\mathsf{dur}}$ are normally distributed with mean and standard deviation $\mu_V^{\Pi,\mathsf{dur}}$ and $\sigma_C^{\Pi,\mathsf{dur}}$, respectively. As such, for a dur-length period $\pi \subseteq \Pi$, we raise an alarm for a protocol graph $\mathfrak{G}(\Lambda_\pi) = \langle \mathfrak{V}(\Lambda_\pi), \mathfrak{E}(\Lambda_\pi) \rangle$ if either of the following conditions holds:

$$v(\Lambda_\pi) \;>\; \mu_V^{\Pi,\mathsf{dur}} + \gamma\ \sigma_V^{\Pi,\mathsf{dur}} \tag{2.4}$$

$$c(\Lambda_\pi) \;>\; \mu_C^{\Pi,\mathsf{dur}} + \gamma\ \sigma_C^{\Pi,\mathsf{dur}} \tag{2.5}$$

Recall that for a normally distributed random variable $\mathcal{X}$ with mean $\mu_{\mathcal{X}}$ and standard deviation $\sigma_{\mathcal{X}}$,

$$\mathbb{P}\left[\mathcal{X} \leq x\right] = \frac{1}{2}\left[1 + \mathsf{erf}\left(\frac{x - \mu_{\mathcal{X}}}{\sigma_{\mathcal{X}}\sqrt{2}}\right)\right] \tag{2.6}$$

where $\mathsf{erf}(\cdot)$ is the so called "error function" [30]. This enables us to compute the contribution of condition (2.4) to the false negative (alarm) rate FNR for a given threshold γ as $1 - \mathbb{P}\left[\mathcal{V}_{\Pi}^{\mathsf{dur}} \leq \mu_V^{\Pi,\mathsf{dur}} + \gamma\ \sigma_V^{\Pi,\mathsf{dur}}\right]$, and similarly for the contribution of condition (2.5) to FNR. Conversely, since $\mathsf{erf}^{-1}(\cdot)$ exists, given a desired FNR we can compute a threshold γ so that our FNR is not exceeded:

$$\gamma = \sqrt{2}\ \mathsf{erf}^{-1}\left(\frac{1}{2} - \frac{\mathsf{FNR}}{2}\right) \tag{2.7}$$

Note that the use of $\frac{\mathsf{FNR}}{2}$ in (2.7) ensures that each of conditions (2.4) and (2.5) contribute at most half of the target FNR and consequently that both conditions combined will yield at most the target FNR.

Finally, recall that each Λ_π represents one dur-length time period π, and

FNR is expressed as a fraction of the log files, or equivalently, dur-length time intervals, in which a false alarm occurs. We can obviously extrapolate this FNR to see its implications for false alarms over longer periods of time. For the remainder of this chapter, we will take as our goal a false alarm frequency of one per day (with $\mathsf{dur} = 60\text{s}$), yielding a threshold of $\gamma = 3.5$.

This estimate depends on accurate calculations for $\mu_V^{\Pi,\mathsf{dur}}$, $\mu_C^{\Pi,\mathsf{dur}}$, $\sigma_V^{\Pi,\mathsf{dur}}$, and $\sigma_C^{\Pi,\mathsf{dur}}$ for the time interval Π in which the monitoring occurs. In the remainder of this chapter, we will compute these values based on data collected on March 12–16, 2007.

2.4 Protocol Graph Change During Attack

We showed in § 2.3 that, for the protocols examined, $\mathcal{C}_{\mathsf{am}}^{\mathsf{dur}}$, $\mathcal{V}_{\mathsf{am}}^{\mathsf{dur}}$, $\mathcal{C}_{\mathsf{pm}}^{\mathsf{dur}}$ and $\mathcal{V}_{\mathsf{pm}}^{\mathsf{dur}}$ are normally distributed (§ 2.3.1), leading to a method for computing the false alarm rate for any given detection threshold (§ 2.3.2). In this section, we test the effectiveness of this detection mechanism against simulated hit-list attacks. § 2.4.1 describes the model of attack used. § 2.4.2 describes the experiment and our evaluation criteria. The detection results of our simulations are discussed in § 2.4.3.

2.4.1 Attack and Defense Model

We simulate hit-list attacks, as described by Staniford et al. [57]. A *hit list* is a list of target servers identified before the actual attack. An apparent example of a hit-list worm is the Witty worm: reports by Shannon and Moore [54] hypothesized that Witty initially spread via a hit list. Further analyses by

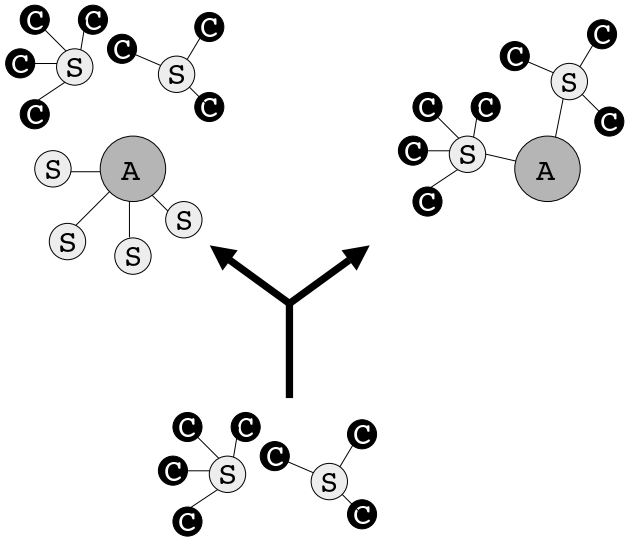


Figure 2.4: Graphical representation of attacks, where "C", "S" and "A" denote a client, server and attacker-controlled bot, respectively. The left hand attack affects total graph size ($v(\Lambda_\pi)$), but not largest component size. The attack on the right affects largest component size ($c(\Lambda_\pi)$), but not total graph size.

Kumar *et al.* [31] identified Witty's "patient zero" and demonstrated that this host behaved in a notably different fashion from subsequently-infected Witty hosts, lending support to the theory that patient zero used a hit list to infect targets.

We hypothesize that an attacker who has a hit list for a targeted network will be detectable by examining $v(\Lambda_\pi)$ and $c(\Lambda_\pi)$ where Λ_π is a log file recorded during a time interval π in which a hit-list worm propagated.

We assume that the attacker has the hit list, but has no knowledge of the targeted servers' current activity or audience. If this is the case, then the attacker contacting his hit list will alter the observed protocol graph through *graph inflation* or *component inflation*.

Figure 2.4 shows how these attacks impact protocol graphs. Graph inflation occurs when an attacker communicates with servers that are not active during the observation period π. When this occurs, the attacker artificially inflates the number of vertices in the graph, resulting in a value of $v(\Lambda_\pi)$ that is detectably large. The vertices of a protocol graph include both clients and servers, while the attacker's hit list will be composed exclusively of servers. As a result, we expect that graph inflation will require communicating with many of the hit-list elements (roughly $\gamma\ \sigma_V^{\Pi,\mathsf{dur}}$ for dur-length $\pi \subseteq \Pi$) to trigger condition (2.4).

Component inflation occurs when the attacker communicates with servers already present in Λ_π during the observation period π. When this occurs, the attacker will merge components in the graph, and $c(\Lambda_\pi)$ will be detectably large. In comparison to graph inflation, component inflation can happen very rapidly; it may occur in a detectable way if an attacker communicates with two servers. However, if the graph already has a small number of components (as is the case with SMTP), or the attacker uses multiple bots to attack, then the attack may not be noticed.

2.4.2 Experiment Construction

The training period for our experiments was March 12–16, 2007. We considered two different values for dur, namely 30s and 60s. Table 2.1 contains the

	HTTP		SMTP		Oracle		FTP	
r.v.	μ	σ	μ	σ	μ	σ	μ	σ
$\mathcal{V}_{\mathsf{am}}^{30\mathrm{s}}$	10263	878	2653	357	65.3	18.7	291.9	57.0
$\mathcal{C}_{\mathsf{am}}^{30\mathrm{s}}$	9502	851	2100	367	17.52	4.00	65.30	8.10
$\mathcal{V}_{\mathsf{pm}}^{30\mathrm{s}}$	16460	2540	3859	336	128.7	32.4	359.8	67.1
$\mathcal{C}_{\mathsf{pm}}^{30\mathrm{s}}$	15420	2420	3454	570	30.60	6.28	80.02	8.23
$\mathcal{V}_{\mathsf{am}}^{60\mathrm{s}}$	14760	1210	4520	634	111.8	28.1	467.4	76.9
$\mathcal{C}_{\mathsf{am}}^{60\mathrm{s}}$	13940	1180	4069	650	12.92	4.24	37.3	11.3
$\mathcal{V}_{\mathsf{pm}}^{60\mathrm{s}}$	23280	3480	6540	935	240.3	31.7	555.5	94.8
$\mathcal{C}_{\mathsf{pm}}^{60\mathrm{s}}$	22140	3320	6200	937	28.84	8.44	45.9	12.2

Table 2.1: Means and standard deviations (to three significant digits on standard deviation) for $\mathcal{V}_{\mathsf{am}}^{\mathsf{dur}}$, $\mathcal{C}_{\mathsf{am}}^{\mathsf{dur}}$, $\mathcal{V}_{\mathsf{pm}}^{\mathsf{dur}}$ and $\mathcal{C}_{\mathsf{pm}}^{\mathsf{dur}}$ for $\mathsf{dur} \in \{30\mathrm{s}, 60\mathrm{s}\}$ on March 12–16, 2007.

computed means and standard deviations for $\mathcal{V}_{\mathsf{am}}^{\mathsf{dur}}$, $\mathcal{C}_{\mathsf{am}}^{\mathsf{dur}}$, $\mathcal{V}_{\mathsf{pm}}^{\mathsf{dur}}$ and $\mathcal{C}_{\mathsf{pm}}^{\mathsf{dur}}$ for the training period and for each choice of dur, which are needed to evaluate conditions (2.4) and (2.5). As shown in Table 2.1, the largest component for HTTP and SMTP was close to the total size of the protocol graph after 60s, and the relative difference between the two values decreases as dur grows.

An important point illustrated in Table 2.1 is that the graph sizes can differ by orders of magnitude depending on the protocol. This demonstrates the primary argument for generating per-protocol graphs: the *standard deviations* in graph size and largest component size for HTTP and SMTP are larger than the mean sizes for Oracle and FTP.

For testing, we model our attack as follows. During a period π, we collect a log $\Lambda_{\pi}^{\mathsf{ctl}}$ of normal traffic. In parallel, the attacker uses a hit-list set $\mathsf{HitList}$ to generate its own traffic log Λ^{atk}. This log is merged with $\Lambda_{\pi}^{\mathsf{ctl}}$ to create a new log $\Lambda_{\pi}^{\mathsf{test}} = \Lambda_{\pi}^{\mathsf{ctl}} \cup \Lambda^{\mathsf{atk}}$. We then examine conditions (2.4) and (2.5) for

Protocol	Servers
SMTP	2818
HTTP	8145
Oracle	262
FTP	1409

Table 2.2: Count of servers observed between 12:00GMT and 13:00GMT on each of March 12–16, 2007.

interval $\Pi \in \{\mathsf{am}, \mathsf{pm}\}$ such that $\pi \subseteq \Pi$; if either condition is true, then we raise an alarm. In our tests, we select periods π of length dur from March 19, 2007, *i.e.*, the next business day after the training period.

To generate the $\mathsf{HitList}$ sets, we intersect the sets of servers which are observed as active on each of March 12–16, 2007 between 12:00GMT and 13:00GMT. The numbers of servers so observed are shown in Table 2.2. The attacker attacks the network over a protocol by selecting $\mathsf{hitListPerc}$ percentage of these servers (for the corresponding protocol) at random to form $\mathsf{HitList}$. The attacker (or rather, his bots) then contacts each element of $\mathsf{HitList}$ in order to generate the log file Λ^{atk}.

More precisely, we allow the attacker to use multiple bots in the attack; let bots denote the bots used by the attacker. We assume that bots do not appear in the log $\Lambda_{\pi}^{\mathsf{ctl}}$ and, given that dur is short, we keep bots static through the simulation. Each bot $\mathsf{bot}_i \in \mathsf{bots}$ is assigned a hit list $\mathsf{HitList}_i$ consisting of a random $\frac{|\mathsf{HitList}|}{|\mathsf{bots}|}$ fraction of $\mathsf{HitList}$. Each bot's hit list is drawn randomly from $\mathsf{HitList}$, but hit lists do not intersect. That is, $\bigcup_i \mathsf{HitList}_i = \mathsf{HitList}$ and for $i \neq j$, $\mathsf{HitList}_i \cap \mathsf{HitList}_j = \emptyset$. Λ^{atk} is generated by creating synthetic attack records from each bot_i to all members of $\mathsf{HitList}_i$.

		HTTP hitListPerc =				SMTP hitListPerc =			
dur	|bots|	25	50	75	100	25	50	75	100
30s	1	73	80	95	100	28	74	100	100
	3	72	80	95	100	25	50	97	100
	5	60	80	92	100	23	45	98	100
60s	1	68	80	100	100	20	50	70	80
	3	65	68	100	100	10	35	65	70
	5	65	63	100	100	5	30	60	55

		Oracle hitListPerc =				FTP hitListPerc =			
dur	|bots|	25	50	75	100	25	50	75	100
30s	1	100	100	100	100	100	100	100	100
	3	33	95	100	100	100	100	100	100
	5	16	87	99	100	100	100	100	100
50s	1	100	100	100	100	100	100	100	100
	3	28	100	100	100	100	100	100	100
	5	12	100	100	100	100	100	100	100

Table 2.3: True alarm percentages for combined detector (conditions (2.4) and (2.5)).

2.4.3 True Alarms

Table 2.3 shows the effectiveness of the detection mechanism as a function of dur for different hitListPerc values. hitListPerc varies between 25% and 100%. The value in each cell is the percentage of attacks that were detected by the system.

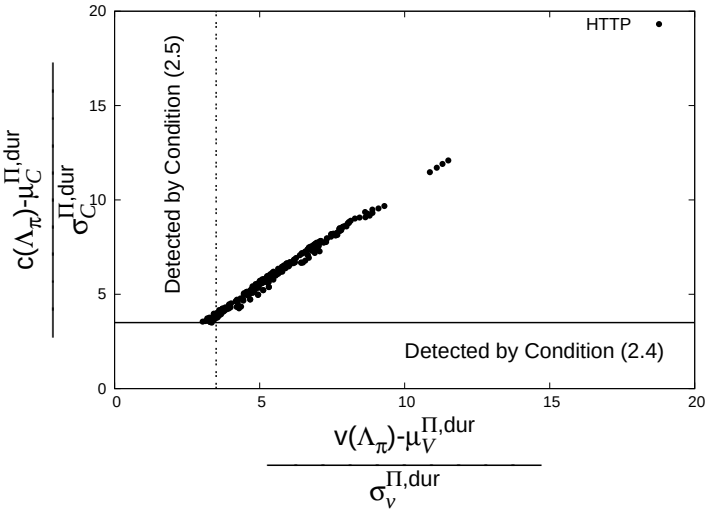
Table 2.3 sheds light on the effectiveness of our approach. The most aggressive worms we considered, namely those that contacted hitListPerc $\geq$ 75% of known servers (see Table 2.2) within dur = 30s, were easily detected: our tests detected these worms more than 90% of the time for all protocols

and all numbers of |bots|, and at least 95% of the time except in one case.

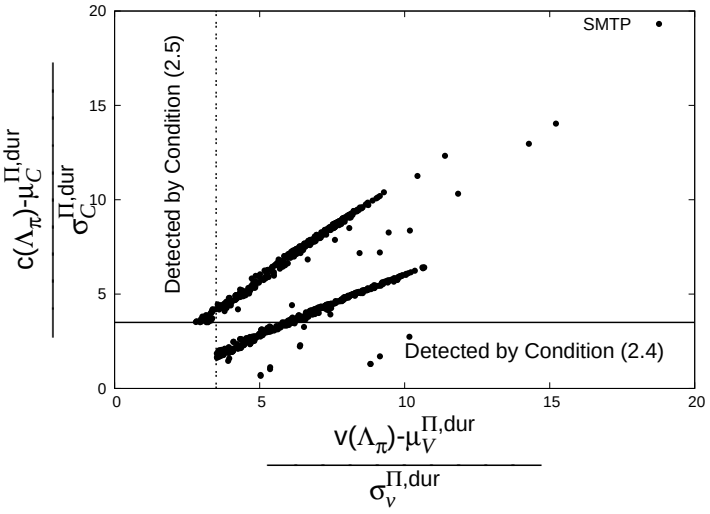
The table also sheds light on approaches an adversary might take to make his worm more stealthy. First, the adversary might decrease hitListPerc. While this does impact detection, our detection capability is still useful: *e.g.*, as hitListPerc is decreased to 50% in dur = 30s, the true detection rates drop, but remain 80% or higher for all protocols except SMTP. Second, the adversary might increase dur. If the adversary keeps hitListPerc $\geq 75\%$, then increasing dur from 30s to 60s appears to have no detrimental effect on the true alarm rate of the detector for HTTP, Oracle or FTP, and it remains at 60% or higher for SMTP, as well.

Third, the adversary might increase |bots|. Note that whereas the previous two attempts to evade detection necessarily slow the worm propagation, increasing |bots| while keeping hitListPerc and dur fixed need not—though it obviously requires the adversary to have compromised more hosts prior to launching his hit-list worm. Intuitively, increasing |bots| might decrease the likelihood of detection by our technique by reducing the probability that one bot_i will merge components of the graph and thereby trigger condition (2.5). (Recall that bots' individual hit lists do not intersect.) However, Table 2.3 suggests that in many cases this is ineffective unless the adversary simultaneously decreases hitListPerc: with hitListPerc $\geq 75\%$, all true detection rates with |bots| = 5 remain above 92% with the exception of SMTP (at 60% for dur = 60s). The effects of increasing |bots| may become more pronounced with larger numbers, though if |bots| approaches $\gamma\ \sigma(\mathcal{V}_{\Pi}^{\mathsf{dur}})$ then the attacker risks being detected by condition (2.4) immediately.

Figure 2.5 compares the effectiveness of conditions (2.4) and (2.5) for

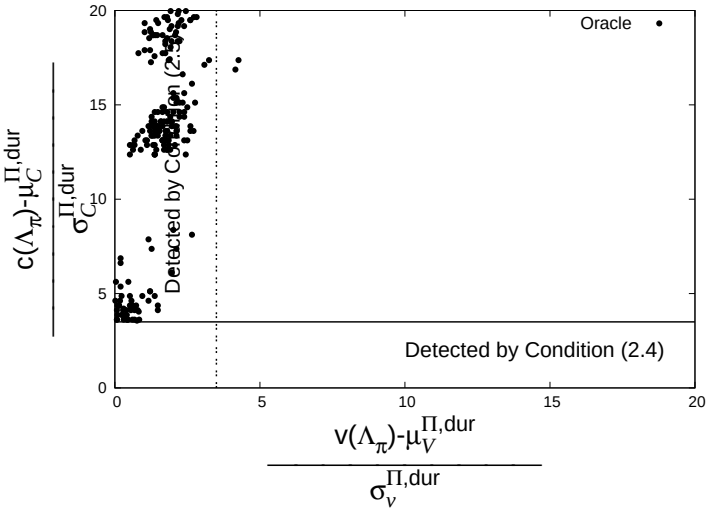


(a) HTTP

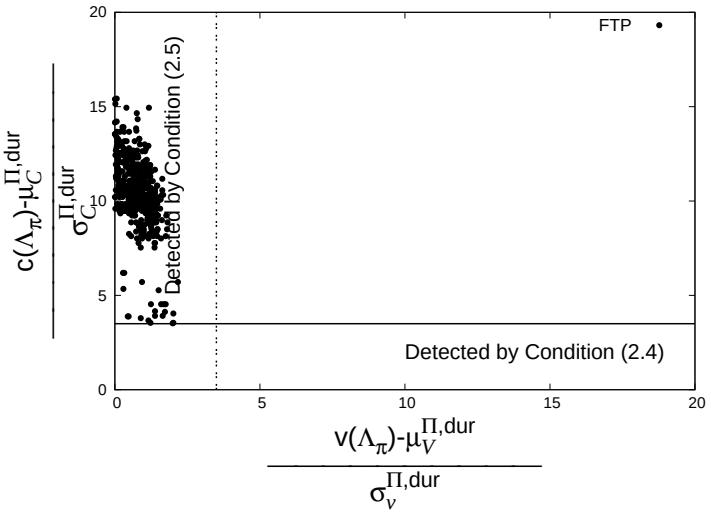


(b) SMTP

Figure 2.5: Contributions of conditions (2.4) and (2.5) to true alarms in Table 2.3. For clarity, only true alarms where $\frac{v(\Lambda_\pi)-\mu_V^{\Pi,dur}}{\sigma_V^{\Pi,dur}} \leq 20$ or $\frac{c(\Lambda_\pi)-\mu_C^{\Pi,dur}}{\sigma_C^{\Pi,dur}} \leq 20$ are plotted.



(c) Oracle



(d) FTP

Figure 2.5 continued.

each of the test protocols. Each plot in this figure is a scatter plot comparing the deviation of $v(\Lambda_\pi^{\mathsf{test}})$ against the deviation of $c(\Lambda_\pi^{\mathsf{test}})$ during attacks. Specifically, values on the x-axis are $\frac{v(\Lambda_\pi^{\mathsf{test}})-\mu_V^{\Pi,\mathsf{dur}}}{\sigma_V^{\Pi,\mathsf{dur}}}$, and values on the y-axis are $\frac{c(\Lambda_\pi^{\mathsf{test}})-\mu_C^{\Pi,\mathsf{dur}}}{\sigma_C^{\Pi,\mathsf{dur}}}$, for $\Pi \supseteq \pi$. The points on the scatter plot represent the true alarms summarized in Table 2.3, though for presentational convenience only those true alarms where $\frac{v\Lambda_\pi^{\mathsf{test}}-\mu_V^{\Pi,\mathsf{dur}}}{\sigma_V^{\Pi,\mathsf{dur}}} \leq 20$ or $\frac{c(\Lambda_\pi^{\mathsf{test}})-\mu_C^{\Pi,\mathsf{dur}}}{\sigma_C^{\Pi,\mathsf{dur}}} \leq 20$ are shown. Each plot has reference lines at $\gamma = 3.5$ on the horizontal and vertical axes to indicate the trigger point for each detection mechanism. That is, a "•" above the horizontal $\gamma = 3.5$ line indicates a test in which condition (2.5) was met, and a "•" to the right of the vertical $\gamma = 3.5$ line indicates a test in which condition (2.4) was met.

We would expect that if both conditions were effectively equivalent, then every "•" would be in the upper right "quadrant" of each graph. While HTTP (Figure 2.5(a)) shows this behavior, the other graphs demonstrate different behaviors. Figure 2.5(c) and (d) shows that the growth of $|C(\Lambda_\pi)|$ is an effective mechanism for detecting disruptions in both Oracle and FTP networks. The only protocol where graph inflation appears to be a more potent indicator than component inflation is SMTP. From this, we conclude that component inflation (condition (2.5)) is a more potent detector than graph inflation (condition (2.4)) when the protocol's graph structure is disjoint, but that each test has a role to play in detecting attacks.

2.5 Bot Identification

Once an attack is detected, individual attackers (bots) are identifiable by how they deform the protocol graph. As discussed in § 2.4.1, we expect a bot to impact the graph's structure by connecting otherwise disjoint components. We therefore expect that removing a bot from a graph $\mathfrak{G}(\Lambda)$ will separate components and so the number of connected components will increase.

To test this hypothesis, we consider the effect of removing all records λ involving an individual IP address from Λ. Specifically, for a log file Λ and an IP address n, define:

$$\Lambda^{\neg n} \quad = \quad \{\lambda \in \Lambda : \lambda.\mathsf{sip} \neq n \wedge \lambda.\mathsf{dip} \neq n\}$$

As such, $\mathfrak{G}(\Lambda)$ differs from $\mathfrak{G}(\Lambda^{\neg n})$ in that the latter includes neither n nor any $n' \in \mathfrak{V}(\Lambda)$ of degree one that is adjacent only to n in $\mathfrak{G}(\Lambda)$.

In order to detect a bot, we are primarily interested in comparing $\mathfrak{G}(\Lambda)$ and $\mathfrak{G}(\Lambda^{\neg n})$ for vertices n of high degree in $\mathfrak{G}(\Lambda)$, based on the intuition that bots should have high degree. Figure 2.6 examines the impact of eliminating each of the ten highest-degree vertices n in $\mathfrak{G}(\Lambda)$ from each log file Λ for FTP discussed in § 2.4 that resulted in a true alarm. Specifically:

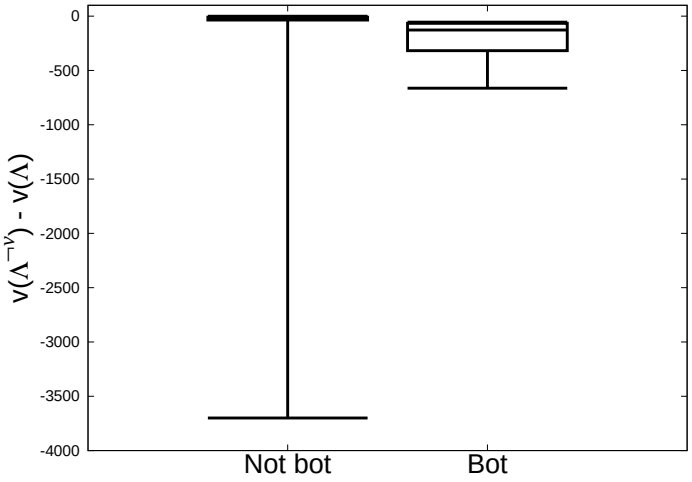
- Figure 2.6(a) represents the empirical distribution of $v(\Lambda^{\neg n}) - v(\Lambda)$, *i.e.*, the difference in the number of vertices due to eliminating n and all isolated neighbors, which will be negative;
- Figure 2.6(b) represents the empirical distribution of $c(\Lambda^{\neg n}) - c(\Lambda)$,

i.e., the difference in the size of the largest connected component due to eliminating n and all isolated neighbors, which can be negative or zero; and

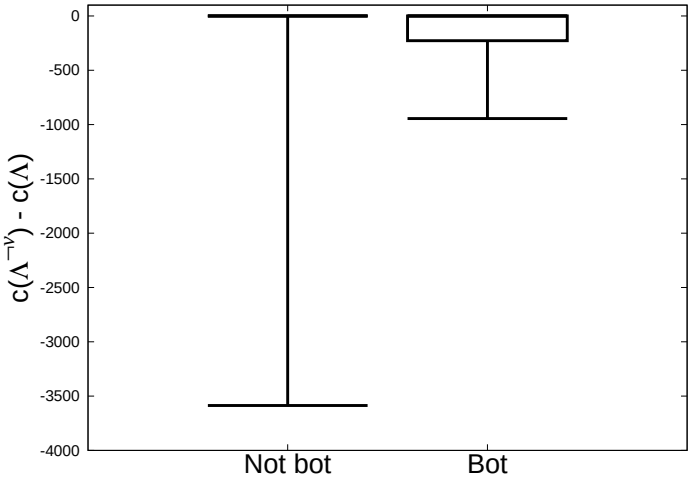
- Figure 2.6(c) represents $k(\Lambda^{\neg n}) - k(\Lambda)$, *i.e.*, the difference in the number of connected components due to eliminating n and all isolated neighbors, which can be positive, zero, or -1 if eliminating n and its isolated neighbors eliminates an entire connected component.

Each boxplot separates the cases in which n is a bot (right) or is not a bot (left). In each case, five horizontal lines from bottom to top mark the minimum, first quartile, median, third quartile and maximum values, with the lines for the first and third quartiles making a "box" that includes the median line. The five horizontal lines and the box are evident, *e.g.*, in the "bot" boxplot in Figure 2.6(c). However, because some horizontal lines are on top of one another in other boxplots, the five lines or the box is not evident in all cases.

This figure shows a strong dichotomy between the two graph parameters used for detection (graph size and largest component size) and the number of components. As shown in Figures 2.6(a) and 2.6(b), the impact of eliminating bots and the impact of eliminating other vertices largely overlap, for either graph size or largest component size. In comparison, eliminating bots has a radically different effect on the number of components, as shown in Figure 2.6(c): when a non-bot vertex is eliminated, the number of components increases a small amount, or sometimes decreases. In contrast, when a bot is eliminated, the number of components *increases* strongly.

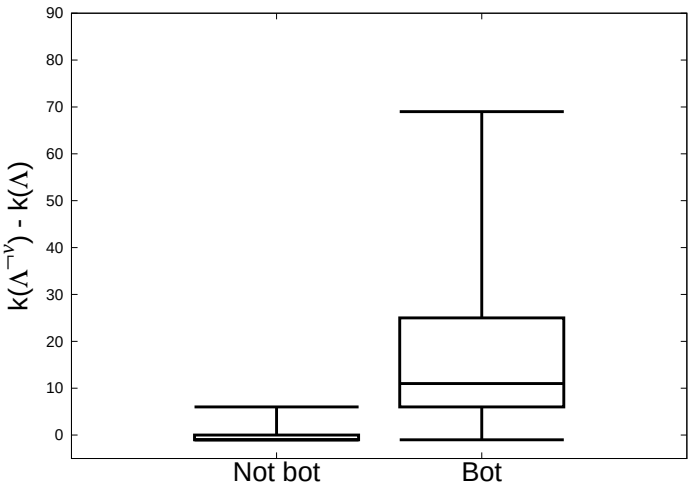


(a) $v(\Lambda^{\neg n}) - v(\Lambda)$



(b) $c(\Lambda^{\neg n}) - c(\Lambda)$

Figure 2.6: Effects of eliminating high degree vertices n from FTP attack traffic logs Λ.



(c) $k(\Lambda^{\neg n}) - k(\Lambda)$

Figure 2.6 continued.

Also of note is that the change in the total number of components (Figure 2.6(c)) is relatively small, and small enough to add little power for attack *detection*. For example, if we were to define a random variable $\mathcal{K}^{\mathsf{dur}}_{\mathsf{pm}}$ analogous to $\mathcal{V}^{\mathsf{dur}}_{\mathsf{pm}}$ and $\mathcal{C}^{\mathsf{dur}}_{\mathsf{pm}}$, and then formulate a worm detection rule analogous to (2.4) and (2.5) for component count—*i.e.*, raise an alarm for log file Λ_π where $\pi \in \mathsf{pm}$ had duration dur, if $k(\Lambda_\pi) > \mu_K^{\mathsf{pm},\mathsf{dur}} + \gamma\ \sigma_K^{\mathsf{pm},\mathsf{dur}}$—then roughly 80% of our hit-list attacks within FTP would go undetected by this check. This is because of the large standard deviation of this measure: $\sigma_K^{\mathsf{pm},60\mathrm{s}} \approx 12.5$.

Despite the fact that the number of components does not offer additional power for attack detection, Figure 2.6(c) suggests that removing a high-degree vertex n from a graph $\mathfrak{G}(\Lambda)$ on which an alarm has been raised, and checking the number of connected components that result, can provide an

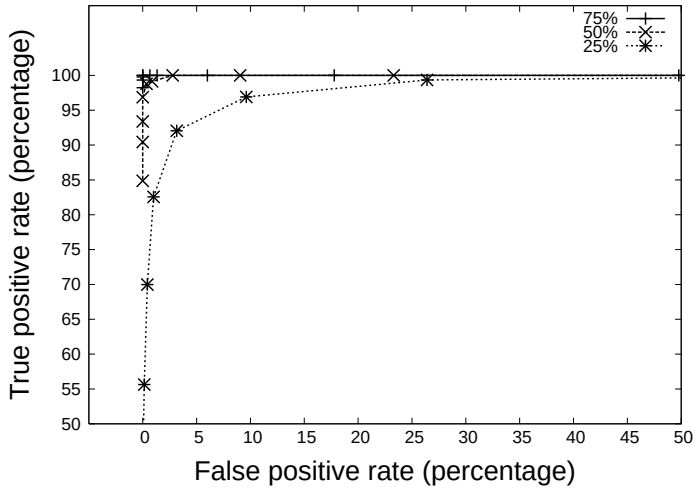
effective test to determine whether n is a bot. More specifically, we define the following bot identification test:

$$\mathsf{isbot}_{\Lambda,\theta}(n) = \begin{cases} 1 & \text{if } k(\Lambda^{\neg n}) - k(\Lambda) > \theta \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

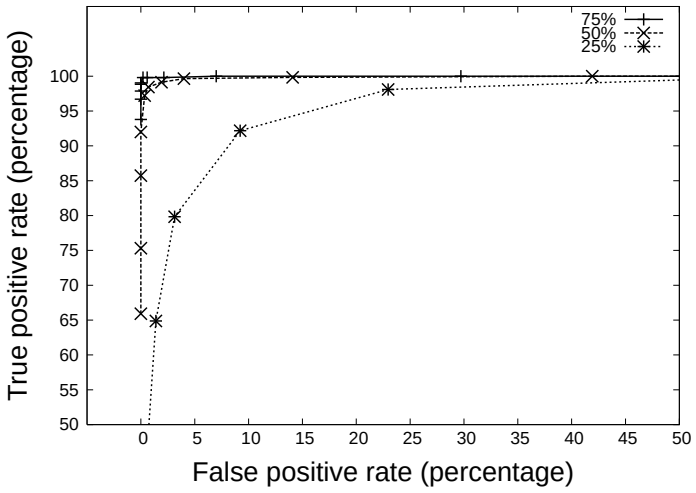
We characterize the quality of this test using ROC curves. Each curve in Figure 2.7 is a plot of true positive (*i.e.*, bot identification) rate on the y-axis versus false positive rate on the x-axis for one of the protocols we consider and for the simulated hit-list worm attacks discussed in § 2.4 that yielded a true alarm with $|\mathsf{bots}| = 5$ (the hardest case in which to find the bots) and $\mathsf{hitListPerc} \in \{25\%, 50\%, 75\%\}$. Each point on a curve shows the true and false positive rates for a particular setting of θ. More specifically, if $\mathsf{hidegree} \subseteq \mathcal{V}(\Lambda)$ is a set of highest-degree vertices in $\mathfrak{G}(\Lambda)$, and if $\mathsf{hidegreebots} \subseteq \mathsf{hidegree}$ denotes the bots in $\mathsf{hidegree}$, then any point in Figure 2.7 is defined by

$$\begin{aligned} \text{true positive rate} &= \frac{\sum_{n \in \mathsf{hidegreebots}} \mathsf{isbot}_{\Lambda,\theta}(n)}{|\mathsf{hidegreebots}|} \\ \text{false positive rate} &= \frac{\sum_{n \in \mathsf{hidegree} \setminus \mathsf{hidegreebots}} \mathsf{isbot}_{\Lambda,\theta}(n)}{|\mathsf{hidegree} \setminus \mathsf{hidegreebots}|} \end{aligned}$$

As Figure 2.7 shows, a more aggressive worm (*i.e.*, as $\mathsf{hitListPerc}$ grows) exposes its bots with a greater degree of accuracy in this test, not surprisingly, and the absolute detection accuracy for the most aggressive worms we consider is very good for HTTP, SMTP and FTP. Moreover, while the curves in Figure 2.7 were calculated with $|\mathsf{hidegree}| = 10$, we have found

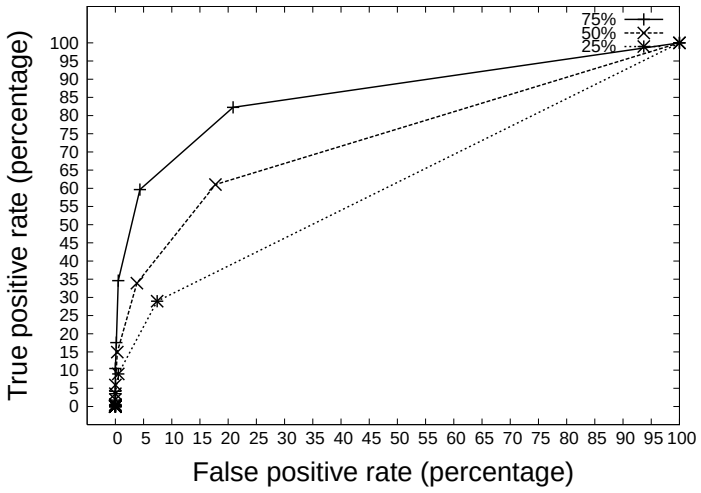


(a) HTTP

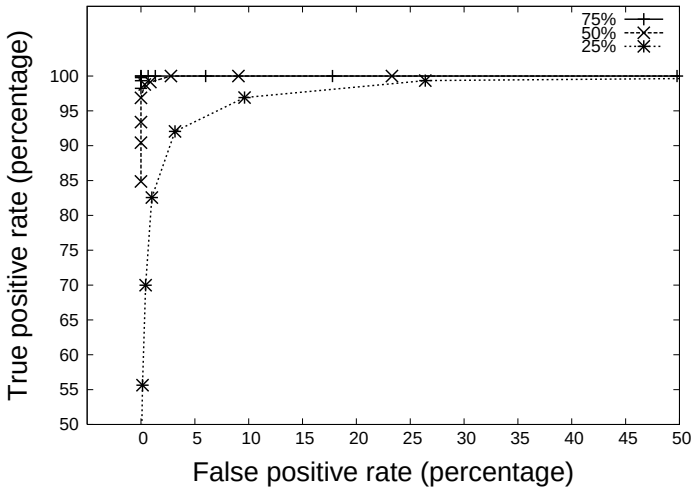


(b) SMTP

Figure 2.7: Attacker identification accuracy of (2.8); $\mathsf{hitListPerc} \in \{25\%, 50\%, 75\%\}$, $|\mathsf{hidegree}| = 10$, $|\mathsf{bots}| = 5$.



(c) Oracle



(d) FTP

Figure 2.7 continued.

that the accuracy is very robust to increasing |hidegree| as high as 100. As such, when identifying bots, it does not appear important to the accuracy of the test that the investigator first accurately estimate the number of bots involved in the attack. We are more thoroughly exploring the sensitivity of (2.8) to |hidegree| in ongoing work, however.

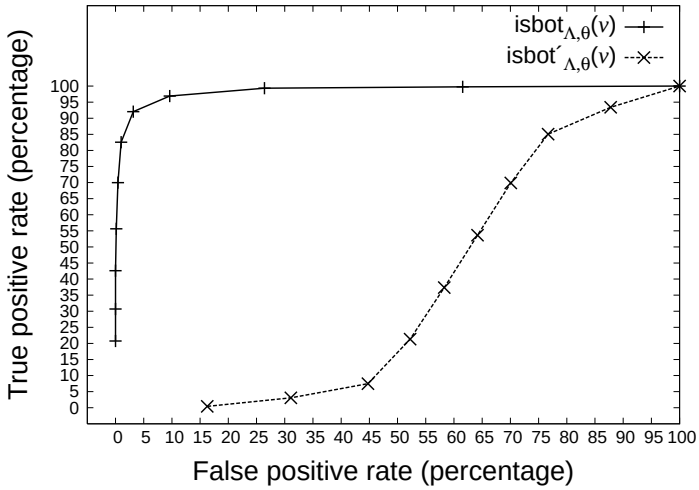
Because we evaluate (2.8) on high-degree vertices in order to find bots, a natural question is whether degree in $\mathfrak{G}(\Lambda)$ alone could be used to identify bots with similar accuracy, an idea similar to those used by numerous detectors that count the number of destinations to which a host connects (*e.g.*, [64, 52]). To shed light on this question, we consider an alternative bot identification predicate, namely

$$\mathsf{isbot}'_{\Lambda,\theta}(n) = \begin{cases} 1 & \text{if } \mathsf{degree}_{\Lambda}(n) > \theta \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

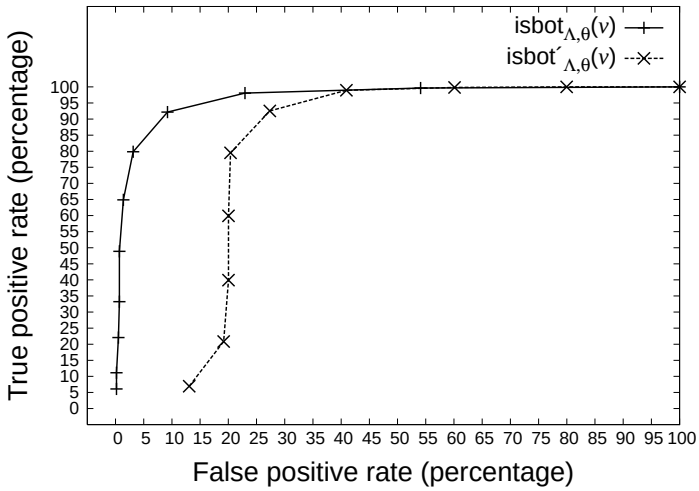
where $\mathsf{degree}_{\Lambda}(n)$ is the degree of n in $\mathfrak{G}(\Lambda)$, and compare this test to (2.8) in Figure 2.8. Figure 2.8 is a ROC curve and interpreted in the same fashion as Figure 2.7. As this figure shows, using (2.9) offers much less accurate results in some circumstances, lending support to the notion that our proposal (2.8) for bot identification is more effective than this alternative.

2.6 Implementation

Any worm detection system must be efficient to keep up with the high pace of flows observed in some protocols. A strength of our detection approach based on conditions (2.4) and (2.5) in § 2.3 is that it admits very efficient im-



(a) HTTP



(b) SMTP

Figure 2.8: Accuracy of (2.8) versus (2.9); $\mathsf{hitListPerc} = 25\%$, $|\mathsf{hidegree}| = 10$, $|\mathsf{bots}| = 5$.

plementation by well-known *union-find* algorithms [20]. Such an algorithm maintains a collection of disjoint sets, and supports three types of operations on that collection: a `makeset` operation creates a new singleton set containing its argument; a `find` operation locates the set containing its argument; and a `union` operation merges the two sets named in its arguments into one set. The size of each set in the collection can be maintained easily because each set in the collection is disjoint: a new set created by `makeset` has size one, and the set resulting from a `union` is of size the sum of the sizes of the merged sets.

The implementation of a worm detection system using a union-find algorithm is straightforward: for each $\lambda \in \Lambda$, the sets containing $\lambda.\mathsf{sip}$ and $\lambda.\mathsf{dip}$ are located by `find` operations (or created via `makeset` if the address has not yet been observed in Λ), and if these sets are distinct, they are merged by a `union` operation. $k(\Lambda)$ is the total number of sets, $c(\Lambda)$ is simply the size of the largest set, $v\Lambda$ is the sum of the sizes of the sets.

The efficiency of this implementation derives from the use of classic techniques (see [20]). A famous result by Tarjan (see [63]) shows that with these techniques, a log file Λ can be processed in time $O(|\Lambda|\alpha(v\Lambda))$, where $\alpha(\cdot)$ is the inverse of Ackermann's function $A(\cdot)$, *i.e.*, $\alpha(n) = \arg\min_k : A(k) \geq n$. Due to the rapid growth of $A(k)$ as a function of k (see [1, 63]), $\alpha(n) \leq 5$ for any practical value of $v(\Lambda)$. So, practically speaking, this algorithm enables the processing of flows with computation only a small constant per flow. Perhaps as importantly, this can be achieved in space $O(v(\Lambda))$. In contrast, accurately tracking the number of unique destinations to which a vertex connects—a component of several other worm detection systems

(*e.g.*, [64, 52])—requires $\Omega(|\mathfrak{E}(\Lambda)|)$ space, a much more significant cost for large networks. Hence our approach is strikingly efficient while also being an effective detection technique.

Once an alarm is raised for a graph $\mathfrak{G}(\Lambda) = \langle \mathfrak{V}(\Lambda), \mathfrak{E}(\Lambda) \rangle$ due to it violating condition (2.4) or (2.5), identifying the bots via the technique of § 2.5 requires that we find the high-degree vertices in $\mathfrak{V}(\Lambda)$, *i.e.*, the vertices that have the most unique neighbors. To our knowledge, the most efficient method to do this amounts to simply building the graph explicitly and counting each vertex's neighbors, which does involve additional overhead, namely $O(|\mathfrak{E}(\Lambda)|)$ space and $O(|\Lambda| \log(|\mathfrak{E}(\Lambda)|))$ time in the best algorithm of which we are aware. However, this additional cost must be incurred only after a detection and so can proceed in parallel with other reactive measures, presumably in a somewhat less time-critical fashion or on a more resource-rich platform than the detection itself.

2.7 Conclusion

In this chapter, we have introduced a novel form of network monitoring technique based on protocol graphs. We have demonstrated using logs collected from a very large intercontinental network that the graph and largest component sizes of protocol graphs for representative protocols are stable over time (§ 2.3.1). We have used this observation to postulate tests to detect hit-list worms, and showed how these tests can be tuned to limit false alarms to any desired level (§ 2.3.2). We have also shown that our tests are an effective approach to detecting a range of hit-list attacks (§ 2.4).

We have additionally examined the problem of identifying the attacker's bots once a detection occurs (§ 2.5). We demonstrated that examining the change in the number of connected components caused by removing a vertex from the graph can be an accurate indicator of whether this vertex represents a bot. We also showed that this indicator is substantially more accurate than examining merely vertex degrees.

Finally, we examined algorithms for implementing both hit-list worm detection and bot identification using our techniques (§ 2.6). We found that hit-list worm detection, in particular, can be implemented using more efficient algorithms than many other worm detection approaches, using classic union-find algorithms. For networks of the size we have considered here, such efficiencies are not merely of theoretical interest, but can make the difference between what is practical and what is not. Our bot identification algorithms are of similar performance complexity to prior techniques, but need not be executed on the critical path of detection.

As a preliminary work in this area, we have focused on a limited number of properties of protocol graphs. However, we believe that other properties of protocol graphs can be useful, as well, and are exploring this avenue of research in ongoing work.

Chapter 3

Training Anomaly Detectors

While scanners and bot harvesters can evade detection by limiting their behavior [42], sometimes they don't have to bother with subtlety. Automated tools used to exploit vulnerabilities on well-known applications are widely disseminated, resulting in a constant stream of crude scanning and infiltration attempts that are largely aimed at nonexistent targets. The presence of these constant clumsy attacks means that anomaly detection systems face a bootstrap problem: before they can build an accurate model of normalcy[1], they need an effective means to filter out hostility.

In this chapter, we demonstrate a method for recovering the normal activity of a network in the presence of constant attacks. We use this method of *attack reduction* to eliminate the noise caused by common high-failure attacks, and are able to produce a model of normal system activity that is

[1]Throughout this chapter, by "normal" behavior we mean behavior that is both normal and legitimate. That is, we do not intend to include these constant, clumsy attacks in what we describe as the "normal" traffic, and in fact the point of this chapter is to eliminate these simplistic attacks to create a model of normal and legitimate traffic.

far more precise than one derived from raw data. To illustrate the utility of our approach, we show that a graph-based anomaly detection method that counts the number of hosts observed over time [13], when trained on raw SSH data observed on a network we monitor, resulted in an anomaly detection threshold of 91,000 hosts in a system with 15,000 known servers. As such, without applying our techniques, the anomaly detection system would fail to detect, say, a distributed denial-of-service from an entire /16. Using our attack reduction approach, the model of normalcy can be tuned far more precisely, to reduce the detection threshold to only 370 hosts.

Our method combines statistical and heuristic filtering to recover periods of normal activity. This approach leads to two distinct advantages outside of support for the statistical learning system we have developed. First, since the attack reduction methodology identifies periods of normal activity, traffic records during those normal periods can be used to train other forms of IDS, such as Jung *et al.*'s TRW [26]. In addition, using our statistical approach, we are able to establish the training time required by our anomaly detection system and other approaches that rely on parametrized distributions. In our case, we demonstrate that we expect to achieve a 95% accuracy within 5 hours of wall clock time, including attacks.

To illustrate the noisy and clumsy attacks that we seek to filter out, Figure 3.1 plots the degree (number of servers contacted) for the first, second and third-highest degree SSH clients observed on our monitored network in 30s periods beginning on September 10th, 2007. Each plot in this figure is a time-series blot showing the degree of the corresponding client on the y-axis and the time (GMT) on the x-axis. As this figure shows, the highest-degree

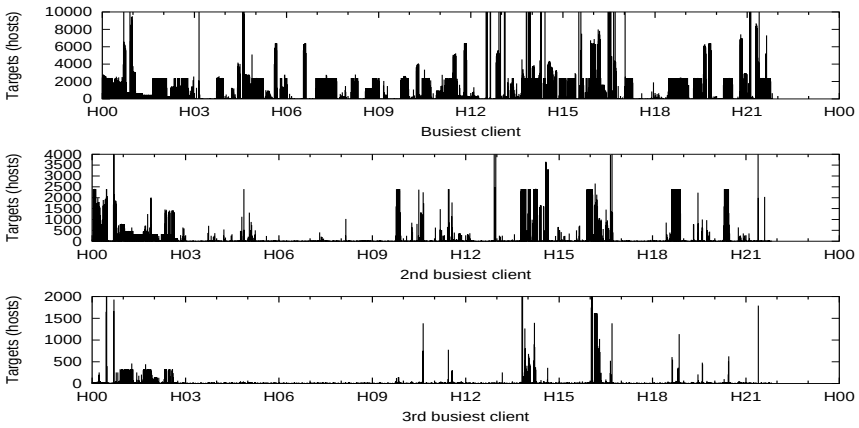


Figure 3.1: Communication of the three highest-degree SSH clients per thirty second period on the monitored network. As the figure shows, multiple clients appear to be communicating with hundreds or thousands of servers at a time, a likely indicator of scanning or failed harvesting.

clients regularly contact over 2,000 targets per period during this time; in the most extreme cases, an individual client contacted over 250,000 distinct targets. The feasibility of a single client opening thousands of simultaneous SSH connections in half a minute notwithstanding, the observed network has less than 15,000 SSH servers.[2]

These results are consistent over the data examined for this chapter: a two week log of SSH traffic in which we are reasonably confident that some form of attack occurs in over 80% of the 30s periods observed. The majority of these attacks are scans and automated botnet harvesting attempts, how-

[2]Servers were identified by looking for hosts that had payload-bearing responses from port 22 during the observation period.

ever since the failure rate for these attacks is so inordinately high, we cannot actually determine what an attacker was doing in the majority of cases. We will refer to these attacks as *High Connection Failure Attacks* (HCFAs) to emphasize this.

The key properties of HCFAs, as seen in the SSH data, is that they are persistent, continuous, massive and almost entirely failures. SSH traffic has been persistently scanned for several years by script kiddies and bots, generally by using brute-force password attacks[3]. The level of hostile traffic is so pervasive that various security publications recommend moving the SSH listener port in order to evade scanning[4].

Anomaly detection systems generally classify attacks as deviations from a model of normal behavior generated from historical data [17, 13, 58]. While anomaly detectors have the potential to detect subtle attacks quickly and effectively, these systems must train on clean data in order to produce an accurate model [23]. As shown above, however, for protocols such as SSH, clean data is rare – HCFAs are, on a flow by flow basis, the most common form of traffic. If an anomaly detection system trains on such data, it will correctly and uselessly model normal activity with the HCFAs. A statistical model based on this traffic will have an inordinately wide variance that will allow subtle attacks to hide in the noise. Raising alerts on HCFAs is also counterproductive – they are so common and so rarely successful, that validating the traffic will distract operators from far more serious threats, if

[3] "Protecting Linux against automated attackers", Ryan Twomey, `http://www.linux.com/articles/48138`, last fetched January 18th, 2008.

[4] "Analyzing Malicious SSH Login Attempts", Christian Seifert, `http://www.securityfocus.com/infocus/1876`, last fetched January 18th, 2008.

those threats are even noticed.

Our attack reduction process assumes that the IDS models normal network activity using a parametrized distribution. In this chapter we use the Gaussian distribution $\mathcal{N}(\mu_{\mathcal{V}}, \sigma_{\mathcal{V}})$ as the model, which we will refer to as the *state model* of the system. To extract this state model from the traffic, attack reduction utilizes three steps. The first, *log-level* filtering, examines records for evidence of *significance*. A significant record is one which may have been received and processed by the targeted application. The majority of HCFA flows will be insignificant since they will communicate with nonexistent targets. This process ensures that the filtered data set is manageably small without eliminating important normal interactions, but the simple heuristics used by the log-level filter mean that it cannot filter all HCFAs.

We therefore complement log-level filtering with a second, *state-level* filtering method. In this work, we use a filter based on the Shapiro-Wilk normality test to identify and eliminate outlier states [55]. The Shapiro-Wilk test statistic is an estimate of how closely a set of observations matches a Gaussian distribution. State-level filtering assumes that if an HCFA's traffic records pass log-level filtering, the HCFA's aggregate behavior will still be an observable outlier. More specifically, an HCFA will only be able to increase the observed state and it will do so in an obvious manner, and the Shapiro-Wilk based filter will identify this deviation.

The third and final step of attack reduction is *impact evaluation*. The previous filtering steps identify outliers in their datasets and pass sources of those outliers, *anomalous addresses*, to an impact evaluator to determine if

the addresses have done anything to warrant further investigation. Since the majority of HCFAs are failures, the evaluation step differentiates between *actionable* and *inactionable* activities. An actionable address is one that has engaged in action that merits operator response, such as successfully communicating with a target; inactionable addresses have done nothing to merit further investigation yet.

This approach will therefore be applicable for the constant HCFAs seen in our SSH data. While the majority of HCFAs are scans and failed harvesting, any attack where the attacker is willing to generate a large number of failed TCP connections will be filtered by this model.

The remainder of this chapter is structured as follows: §3.1 is a survey of previous work in anomaly detection and the problem of error reduction in IDS. §3.2 describes the architecture of our system, the source data and the problems associated with log-level filtering. §3.3 describes our methods for state-level filtering. §3.4 examines the impact of our methods on SSH data. §3.5 discusses impact evaluation and notification, while §3.6 discusses methods that an attacker can use to evade detection or control the system. Finally, §3.7 concludes our work.

3.1 Related Work

While our work is within the domain of anomaly detection, we emphasize that the focus of this chapter is on IDS training and use in the presence of persistent interference from HCFAs. Since HCFAs are likely to be scans or failed harvesting attempts, our work is strongly influenced by existing

anomaly and scan detection methods, with the notable difference that we assume attacks are the norm within our data.

The pollution from HCFAs has been studied by multiple researchers. Moore *et al.* [40] developed the standard method for examining aggregate behavior by studying *dark space*. Pang *et al.* [43] have discussed the characteristics of the random traffic on the Internet as "background radiation". More directly relevant to our work is the approach of Garg *et al.* [21] for managing the data from this background radiation in NIDS. Our approach differs from that of Garg *et al.* both in application (our method is anomaly-based, whereas theirs is payload-dependent) and implementation, as their work is a hardware-driven solution and ours is more general.

The problem addressed by eliminating HCFAs is best understood by addressing the types of learning done by anomaly detection systems. For the purposes of this chapter, we break down these systems into three broad categories: *threshold* systems, which include GrIDS [58] and our graph-based method [13]; *data mining* systems, such as MINDS [17] and LERAD [36]; and *map-based* systems, most notably TRW [26]. Each of these systems relies on past history and is consequently vulnerable to HCFAs.

Threshold systems detect anomalous behavior by looking for a measurable change in traffic levels. Examples of these systems include the original IDES [35] and EMERALD [46], and systems for identifying and quarantining worms [67, 25, 69]. These systems all identify anomalies by looking for a change in behavior that exceeds a threshold estimate. This threshold may be generated by an automatic process, as is the case with Soule *et al.* [56], or provided by a human expert, as is the case with GRiDS [58] and

MISSILE [22]. When these thresholds are set automatically (a process we seek to enable here), such systems have assumed that attacks are rare (often implicitly). Here we focus on situations in which this is not the case.

Data mining systems apply undirected learning and clustering techniques to traffic data [6, 17, 36, 33, 24, 65]. These systems group together traffic by log features in order to differentiate attacks from normal activity. As before, they expect that traffic will generally be attack-free (in particular, LERAD [36] is specifically designed to detect "rare time series events"). An exception to this is Barbara *et al.* [7] who address the problem of bootstrapping a data-mining IDS. However, the data set they use, the 1999 Lincoln Labs data [34] is synthetic and does not contain attacks with remotely the frequency of our SSH data.

Mapping systems rely on an inventory of network hosts to identify aberrations. The most well-known of these systems is Jung *et al.*'s TRW method of scan detection [26] and Shankar and Paxson's Active Mapping tool for Bro [53]. These systems have a varying degree of vulnerability to HCFAs based primarily on the mapping method used. Active approaches will not be affected by HCFAs but on larger networks, passive mapping approaches become increasingly attractive [68]. Since passive mapping systems must also deal with HCFAs communicating with nonexistent hosts, HCFAs can become a significant challenge.

Our system is not intended to supplant these IDS but rather supports them by providing a method to identify normal activity. Consequently, the results generated by our attack reduction methodology can also be used as a source of labeled normal data for data mining systems or to build the maps

used by mapping systems.

3.2 System Architecture and Log-level Filtering

The majority of connections in HCFAs fail to reach real targets and consequently result in a large number of flow records that are easily distinguished from normal traffic. Log-level filtering is the process by which these *insignificant* flows are distinguished from *significant* flows which may have actually communicated with a server process. Log-level filtering is stateless and inexpensive in order to manage the expected volume of flow records.

This section describes the system architecture and the process of log-level filtering. It is divided into three sections: §3.2.1 describes our source data and the notation we use to describe it. §3.2.2 provides an information flow diagram for the system and describes the expected format of log data and state data. §3.2.3 describes the log filtering process and shows the impact of log-level filtering on representative SSH data.

3.2.1 Source Data

The source data used in this chapter consists of NetFlow V5 records[5] generated by internal routers in a large (in excess of 16 million distinct IP addresses) network. We use a two-week NetFlow trace of SSH traffic collected over this network during the period of September 10th-24th, 2007. The routers record both internal and cross-border traffic.

[5]CISCO Systems, "CISCO IOS Netflow Datasheet", `http://www.cisco.com/en/US/products/ps6601/products_data_sheet0900aecd80173f71.html`, last fetched October 8th, 2007.

NetFlow records approximate TCP sessions by grouping packets into *flows*, sequences of identically addressed packets that occur within a timeout of each other [11]. NetFlow records contain size and timing information, but no payload. We treat NetFlow records as tuples of the form (clntip, srvip, direction, stime, bytes, packets, syn, ack). These elements are a modified subset of the fields available in CISCO NetFlow. When referring to a single flow record λ, we will reference constituent elements of a single flow record via "dot" notation (*e.g.*, λ.srvip or λ.packets).

We classify traffic by port number; any flow which is sent to or from TCP port 22 is labeled an SSH flow. Because we examine only SSH traffic, port numbers are not germane to our analyses. Instead, we use the port number to define the srvip, clntip and direction fields. srvip is the address that uses port 22, clntip is the address which used an ephemeral port[6], and direction indicates which direction the recorded traffic flowed (*i.e.*, traffic in the flow was from srvip to clntip or vice versa).

syn and ack refer to the TCP SYN and ACK flags respectively. CISCO's NetFlow implementation contains a field providing a logical OR of all TCP flags observed in a TCP flow. However, this field is not available under all IOS implementations. We therefore limit the use of flag data to exploratory analysis and scan validation; the *implementations* of log- and state-level filtering in this chapter do not use these fields.

The source data for state filters are *observations* of a *state function* constructed from flow records that have already gone through the log filtering

[6]We constrain the flows used in this chapter to flows which used an ephemeral port between 1024 and 5000.

process. We group flow records into distinct *log files* $\Lambda = \{\lambda_1 \dots \lambda_n\}$. The contents of any log file are determined by the stime field. In this chapter, each unique log file contains all records whose stime is within a distinct 30s period, and the records within the log file are ordered by stime.

In chapter 2, we demonstrated that the graph size and largest component size of graph representations of several major protocols could be satisfactorily modeled using a Gaussian distribution. For this chapter we rely exclusively on graph size for brevity and generality – graph size is equivalent to the the total host count for a protocol, and consequently the state model does not rely on any special features of our previous work.

An *observation* of the graph size, $v(\Lambda)$, is the graph size for a particular log file Λ. Recall that these log files cover unique 30s periods within the source data, so the graph size is the size of the observed protocol graph for SSH within a 30s observation period. Recall that $v(\Lambda)$ is defined in Equation 2.1 as the cardinality of the set of all IP addresses (client and server) in Λ.

State-level data is a vector of graph size observations $V = (v_1 \dots v_k)$ where each v_i is a $v(\Lambda)$ value. V's observations are ordered by increasing magnitude rather than chronological order, so v_{10} may have been observed some time after v_{11}. The source Λ^{sig} data is comprised of 39,600 distinct log files, each covering a 30s period beginning on September 10th 2007 and continuing until September 24th. 720 log files were discarded from the data set due to sensory or collector failures; if a single log file in an hour had an observed failure, we discarded the entire hour. We assume the graph sizes are the result of a random process modeled as $\mathcal{N}(\mu_{\mathcal{V}}, \sigma_{\mathcal{V}})$ where $\mu_{\mathcal{V}}$ is the

mean and $\sigma_{\mathcal{V}}$ is the standard deviation of the random variable $\mathcal{V}$.

When referring to the mean and standard deviation of the vector V, we use the same notation as in Chapter 2: μ_V refers to sample mean of the vector V, and σ_V the observed standard deviation of the vector V. Recall that in Chapter 2 we also adapted a specific notation for the log file duration and the interval during which the log file was collected. Our observed SSH data does not show the time dependency of the other data sets, so we do not use the interval notation. Similarly, since we now used a fixed duration (30s), we no longer use the duration notation.

3.2.2 System Architecture

Figure 3.2 is an information flow diagram that describes how log-level and state-level data are handled by the attack reduction system. This diagram describes four processes: log-level filtering, state-level filtering, normal operation, and impact evaluation/notification. Recall that a single log file contains all the flow records observed during a particular 30s period – the information flow in Figure 3.2 takes place within this time constraint (*i.e.*, all processing decisions take place within the 30s after Λ is reported). Numbers in the following text refer to the corresponding arrows in Figure 3.2.

The first phase of the process is log-level filtering, during which data are collected from multiple *sensors* located throughout the monitored network. These sensors generate NetFlow records which are then passed (1) to a *log filter* that partitions the log file Λ into two log files: Λ^{sig}, a log file of significant flow records, and Λ^{insig}, a log file of insignificant flow records. We approximate significance using the number of packets in a flow, a process

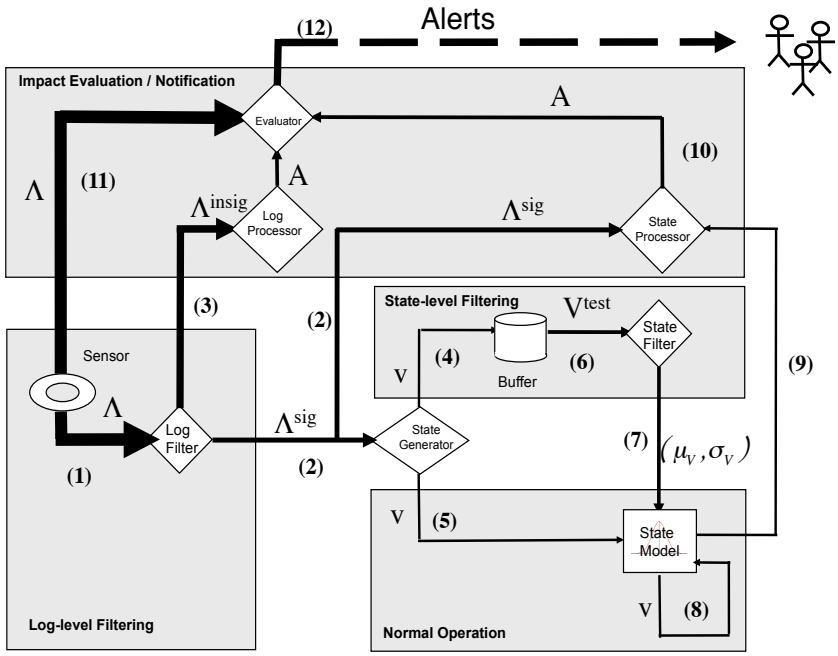


Figure 3.2: Information flow diagram for the attack reduction system.

described in §3.2.3. Λ^{insig} is passed (3) to the *log processor*, which produces a list of the busiest hosts for further examination by the evaluator.

After log-level filtering, Λ^{sig} is passed (2) to a *state generator* which realizes the observation $v(\Lambda^{\text{sig}})$. This is then processed either by the state-level filter (4) or normal operations (5). Where observations are processed is a function of whether the system is in bootstrap phase (during which time, the data is processed by the state-level filter) or operating normally.

During the bootstrap phase, the system has insufficient information to judge whether or not any particular observation is an outlier. Consequently,

during this phase, the state data is passed to a buffer to create a *test vector*, V^{test}. When $|V^{\text{test}}|$ exceeds a *sample limit*, z, it is run through the state-level filter (6) and outliers are removed. If the filtered sample still exceeds the threshold, then the filter generates parameters for a model of normal traffic: $(\bar{v}, \sigma_{\mathcal{V}})$. The filter then passes those parameters to the *state model* (7) and the system shifts into normal operations. If the filtered test vector is smaller than the sample limit, additional data is collected (4) and the process repeated (6) until the filtered vector reaches sufficient size.

During normal operation, v observations are passed directly to the state model (5). Recall that the state model used in this chapter is a simple statistically derived threshold: $v(\Lambda^{\text{sig}})$ is calculated and compared to the model. If $v(\Lambda^{\text{sig}})$ exceeds the anomaly threshold, then an alert is passed to the *state processor* (9). During normal operation, the state model can update by feeding normal observations back into the model (8).

Anomaly processors examine anomalous addresses to determine whether they are worth deeper investigation by the *evaluator* for validation and notification. The method used by each processor is dependent on the source data (Λ^{sig} or Λ^{insig}). The log processor receives Λ^{insig} (3) data and consequently must process much higher volumes than the state processor. (In our present implementation the log processor simply returns the top-5 highest degree addresses; see Section 3.5.) The state processor receives the much smaller Λ^{sig} (2) and will use whatever method is appropriate for its state model. Our implementation uses a graph-based method described in our previous work [13]: it returns a list of those vertices whose removal substantially alters the shape of the graph (see Section 3.5).

Once each processor generates a list of anomalous addresses, this list is passed to the evaluator. The evaluator examines the anomalous addresses for evidence of *actionability*. We expect the majority of scans to be, if not harmless, then pointless – they include residue from ancient worms (a phenomenon noted by Yegneswaran *et al.* [70]) and blind bot scanning. As such, an anomaly detection system which raises an alarm for every scan will be counterproductive; unless a HCFA is actionable, it is noted in other collection systems but otherwise ignored.

3.2.3 Log-level Filtering

During log-level filtering, we apply simple heuristics to individual flow records in order to identify the majority of HCFA traffic and remove it from further analysis. The criterion for keeping or removing a flow record is a property we term *significance*. A flow record is significant if the traffic it describes constitutes communication with the targeted host's process rather than just the TCP stack; *e.g.*, an SSH flow record is significant if the target's SSH process receives payload. Otherwise, the flow record is insignificant.

We approximate significance by using properties of the TCP stack. Since TCP implements a state machine on top of packet switching protocols, a correct TCP session requires 3 packets of synchronization in addition to any packets for payload [59]. Flows which contain three packets or fewer are likely to be insignificant. While it is possible to have a significant flow below this limit (such as keep-alives), we expect that they will represent a small proportion of the total number of short flows. Of more concern is that, due to the timeout-based aggregation mechanism used by NetFlow, multiple

SYN packets to the same host will be grouped together as a single flow.

Potential alternative significance criteria include TCP flag combinations and payload. We have elected not to use flags because scanning applications such as `nmap`[7] specifically include packet-crafting options that use eccentric flag combinations. We elect not to use payload because the prevalence of SYN packets also makes estimating the actual payload of a flow problematic: SYN packets usually contain nontrivial stack-dependent TCP options [45], meaning that a "zero payload" TCP flow may contain 12 bytes of payload per SYN packet. Alternatively, a flow record describing the ACK packets from a host receiving a file may have no recorded payload, but is still significant. Payload estimates are used for impact evaluation in §3.5, but at that time we are specifically looking for activity initiated by a single client.

We therefore label a flow λ as significant if $\lambda.\mathsf{packets} > 3$ and insignificant otherwise. Table 3.1 is a contingency table showing the breakdown of our source data traffic using this rule. As this table shows, the total data describes approximately 2.4 TB of traffic contained in approximately 230 million records, and that approximately 87% of the observed flows are insignificant but make up less than 8% of the observed traffic by volume.

Examining those sensors that provide flag data indicates that approximately 70% of insignificant flows have no ACK flag – an indication that the flow was part of a HCFA. Of the remaining insignificant flows, 21% have SYN, ACK and FIN flags raised. While this type of traffic is potentially normal, it is also characteristic of certain forms of SYN scans[8].

[7] `http://www.insecure.org`

[8] This particular scan can be implemented using the '-sS' option in nmap.

		λ.ack		
		0	1	Total
λ.packets	1-3	1.59e+08 (1.37e+10)	3.75e+08 (4.84e+09)	1.97e+08 (1.86e+11)
	4-∞	1.15e+07 (8.36e+11)	1.86e+07 (1.53e+12)	3.02e+07 (2.37e+12)
Total		1.71e+08 (8.50e+11)	5.61e+07 (1.54e+12)	2.27e+08 (2.39e+12)

Table 3.1: Breakdown of activity in recorded data set by flows and bytes (bytes in parentheses); source data is all SSH traffic from September 10th to 24th, 2007.

From these results, we conclude that insignificant flows are dominated by HCFAs. A more stringent criterion for significance, such as a higher number of packets, risks removing more normal traffic from the data.

Figure 3.3 shows the impact of removing insignificant flows from the data set. This figure plots total graph size for the periods between 12:00 and 14:00 GMT for September 4th, 2007. The x-axis of this plot is the time of the observation (GMT), and the y-axis is the value of $v(\Lambda^{\text{sig}})$ (the filled-in shape) or $v(\Lambda)$ (the outlined shape). As this figure shows, even with the removal of insignificant flows, it is likely that scanning still appears in the data.

3.3 Implementing a State-level Filter

In §3.2 we demonstrated that log-level filtering was a necessary but insufficient method for recovering a system's normal behavior. While log-level filtering can reduce the amount of data to a manageable level and remove the most obvious effects of HCFAs, aggressive log-level filtering raises a serious risk of eliminating normal traffic.

Recall from §3.2.1 that our hypothetical anomaly detection system tracks

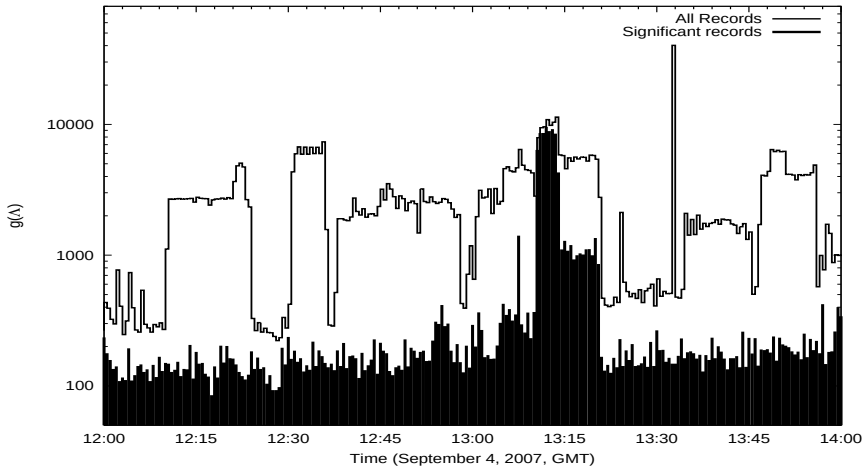


Figure 3.3: Contribution of significant and insignificant flows to traffic.

a single type of *observation* (in the case of this chapter, observed graph size). In this section we introduce a *state-level filter* which is used during the bootstrap phase of the IDS to identify and eliminate anomalies in training data.

State-level filtering applies the same clumsiness assumption that was used in log-level filtering. Specifically, we assume that the attacker's activities will increase the observed state function and regularly produce outliers due to the attacker's general lack of knowledge about the observed state function.

This section describes the construction of a state-level filter that uses the v value discussed before. We emphasize that this approach is not dependent on protocol graphs, only that the state function have a parametrizable distribution. This section is divided as follows: §3.3.1 describes the source

data, §3.3.2 process of state filtering, and §3.3.3 describes how much data is required to build a satisfactory model.

3.3.1 State Data

Figure 3.4 plots the frequency of $v(\Lambda^{\text{sig}})$ values for all Λ^{sig} in the source data. In this figure, the x-axis is the magnitude of $v(\Lambda^{\text{sig}})$ and the y-axis is the frequency with which that particular value was observed. As this figure shows, the distribution of $v(\Lambda^{\text{sig}})$ observations has a very long tail, but the majority of the observations (approximately 75%) can be modeled by a Gaussian distribution. This Gaussian fraction of the data is shown in the closeup in Figure 3.4. We hypothesize that the data set consists of two classes of periods: *normal* and *attack*. During a normal period, $v(\Lambda^{\text{sig}})$ is a function of the true population of the network and can be modeled by a Gaussian distribution. During an attack period, the $v(\Lambda^{\text{sig}})$ observation increases to the impact of HCFAs. Since HCFAs are caused by clumsy attackers, we further expect that the attacker cannot *decrease* $v(\Lambda^{\text{sig}})$, and consequently all observations above a certain threshold are likely to be attacks.

In order to test this, we sampled log files from the upper 40% of the Λ^{sig} set for scanning activity. Our sample consisted of 60 Λ^{sig} log files chosen randomly, 20 from each of three strata: $300 < v(\Lambda^{\text{sig}}) < 600$, $600 < v(\Lambda^{\text{sig}}) < 5000$ and $v(\Lambda^{\text{sig}}) > 5000$.

Within each log file, we examined the traffic for signs of HCFAs such as an unusually high-degree client, or a client which appeared only during the associated phenomenon. Of the 60 log files we examined, a single client dominated each log file, in each case communicating with at least 200 dis-

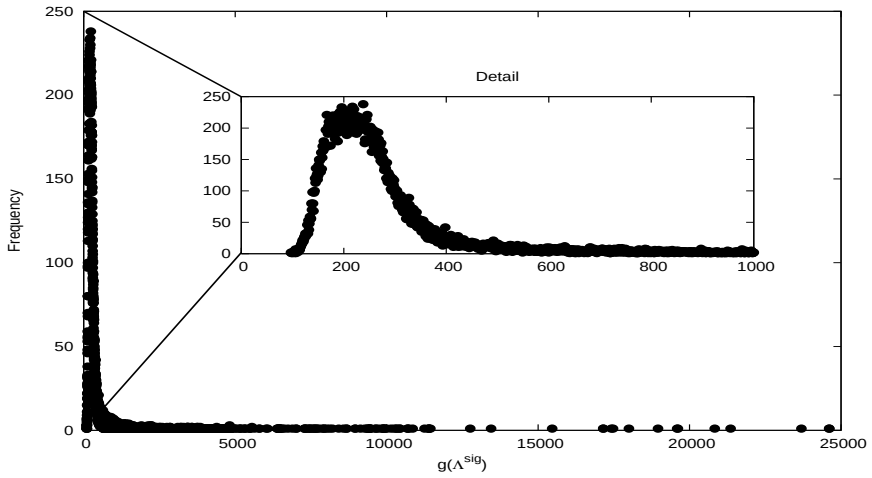


Figure 3.4: Histogram of total graph size with closeup on normally distributed minimal area; this figure supports our hypothesis that the majority of the high-level traffic are simply HCFAs.

tinct targets. While different clients dominated different log files, the set of clients was small: 31 distinct clients out of 60 log files. This result is due to the brazenness of the attackers; in certain cases, an attacker scanned continuously for several hours in a row. Using DNS, we checked client identity and verified that all of the clients were from outside the monitored network. By extracting full log files of the clients' activity over the observation period, we found no examples of payload-bearing interactions from these clients. From these results, we conclude that the clients were engaged in either scanning or the use of automated harvesting tools.

Given these results, we partition V, the vector of graph size observations, into two subsidiary vectors, V^{atk} and V^{normal}. The partitioning criterion is

$v(\Lambda^{\text{sig}})$: V^{atk} consists of all $v(\Lambda^{\text{sig}})$ observations where $v(\Lambda^{\text{sig}}) > 300$, V^{normal} consists of everything else. Out of the 39,600 observations in V, 10,101 or approximately 25.5% ended up in V^{atk}. As with V, V^{atk} and V^{normal} are ordered by increasing magnitude. We emphasize that this classification is intended for experimental validation and does not necessarily indicate that these observations actually contain HCFAs. During our experiments, we use the top 3000 values of V^{atk} in order to gain confidence that all of those observations are due to attacks.

3.3.2 Process

Assume V^{test} is a vector of graph size observations wherein the majority of observations are normal. A state-level filter is a process which examines V^{test} and returns one of two values: either an observed (μ_V, σ_V), or a requirement for further data.

For this chapter, we eliminate outliers by using the Shapiro-Wilk test [55]. The Shapiro-Wilk test specifically evaluates the similarity of an ordered vector of observations to a Gaussian distribution. In comparison to other statistical tests, the Shapiro-Wilk test can generate results with a wide variety of sample sizes (between 40 and 2,000 observations) and without advance knowledge of the modeled distribution (*i.e.*, the $\mu_{\mathcal{V}}$ and $\sigma_{\mathcal{V}}$ do not have to be supplied). The statistic generated by the Shapiro-Wilk test (W) is a value in the range of $[0, 1]$, with 1 representing a perfect fit to a Gaussian model.

We calculate the W-statistic, using a vector of observations $V^{\text{test}} = (v_1 \dots v_l)$ ordered by increasing magnitude and an equal size vector M that consists of order statistics for a Gaussian distribution of the form $(m_1 \dots m_l)$.

W is then formulated as:

$$W = \frac{\left(\sum_{i=1}^{l} a_i v_i\right)^2}{\sum_{i=1}^{l}(v_i - \mu_V)^2} \tag{3.1}$$

where

$$(a_1 \dots a_l) = \frac{M^T V^{-1}}{\sqrt{M^T V^{-1} V^{-1} M}} \tag{3.2}$$

and where V is the covariance matrix of M and μ_V is the sample mean of V.

To filter out HCFAs we use a state-level filter $\mathsf{shapfilt}(V^{\text{test}}, \theta_W)$. shapfilt takes two parameters: V^{test} and θ_W, which is the minimal W value we use to determine if a vector contains a Gaussian distribution. Recall that V^{test} is ordered by magnitude and that attackers cannot decrease $v(\Lambda^{\text{sig}})$. We therefore expect that there will be a pivot value below which $v(\Lambda^{\text{sig}})$ observations are normal and above which $v(\Lambda^{\text{sig}})$ observations are attack observations. shapfilt will return the index of that pivot.

shapfilt starts by calculating the W value for the complete V^{test}. If the resulting W exceeds θ_W, then shapfilt terminates and returns $|V^{\text{test}}|$ as its result. Otherwise it removes the largest value and repeats the process with the reduced vector. shapfilt will iteratively test every vector, progressively eliminating the larger observations until $W > \theta_W$ or all the observations are eliminated. The resulting index, z, is then checked against a minimum sample size. If z is too small (*i.e.*, there were too many attacks in the log data), then more observations will be requested and added to V^{test}. If z is

sufficiently large, then μ_V and σ_V are calculated from the reduced vector and passed to the state model.

3.3.3 State Filtering: Sample Size

Recall that shapfilt is intended for the initial training and configuration of an IDS. The longer the system remains in training, the more accurate the resulting model. However, while the system is training, it cannot react to attacks.

When shapfilt completes operation on a V^{test} vector, it returns its estimate of how many observations within that vector can be used for training. We can calculate the lower limit for this value by relying on our previous assumption that, in the absence of HCFAs, V^{test} would consist of observations following a Gaussian distribution and calculate the *margin of error*. For our model, the error is the difference between the true mean and the sample mean (*i.e.*, $|\mu_V - \mu_{\mathcal{V}}|$); the margin of error is the upper bound on that value.

Assume a random process modeled as $\mathcal{N}(\mu_{\mathcal{V}}, \sigma_{\mathcal{V}})$. We estimate $\mu_{\mathcal{V}}$ via the sample mean μ_V. The margin of error, E, is then written as:

$$E = Z_{\alpha/2}\frac{\sigma_{\mathcal{V}}}{\sqrt{z}} \tag{3.3}$$

$Z_{\alpha/2}$, the *critical value*, is a factor derived from α, the probability that the error is *greater* than the margin of error. Since the process is modeled with a Gaussian distribution, the critical value is expressed as the number of standard deviations to produce a $(1 - \alpha)$ probability of the error being less

than the margin. For example, 68% of all observations within 1 standard deviation of the mean of a Gaussian distribution. In this case, $\alpha = 0.32$ (that is, the probability of the error falling outside this range is 32%), and $Z_{\alpha/2} = 1$.

Given a known standard deviation, $\sigma_{\mathcal{V}}$, we can express z as follows:

$$z = \left(\frac{\sigma_{\mathcal{V}} Z_{\alpha/2}}{E} \right)^2 \tag{3.4}$$

We will limit the margin of error to $0.05\mu_{\mathcal{V}}$ (5% of the true mean). To achieve a 95% degree of confidence in the margin of error, $\alpha = 0.05$ and the corresponding critical value is 1.96. We can then rewrite Equation 3.4 to estimate that z should be at least:

$$z = 1536 \left(\frac{\sigma_{\mathcal{V}}}{\mu_{\mathcal{V}}} \right)^2 \tag{3.5}$$

This formula provides us with a method to estimate a lower limit on z, number of samples required. In Table 2.1 we encountered standard deviations which were anywhere from approximately one quarter to one twelfth of the mean. Given this, we will assume that our worst case scenario is where the standard deviation is half the mean. If so, then $z = 1536/4 = 384$. For convenience, we will round this value up to 400 observations, which equates to 3 hours and 20 minutes of 30s samples if there are no HCFAs.

This value assumes that normal traffic can be modeled via a Gaussian distribution. This may result in a longer clock time to gather those 400 observations based on seasonal phenomena, such as the business cycle. For example, in our original work, we divided the traffic into active and fallow

periods of 12 hours each. Under this model, each period would require a distinct 400 observations, changing the minimum time required to just under 7 hours. If a different model was calculated for every hour of the day, than the minimum time required would be slightly longer than 3 days.

HCFAs within the data set will also impact the training time. The system will not switch from training mode until it receives a sufficient amount of uncorrupted data, and will reject outlying scans.

3.4 Evaluation

We now evaluate the ability of state-level filtering to recover an accurate model of normal activity. Our evaluation method uses synthetic V^{test} vectors created by randomly selecting observations from $V^{\text{atk}}$and V^{normal}. Since HCFAs cannot *decrease* graph size, we expect that every V^{test} will have a pivot point: all observations before the pivot point will be normal and all observations after the pivot point will be HCFAs.

We measure the efficacy of the state filtering mechanism by comparing the true pivot point of V^{test} against an estimate generated using the Shapiro-Wilk test. This metric, the *pivot error*, is the difference between the pivot point as estimated by shapfilt and the true pivot point generated when the test vector is constructed. Intuitively, the preferred value for the pivot error is zero, which indicates that shapfilt was able to perfectly separate attack and normal events. However, given the choice of a negative or a positive pivot error, a negative pivot error is tolerable. We expect normal observations will be clustered around the mean, and consequently removing some of them will

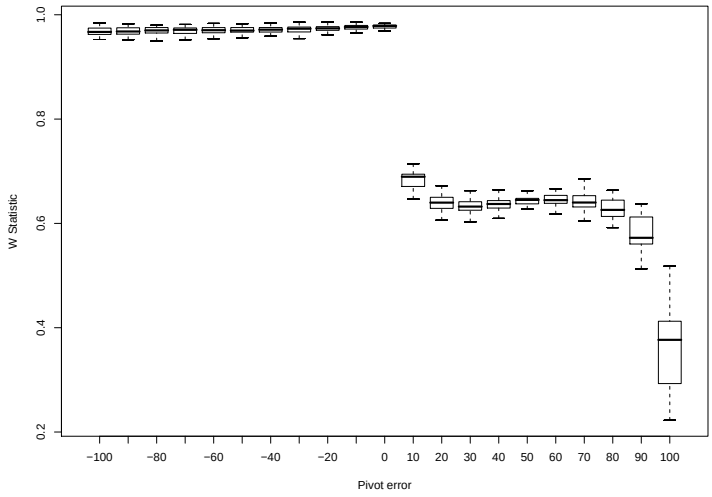


Figure 3.5: Impact of removing observations using shapfilt; note the decided increase in the W-statistic after removing all HCFAs.

have minimal effect on the resulting model. However, since HCFAs will be extreme outliers, including one will severely damage the model.

We use pivot error to calculate the efficacy of the W statistic. To do so, we calculate W as a function of pivot error and see how W changes as HCFAs are removed from V^{test}. We ran 500 simulations using synthetic V^{test} vectors generated by randomly selecting 300 observations from V^{normal} and 100 observations from V^{atk}. This vector agrees with our calculated z (400 observations), and matches the observed ratio of attack observations to non-attack observations. Recall that in §3.3.1 we determined that after log-level filtering, approximately 25.5% of $v(\Lambda^{\text{sig}})$ observations were attacks.

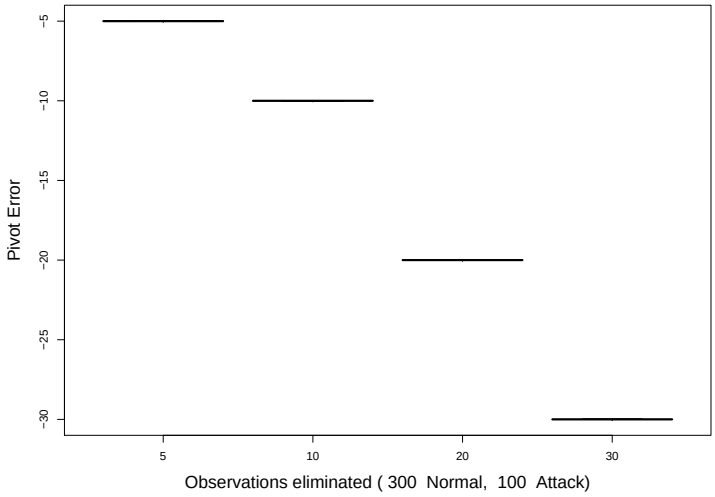
Figure 3.5 plots the result of our simulation. The shape in Figure 3.5 is a series of boxplots, with the x-axis describing the pivot error and the y-axis

the range of values taken by the boxplot. In this figure, each boxplot shows the minimum, first quartile, median, third quartile and maximum observed value for each pivot error. As this figure shows, W jumps significantly as the pivot error approaches zero. Once the scans are removed, the W value remains stable even as we remove more observations from the data set. This result justifies using a relatively stringent θ_W, and so we will set θ_W to 0.95.

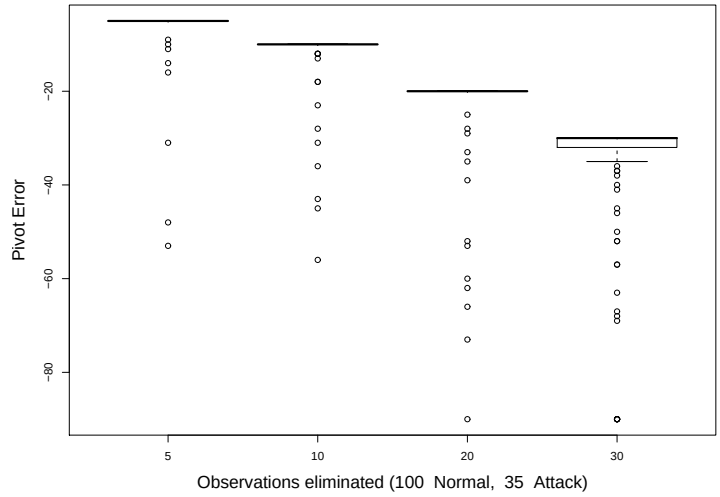
Given the stability of W with negative pivot errors, we now address two problems: removing observations after W is above θ_W, and using different V^{test} sizes. Our interest in the former problem is a redundancy and reliability issue. Recall that a positive pivot error is unacceptable while a negative error is tolerable. By removing additional points we guarantee that the resulting vector has a negative pivot error. The second problem is a test of the validity of the estimate we generated in §3.3.3: if the efficacy of the estimate improves with larger vectors, then the investment in additional learning time may be worthwhile.

Figure 3.6 plots the pivot error as a function of the number of observations removed after $W > \theta_W$: 5, 10, 20 and 30 observations respectively. As before, each plot in Figure 3.6 is a series of boxplots; in this case, the x-axis is the number of observations eliminated once θ_W is reached, and the y-axis is the pivot error. Each plot uses a boxplot with outliers represented as single points. Furthermore each figure uses a different total V^{test} size: 400 observations (our standard case) in Figure 3.6(i), 135 in Figure 3.6(ii) and 4000 in Figure 3.6(iii).

Our standard case, Figure 3.6(i), shows that the pivot error is identical to the number of observations removed after reaching θ_W. For that sample

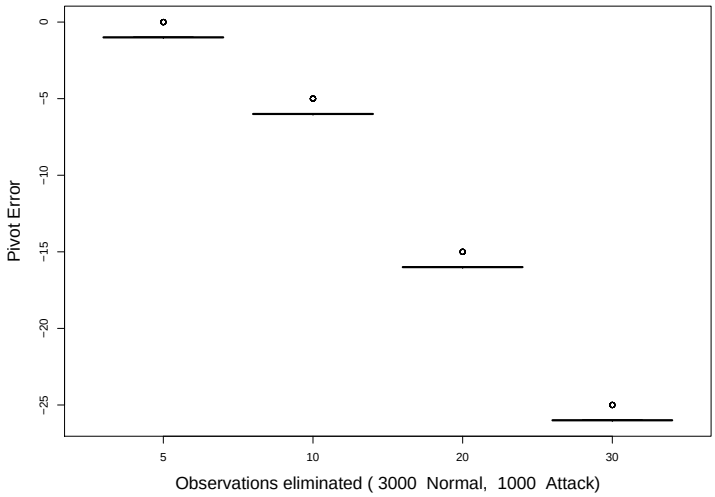


(a) 300 normal, 100 attack



(b) 100 normal, 35 attack

Figure 3.6: Impact of removing vertices after W exceeds θ_W for different sample sizes of equivalent composition.



(c) 3000 normal, 1000 attack

Figure 3.6 continued.

size, there is no reason to remove additional observations. Furthermore, we note that this boxplot has no variation. However, for different vector sizes, we see greater variation, and of different types depending on the sizes.

Figure 3.6(ii) plots the results with 135 observations, a considerably smaller vector size. This figure shows that the value of the pivot error is relatively stable, but that there are a significant number of outliers (approximately 15% of all the runs). More significant, however is that the pivot error was consistently negative even before removing observations, sometimes by up to 20 observations. In these cases, shapfilt is overestimating the pivot. This is likely a result of the small size of V^{test} in these simulation runs. We can compensate for these results by using a less stringent θ_W, but see no compelling reason to do so. We have access to data as long as we are willing

to wait a sufficiently long time, and the savings on training time is minimal compared to the loss in accuracy.

Figure 3.6(iii) plots the pivot error using a much larger vector: 4000 total observations. In this case, the pivot error is actually slightly higher than the number observations removed (*e.g.*, when five observations are removed, the actual pivot error is -2). As before, the estimates are very stable, with a limited number of outliers that are even higher. This result indicates that, at least for these vector sizes, the number of normal observations reduces the loss in accuracy caused by missing attack observations. However, we also have to note that this is a very large amount of normal observations – 3,000 normal observations are offsetting the effect of around 2 attack observations. Consequently, we conclude that at larger sample sizes, removing additional observations is useful, but not critical for acquiring a more accurate estimate. Furthermore, we note that we have no compelling reason for training the system for forty hours when we have achieved comparable results after three.

Given these results, we are confident that we can train the system with 500 observations in V^{test}, setting θ_W to 0.95 and without removing any additional observations. We must now evaluate the impact of this filtering method. To do so, we calculate the *anomaly threshold*: $\mu_V + 3.5\sigma_V$. This value is the expected point at which our IDS, trained on a filtered or unfiltered dataset, would raise an alarm. Figure 3.7 compares the anomaly thresholds generated using μ_V and σ_V values for three different scenarios: raw data (*i.e.*, no filtering at either the log or the state level), after log-level filtering and after state-level filtering. In this figure, the x-axis is the estimated anomaly threshold, and the y-axis is the frequency with which we

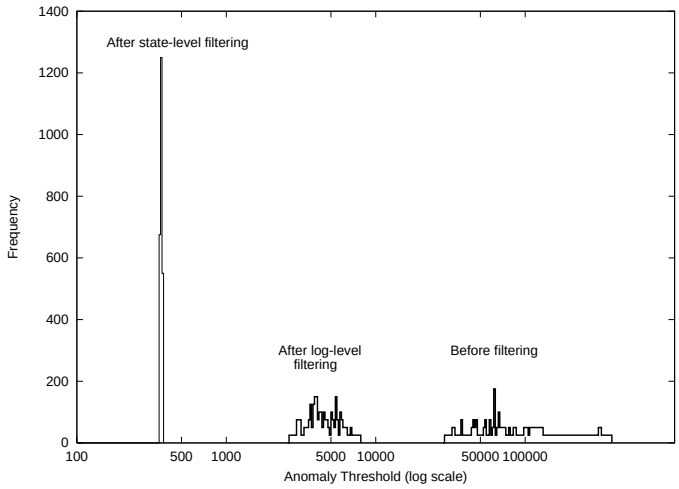


Figure 3.7: Impact of state-level and log-level filtering on data set.

derived that threshold in our tests. The x-axis on this plot is logarithmically scaled, and plots the distribution of observed thresholds over 5000 test runs for each scenario. As this plot shows, the impact of the filtering is significant - the expected threshold without filtering is 91,000 vertices, with expected thresholds of 4,600 and 370 vertices for the other two categories.

These numbers show that unless aggressive traffic filtering such as ours is implemented, the attacker needn't bother with subtlety. In the worst case scenario an attacker can use the resources of a full /16 without the IDS noticing an attack. Even with just log-level filtering, the attacker can DDoS a target and hide in the noise.

3.5 Impact Evaluation and Notification

Since HCFAs are common and rarely successful, simple alerts mean that an accurate warning system is still counterproductive: we have developed a system that can now accurately detect attacks and, based on the observed attack frequency, will raise an alarm every two minutes.

We address this problem by dividing alerts on a criterion of *actionability*. The majority of HCFAs are inactionable in the sense that there is relatively little do that an operator can do about them. They are likely to be automated and conducted from compromised hosts, meaning that the actual culprit is elsewhere and the actual owner of the host is unaware of the attack. An attack is actionable if it merits activity by an operator, inactionable otherwise.

Determining actionability is a two-stage process. In the first stage, each of the processors (log-level and state-level) generates a list of anomalous addresses, A, from the current period's log data. These addresses are then passed to an *evaluator*, which determines if there is any activity from the addresses in A that is actionable. As with significance, actionability is a quality that we approximate from concrete behaviors in the log files. Currently, an anomalous address is actionable if there exists a flow from the anomalous address which includes nontrivial payload. For example, if an anomalous address does find an actual target and communicates with it, then that address is actionable. If an anomalous address is actionable, it is sent to operators for further examination, while inactionable addresses are placed on a watchlist which can be examined by operators at their discretion.

Log-level and state-level processors use different criteria for determining if an address is anomalous. Recall that Λ^{insig} will generally be orders of magnitude larger than Λ^{sig} for any time, and consequently the log processor cannot conduct particularly subtle analyses. In the current implementation, the log processor uses a simple top-5 list, identifying the 5 highest degree clients and passing them to the evaluator. The state-level processor will use different methods based on the state model, since our implementation of state-level filtering is graph based, we use the component counting method from §2.5.

An important consequence of this processing approach is that the log processor will generate a steady stream of anomalies and will have a high false positive rate. However, while the log processor has a high false positive rate, the aggregate system does not: the evaluator will not pass an alert about a high-degree host unless that host has done something actionable. In comparison to the log processor, the state processor will generate alerts infrequently (based on the provided data, approximately once every two minutes). The evaluator also expects some degree of redundancy between the anomaly lists - an attack with mixed success may appear in both Λ^{sig} and Λ^{insig}.

3.6 Attacking Attack Reduction

Attack reduction reduces distractions from HCFAs so that an IDS can focus on more successful, and therefore higher-risk, attacks. To do so, it assumes that the majority of attacks are HCFAs high failure, reaching targets that

don't exist and having little impact on the network. We must now address intelligent attackers: how an attacker conversant in attack reduction can manipulate the system to their advantage.

We contend that the primary result of our statistical IDS is to effectively throttle attacker behavior. We assume that an intelligent attacker does not want to be noticed, and too much activity is going to raise alerts from the IDS. Using our graph-based IDS, he can effectively only engage in any activity that involves payload-bearing communications to less than 370 clients and servers combined in a 30s period. This is an upper limit, since it does not include normal activity. Evasion therefore consists of operating under this limit.

An attacker could manipulate the attack reduction system by inserting consistent chaff. Assume that the attacker is aware of when the IDS is in state-level filtering. Throughout this training period, the attacker consistently sends short sessions crafted to pass the log-level filter into the model to nonexistent targets. Because the other traffic is normally distributed, the constant attacks will increase μ_V and provide the attacker with a buffer. Then, during normal operations the attacker can disengage the sources sending chaff traffic and use that breathing room for his own activity.

To do this, the attacker must consistently commit hosts to an attack. Once the system is trained, however, he may return those hosts to his own uses. One way to identify for this type of anomaly would be to use symmetric thresholds. Since Gaussian distributions are symmetric, an unusually low value should be as rare as an unusually high value. If those thresholds are incorporated, than an attacker would have to maintain his hosts in place

until the attack was actually conducted.

3.7 Conclusions

This chapter has developed a method to implement an anomaly detection model in protocols that are under constant attack. These consistent HCFAs impact the training and notification processes of anomaly detection systems, and we have shown that without aggressive filtering, an anomaly detection system cannot detect massive attacks in popularly targeted protocols. Using attack reduction we are able to develop a more precise model of normalcy. The impact of having a more precise model is substantial. For example, we showed that in our monitored network, a simple form of anomaly detection improves by more than two orders of magnitude: it will detect 370 attacking hosts with our techniques, but would miss 91,000 attacking hosts (*e.g.*, in a massive DDoS) in the absence of our techniques. By then filtering these detections into actionable and inactionable anomalies, we are then able to better focus the attention of human analysts on attacks that otherwise would have been lost in the noise of HCFAs and that are likely to be more important to examine.

Chapter 4

Evaluating Anomaly Detection Systems

We address the problem of evaluating network intrusion detection systems, specifically against scan and harvesting attacks. In the context of this work, a harvesting attack is a mass exploitation where an attacker initiates communications with multiple hosts in order to control and reconfigure them. This type of automated exploitation is commonly associated with worms, however, modern bot software often includes automated buffer-overflow and password exploitation attacks against local networks[1]. In contrast, in a scanning attack, the attacker's communication with multiple hosts is an attempt to determine what services they are running; i.e., the intent is reconnaissance.

[1] A representative example of this class of bot is the Gaobot family, which uses a variety of propagation methods including network shares, buffer overflows and password lists. A full description is available at `http://www.trendmicro.com/vinfo/virusencyclo/default5.asp?VName=WORM_AGOBOT.GEN`.

While harvesting attacks and scanning may represent different forms of attacker *intent* (*i.e.*, reconnaissance vs. host takeover), they can appear similarly in traffic logs. More specifically, a single host, whether scanning or harvesting, will open communications to an unexpectedly large number of addresses within a limited timeframe. This behavior led to Northcutt's observation that in the absence of payload—either due to the form of log data, encryption or simply a high connection failure rate—methods for detecting these attacks tend to be threshold-based [42]. That is, they raise alarms after identifying some phenomenon that exceeds a threshold for normal behavior.

Historically, such IDS have been evaluated purely as alarms. Lippmann *et al.* [34] established the practice for IDS evaluation in their 1998 work on comparing IDS data. To compare intrusion detectors, they used ROC curves to compare false positive and false negative rates among detectors. Since then, the state of the practice for IDS evaluation and comparison has been to compare the effectiveness of ROC curves [23].

The use of ROC curves for IDS evaluation has been criticized on several grounds. For our purposes, the most relevant is the critique of the *base rate fallacy* described by Axelsson [5]. Axelsson observes that a low relative false positive rate can result in a high number of actual false positives when a test is frequently exercised. For NIDS, where the test frequency may be thousands or tens of thousands of per day, a false positive rate as low as 1% may still result in hundreds of alarms.

In this chapter, we introduce an alternative method of evaluating IDS based around their capacity to *deter* attackers. In order to do so, we develop a model for evaluating IDS over an *observable attack space*. The observable

attack space represents a set of attacks an attacker can conduct as observed by a particular logging system. The role of logging in the observable attack space is critical; for example, NetFlow, the logging system used in this chapter, does not record payload. As such, for this chapter, we define an observable attack space that classifies attacks by the attacker's *aggressiveness* (the number of addresses to which they communicate in a sample period) and their *success* (the probability that a communication opened to an address actually contacts something).

Using this notion of an observable attack space, we develop a payoff-based method for evaluating IDS effectiveness. This mechanism consists of constructing a *detection surface*, which is the aggregate probability of detection over the observable attack space, and then applying a *payoff function* to this detection surface. The payoff function is a function representing the rate at which an attacker achieves the strategic goal of that attack, which is either occupying hosts (in a harvesting attack) or scouting network composition (in a scanning attack).

By combining detection surfaces with a payoff function, we are able to compare IDS with greater insight about their relative strengths and weaknesses. In particular, we are able to focus on the relationship between detection capacity and attacker payoff. Instead of asking what kind of false positive rate we get for a specific true positive rate, we are able to map the false positive rates to the attacker's goals. By doing so, we are able to determine how high a false positive rate we must tolerate in order to prevent an attacker from, say, substantially compromising a network via a harvesting attack. Our work therefore extends ROC-based analysis into a

payoff framework, and by doing so we are able to address problems such as the minimal acceptable false positive rate to deter attackers.

Using this approach, we compare the efficacy of five different detection techniques: client degree (*i.e.*, number of addresses contacted); protocol graph size and protocol graph largest component size [13]; server address entropy [32]; and Threshold Random Walk [26]. We train these systems using traffic traces from a large (larger than /8) network. Using this data, we demonstrate the configurations of aggressiveness and success rate with which an attack will go undetected by any of these techniques. Furthermore, we show that in order to ensure that an attacker is consistently detected, these anomaly detection systems must accept false positive rates resulting in hundreds of false alarms daily.

To summarize, the contributions of this chapter are the following. First, we introduce a new methodology for evaluating NIDS that do not utilize payload. Second, we apply this methodology to evaluate several attack detection methods previously proposed in the literature, using data from a very large network. And third, we demonstrate via this evaluation the limits that these techniques face in their ability to deter attackers from reaching harvesting or scanning goals.

The remainder of this chapter is structured as follows. §4.1 is a review of relevant work in IDS evaluation and anomaly detection. §4.2 describes the IDS that we evaluate in this chapter, and how we configure them for analysis. §4.3 describes the observable attack space and detection surface. §4.4 describes the first of our two attack scenarios, in this case the acquisition of hosts by an attacker with a hit list. §4.5 describes the second scenario:

reconnaissance by attackers scanning networks. §4.6 concludes this work.

4.1 Previous Work

Researchers have conducted comparative IDS evaluations in both the host-based and network-based domains. In the host-based domain, the most extensive work done is by Tan and Maxion [62, 37] who develop an experimental and evaluation methodology for comparing the effectiveness of multiple host-based IDS. Of particular importance in their methodology is the role of the data that an IDS can actually analyze, an idea further extended in Killourhy *et al.*'s work on Defense-centric taxonomy [27]. The methods of Tan, Maxion and Killourhy informed our experimental methodology and the concept of an observable attack space, however their approach is focused on host-based IDS and they consequently work with a richer dataset then we believe feasible for NIDS.

A general approach to evaluating IDS was proposed by Cárdenas *et al.* [9], who developed a general cost-based model for evaluating IDS based on the work of Gaffney and Ulvila [19] and Stolfo *et al.* [60]. However, these approaches all model cost from a defender-centric viewpoint — the defensive mechanism is assumed to have no impact on the attacker. Our work is motivated by previous work on opportunistic attackers [12], where we assumed that attackers had no interest in a target except insofar as it was exploitable.

The general problem of NIDS evaluation was first systematically studied by Lippmann *et al.* [34]. Lippmann's comparison first used ROC curves

to measure the comparative effectiveness of IDS. The ROC-based approach has been critiqued on multiple grounds [38, 23, 5]. Our evaluation model is derived from these critiques, specifically Axelsson's [5] observations on the base-rate fallacy. Our work uses a ROC based approach (specifically, comparing Type I and Type II errors) as a starting point to convert the relative error rates into payoffs.

4.2 Alarm Construction and Training

In the context of this work, an *alarm* is an anomaly based detection system that compares the current state of a network against a model of that network's state developed from historical data. In this section, we describe our candidate alarms, and the method for training and configuring them. This section is divided as follows: §4.2.1 describes the raw data, §4.2.2 describes the types of alarm used, and §4.2.3 describes the measurement and final estimates used for our alarm.

4.2.1 Raw Data

Every alarm in this chapter is trained using a common data source over a common period of time. The source data used in this chapter consists of unsampled NetFlow records[2] generated by internal routers in a large (in excess of 16 million distinct IP addresses) network. For training and evaluation, we use SSH traffic. SSH was chosen because it is a very popular

[2]CISCO Systems, "CISCO IOS Netflow Datasheet", `http://www.cisco.com/en/US/products/ps6601/products_data_sheet0900aecd80173f71.html`, last fetched October 8th, 2007.

target for attack, as observed by Alata *et al.* [3].

For the purposes of this chapter, we treat NetFlow records as tuples of the form (clntip, srvip, success, stime). The elements of this tuple are derived from the fields available in CISCO NetFlow. The clntip, srvip, success and stime fields refer, respectively, to the client address, server address, whether a session was successful, and the start time for the session. Since SSH is TCP based, we rely on the port numbers recorded in the original flow record both for protocol identification and classifying the role a particular address played in the flow. Any flow which is sent to or from TCP port 22 is labeled an SSH flow, srvip is the address corresponding to that port and clntip the other address[3]. stime, the start time, is derived directly from the corresponding value in the flow record, and is the time at which the recording router observed the flow's earliest packet.

The success element is a binary-valued descriptor of whether the recorded flow describes a legitimate TCP session. success is 0 when the flow describes a TCP communication that was not an actual session (*e.g.*, the target communicated with a nonexistent host), 1 when the flow describes a real exchange between a client and a server. *In situ*, this value is an approximation of a property that that can only be truly determined by the receiving host; a sufficiently perverse attacker could generate one side of a session and send that information in order to fool the data collection system. During live use, success can be approximated using other flow properties such as the number of packets in the flow or TCP flag combinations. For our simulations, we

[3]We constrain the flows used in this chapter to flows which used an ephemeral port between 1024 and 5000.

generate success ourselves: the source data consists of traffic already subject to log-level filtering as specified in our work on IDS training [14], and we therefore treat all of that traffic as successful.

Alarm properties are generated using 30 second (s) samples of traffic data. We refer to a distinct sample as a *log file*, Λ, consisting of all the flows $\lambda_1 \ldots \lambda_l$ whose stime values occur in the same 30s period.

4.2.2 Alarm State Variables

In the context of this chapter, an *alarm* is a threshold on a value derived from a log file Λ; if this value exceeds the alarm's threshold, then an alert is generated for processing by the IDS. Each alarm in this chapter is based around a single *state variable* which, when evaluated against a log file produces a scalar *state value*. For this chapter, we evaluate the state of a log file using five distinct state variables: v, c, h, d and r. Each state variable is used by one alarm; we will refer to the alarms by their state variable (*e.g.*, v is an alarm).

We note that an alarm is *not* an IDS, but a component thereof. As noted in Chapter 3, in addition to simply raising an alert, a useful IDS must also provide context for the alert. We therefore consider an IDS to be a system that interprets alarms and recommends courses of action, such as identifying the cause of an anomaly. For the purposes of this chapter, we assume that if an alarm is accurate, the IDS managing that alarm operates with perfect accuracy (*e.g.*, if an alarm is raised about a bot taking over hosts, the IDS will identify that bot and provide that information to operators).

The first two alarms, $v(\Lambda)$ and $c(\Lambda)$ are, respectively, the total graph

size and the largest component size of the protocol graph constructed from Λ, as described in §2.2.1. $h(\Lambda)$ is the entropy of server addresses in Λ. This metric is derived from work by Lakhina *et al.* [32] on mining traffic features. The entropy is defined as:

$$h(\Lambda) = -\sum_{i \in \mathsf{srvs}(\Lambda)} \left(\frac{|\{\lambda \in \Lambda | \lambda.\mathsf{srvip} = i\}|}{|\Lambda|} \right) \log_2 \left(\frac{|\{\lambda \in \Lambda | \lambda.\mathsf{srvip} = i\}|}{|\Lambda|} \right) \tag{4.1}$$

where $\mathsf{srvs}(\Lambda) = \bigcup_{\lambda \in \Lambda} \lambda.\mathsf{srvip}$ is the set of all server addresses observed in the log file. During a harvesting attack, an attacker will increase $|\mathsf{srvs}(\Lambda)|$, which reduces the probability of any one server being the target of a communication and therefore increases the entropy.

$d(\Lambda)$, the maximum degree of Λ, is the number of servers with which the busiest client in Λ communicated. $d(\Lambda)$ is arguably the simplest form of scan detection and consequently has been used by a variety of anomaly detection systems, notably GrIDS [58] and Bro [44].

$r(\Lambda)$ is the maximum *failed connection run* observed in Λ. A failed connection run is a sequence of flow records $\lambda_1 \ldots \lambda_n$ where each λ in the run has the same server address and $\lambda_i.\mathsf{success} = 0$. This method is used by TRW scan detection [26] to determine if an address is actively scanning. As implemented by Jung *et al.*, TRW is a streaming algorithm that employs sequential hypothesis testing in order to determine if a particular host is scanning. We use the maximum failed connection run measure to indicate whether or not the streaming TRW approach would have detected at least one attack during the period.

4.2.3 Alarm Thresholds

In order to calculate detection thresholds for four of the alarms we consider (v, c, h and d), we first must train the alarm using log files of benign traffic from the monitored network. However, SSH traffic is prone to constant scanning [3] which, unless aggressively filtered, will result in artificially high thresholds. We address this by using the attack reduction methodology from Chapter 3. The initial training data consisted of 7,200 log files for the five business days between February 11–15, 2008. Source data was chosen exclusively from 1200GMT to 2359GMT for each day, a period corresponding to peak activity for the network. After filtering, the resulting set consisted of 5,619 log files from a source set of 7,200.

Applying this filtering technique in order to isolate benign traffic yields a vector $\Lambda_1 \ldots \Lambda_m$ of log files, each representing benign traffic in a 30s interval. State values are calculated for each log file in this vector; we refer to the resulting vector of state values using the same subscript notation, *e.g.*, $r(\Lambda_i) = r_i$. We refer to the complete vector of values for a vector of log files by the corresponding capital letter (*e.g.*, $V = \{v(\Lambda_1) \ldots v(\Lambda_m)\}$).

We examined the H and D distributions in the filtered data to see if they could be modeled via a Gaussian distribution. (Our previous work already established that V and C are Gaussian for the monitored network [13].) Figure 4.1(a) shows a distribution of H after filtering. As this figure shows, entropy can be satisfactorily approximated using a normal distribution, with a Shapiro-Wilk statistic of $W = 0.97$ and negligible p-values. D, shown in Figure 4.1(b), had a Shapiro-Wilk statistic of $W = 0.77$ with negligible

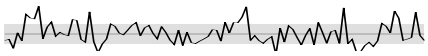
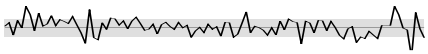
State variable x	Range	$\mu_X \pm \sigma_X$
v		299.27±42.49
c		35.13±21.32
h		6.88±0.35

Table 4.1: Summary of Gaussian state variables in SSH training set.

p-value, and consequently was not considered Gaussian.

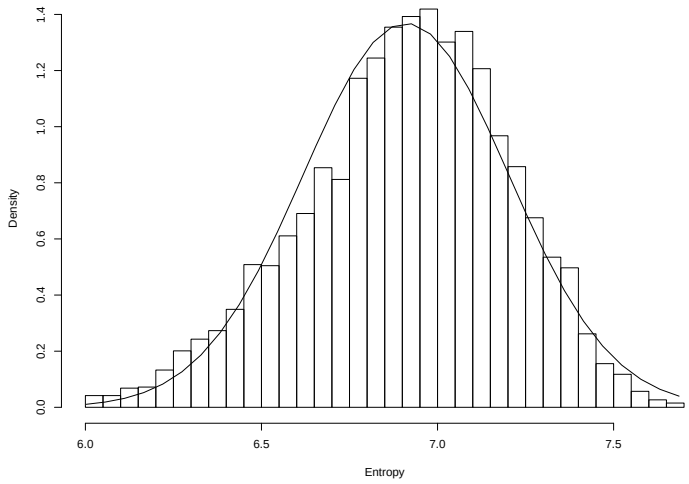
Table 4.1 summarizes the Gaussian state variables, *i.e.*, v, c, and h. This table shows the summary data (left hand column), the mean and standard deviation (right side) and a sparkline for each data set. The sparkline is a time series plot of the activity over the training period. We plot the mean and shade an area one standard deviation from the mean in each sparkline.

For these three state variables, we can use (4.2) to calculate the detection threshold. For a given false positive rate, FPR, the corresponding threshold for a Gaussian alarm x is given by:

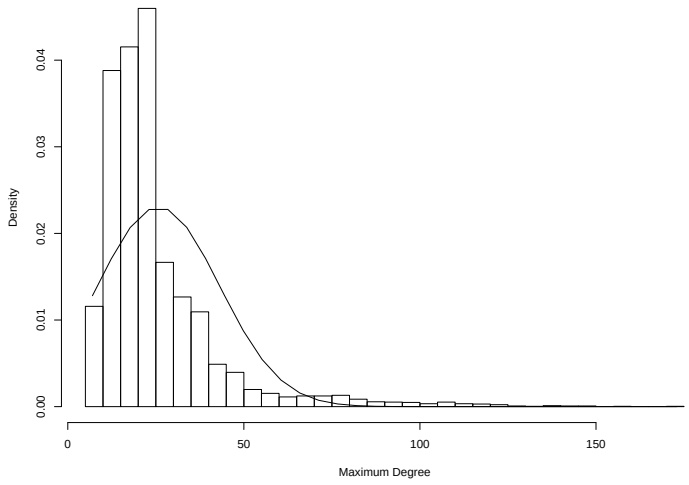
$$\theta_x = \mu_X + \gamma\sigma_X \tag{4.2}$$

Where γ is the threshold (in standard deviations) for a given false negative rate $\mathsf{FNR} = 1 - \mathsf{FPR}$, as in Equation 2.7. μ_X is the arithmetic mean of the vector of observations X, and σ_X is the standard deviation of the same vector.

The detection threshold for d is computed differently since, as shown above, d is not normally distributed over the sample space. We use d's



(a) Distribution of h in SSH data. Resulting fit has a W statistic of 0.97 with negligible p-value.



(b) Distribution of d in SSH data. Resulting fit has a W statistic of 0.72 with negligible p-value.

Figure 4.1: Distribution and normal approximations of h and d in observed data set.

maximum observed value over the benign log files as the detection threshold:

$$\theta_d = \max(\mathrm{D}) \tag{4.3}$$

The detection threshold for r is the only one that is not based on the observed history of the monitored network. Rather, Jung *et al.* express the threshold for detection as follows:

$$\theta_r = \frac{\mathsf{TPR} \ln \frac{\mathsf{TPR}}{\mathsf{FPR}} + (1 - \mathsf{TPR}) \ln \frac{1-\mathsf{TPR}}{1-\mathsf{FPR}}}{t_1 \ln \frac{t_1}{t_0} + (1 - t_1) \ln \frac{1-t_1}{1-t_0}} \tag{4.4}$$

Here, FPR and TPR are user configured thresholds for the maximum false positive rate (FPR) and the minimum true positive rate (TPR). t_0 and t_1 are, respectively, the probabilities that a randomly scanning attacker will successfully communicate with a target, and the probability that a normal user will fail to communicate with a target. For this work, we set $\mathsf{TPR} = 1 - \mathsf{FPR}$, and set FPR to our acceptable FPR (see below). We use Jung's original values of $t_0 = 0.8$ and $t_1 = 0.2$.

Recall that for this work, we monitor traffic over 30s periods. We restrict an alarm to sending one alert at the end of that period. If we constrain the *aggregate* false positives for *all* of the detectors to one false alarm per eight hours (*i.e.*, the duration of a typical network analyst's shift), this yields a combined rate of 0.1% for the five alarms together. If we permit each detector to contribute equally to this aggregate false positive rate, and if we assume, for simplicity, that the five alarms are independent, then we can

solve for their individual false positive rates FPR using

$$0.001 = 1 - (1 - \mathsf{FPR})^5 \tag{4.5}$$

Plugging this value of FPR into (4.2) yields detection thresholds $\theta_v = 447$, $\theta_c = 110$, and $\theta_h = 8.105$, and setting $\mathsf{FPR} = \mathsf{FPR}$ in (4.4) yields $\theta_r = 6$. We also use the value $\theta_d = 150$, computed directly from (4.3). These are the detection thresholds we utilize in our evaluations in the subsequent sections.

4.3 Observable Attack Spaces and Detection Probability

In §4.2, we developed and configured a combined alarm based around alarms using five different state variables: graph size v, largest component size c, server address entropy h, maximum client degree d and maximum failed connection run r. In doing so, we specifically configured these systems to yield a low false positive rate, resulting in one false positive per eight-hours as predicted by our training data. Now that we have developed this hybrid alarm, we can evaluate its efficacy for deterring attackers.

In order to do this, we develop a method for describing attacker utility which we call the *observable attack space* (OAS). An observable attack space describes the range of attacks that an attacker can conduct *as observed by a particular logging mechanism.* In this section, we develop an observable attack space common to our logging system (NetFlow) and our five candidate arlams. Using this approach, we model the aggregate *detection surface* of the

OAS and use this to evaluate both our combined alarm and the constituent alarms individually.

This section is structured as follows. §4.3.1 describes the observable attack space relation between OAS and alarms. §4.3.2 describes how we estimate the detection surface. §4.3.3 then compares the effectiveness of our five detection methods both in aggregate and as individual detection schemes.

4.3.1 Observable Attack Spaces and Alarms

The type of log data that an alarm uses strongly impacts the types of attacks which that alarm can detect. An example of this is the impact of choosing NetFlow. NetFlow is a compact representation of network traffic, and therefore can be effectively used on larger networks than more payload-intensive methods will allow. However, since NetFlow lacks payload, signature-matching techniques are not viable with this log format.

An observable attack space is therefore a parameterized representation of all possible forms of a particular attack, as observable by a particular logging system. For this work, the observable attack space has two attributes: *aggressiveness* (a) and *success* (s) observed within a 30s period. The aggressiveness is a natural number describing the number of distinct addresses with which the attacker communicates in the observation period. The success of an attack is fraction of these communications that were successful, and is within the range of $[0, 1]$.

When conducting simulations, we limit the a to the range of $(0, \theta_d)$ because we treat the d alarm as deterministic — it will trigger if *and only*

if $a \geq \theta_d$. In doing so, we ignore the possibility that during an attack, a benign host contacts more than θ_d addresses, thus "accidentally" causing a true detection even though $a < \theta_d$. This treatment also presumes that the attack is launched from a bot that is not also contributing benign traffic at the same time, *i.e.*, $a < \theta_d$ implies that the bot host does, in fact, contact fewer than θ_d addresses in a 30s interval. The other alarms' chances of detecting attacks are not so directly dependent on an attack's characteristics within the OAS. We express the uncertainty involving these alarms by their *probability of detection*, discussed in §4.3.2.

4.3.2 Estimating the Detection Surface

Consider a particular alarm $x \in \{v, c, h, r\}$. Given an arbitrary log file of control data Λ^{ctl} that we are confident does not contain an attack, $\mathcal{P}^x_{\mathsf{det}}(a, s)$ is the probability that the alarm x can identify that the log file resulting from Λ^{ctl} merged with an attack Λ^{atk} with aggressiveness a and success s. That is,

$$\mathcal{P}^x_{\mathsf{det}}(a, s) = \mathbb{P}\left[x(\Lambda^{\mathsf{atk}} \cup \Lambda^{\mathsf{ctl}}) \geq \theta_x\right] \tag{4.6}$$

where the probability is taken with respect to the selection of Λ^{ctl} and the generation of Λ^{atk} with aggressiveness a and success rate s. For a particular alarm x, the *detection surface of* x is the surface of values $\mathcal{P}^x_{\mathsf{det}}(a, s)$ for $a \in (0, \theta_d)$ and $s \in [0, 1]$.

More specifically, to estimate the probability of detection and the corresponding detection surface, we evaluate the distribution of state variables for normal behavior merged with randomly generated attacks meet-

ing the aggressiveness and success requirements specified by a and s. For this chapter, we limit our simulations to $a \in \{10, 20, 30, 40, \ldots, 140\}$ and $s \in \{0.1, 0.2, 0.3, \ldots, 1.0\}$. At each point, we conduct 100 simulations, each using one of fifty randomly selected 30s periods from the week of February 18–22 (the week following that used for training) for Λ^{ctl}. Λ^{atk} is randomly generated for each simulation. Λ^{atk} contains a unique records, where each record has the same client address, and a different server address. The composition of the server addresses is a function of s: as addresses are chosen from a hit list $\mathsf{HitList}$ of internal SSH servers identified in the training data[4] in order to approximate hit list attacks; the remainder are sent to addresses with no listening server. We then merge Λ^{atk} with a randomly selected control log Λ^{ctl} and then calculate the state variables.

Four of the alarms examined by this chapter (v, c, h, and d) are unaffected by the order of log records within the monitored 30s period. The fifth, r, is order-sensitive, however, in that TRW triggers an alert if any host is observed making more than θ_r failed connections *in a row*. This order sensitivity is a weakness, since an attacker can interleave scanning with connections to known hosts in order to avoid a failed connection run greater than θ_r [28]. As such, to avoid this weakness in TRW, we randomly permute the records originating in each 30s interval. After this permutation, r is calculated for each host in the network.

Figure 4.2 plots the detection surface for all the alarms combined. As this figure shows, the combined detection mechanism generally increases

[4]This hit list is composed of all internal addresses in the training data which had one flow originating from them on port 22 and with a payload of greater than 1kB.

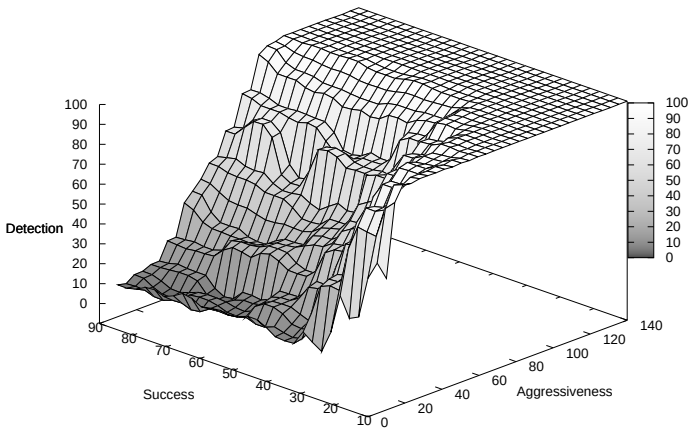


Figure 4.2: Detection surface ($\mathcal{P}_{\mathsf{det}}^{\mathsf{all}}(a, s)$, as a percentage) for combined alarms.

proportionally to the aggressiveness of the attack and inversely relative to the success of the attack. Furthermore, the detection mechanisms tend to vary more as a function of the aggressiveness than due to the success rate.

The effectiveness of the aggregate alarm may be considered low, in the sense that an attacker with a relatively small hit list ($a = 40, s = 0.5$) can communicate with the network with little fear of detection. However, we should note that the attacks represented by this OAS are the most subtle space of attacks available. Our own experience indicates that the majority of attacks are orders of magnitude more aggressive than these hypothetical SSH scans, at which point *any* alarm will identify them. This latter point is particularly critical. As Figure 4.2 shows, once $a \geq 100$, the combined alarm will be triggered.

4.3.3 Detection Surface Comparison

In addition to the detection surface for the aggregate alarm, we have also calculated the detection surfaces for each component alarm in this system. We can use these results to evaluate the comparative effectiveness of each alarm.

Figure 4.3 plots detection surfaces for each alarm $x \in \{v, c, h, r\}$ as *contour plots*. A contour plot maps a 3-dimensional value into a 2-dimensional representation using *contour lines* — each pair of lines bounds a range of values indicated by the value of the line.

These plots show that the most successful individual alarms are c and r: these alarms are the only ones to have significant ($\geq$ 10%) detection rates over the majority of the OAS. In contrast, the entropy-based detection method has a very *low* detection rate, less than 6% over the entire OAS. Of particular interest with c and r is their relative disconnectedness to each other: r's detection rate is dependent on s and less dependent on a. Conversely, c is largely independent of s, while a plays larger role in detection.

We note at this point that all of the detection methods have relatively low success rates. There are two reasons for this: the first is that these techniques are calibrated to have an effective false positive rate of zero — as a result, they are far less sensitive to anomalies. Second, as noted above, the range of attacks represented here are extremely clever — our training data, we eliminated scanners who would communicate with upwards of 50,000 hosts during a 30s period. Those clumsy attackers would be identified and

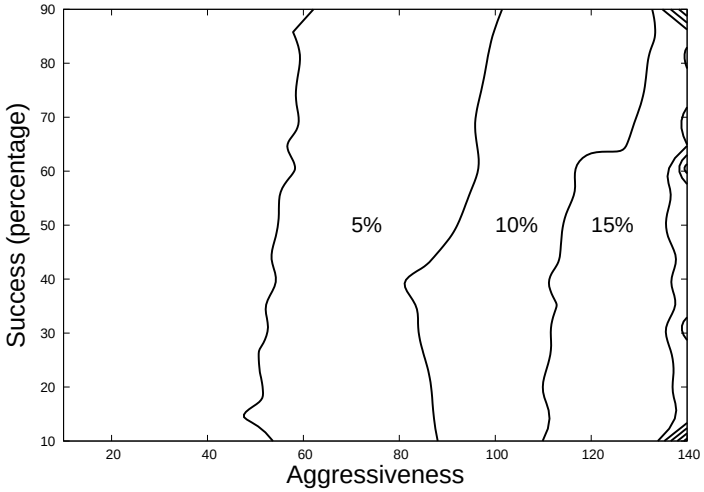
eliminated *regardless* of the detection strategy used — by the time an attacker communicates with 150 addresses, the d alarm will raise an alarm, making other approaches effectively moot.

This phenomenon is partly observable in our models in Table 4.1. Recall that, for example, the model of graph size v, was 299 ± 42.47 hosts. If $v(\Lambda) = 299$ for some log file Λ, then an attacker will not trigger an anomaly until he has communicated with 149 hosts, at which point he is close to triggering d as well as v.

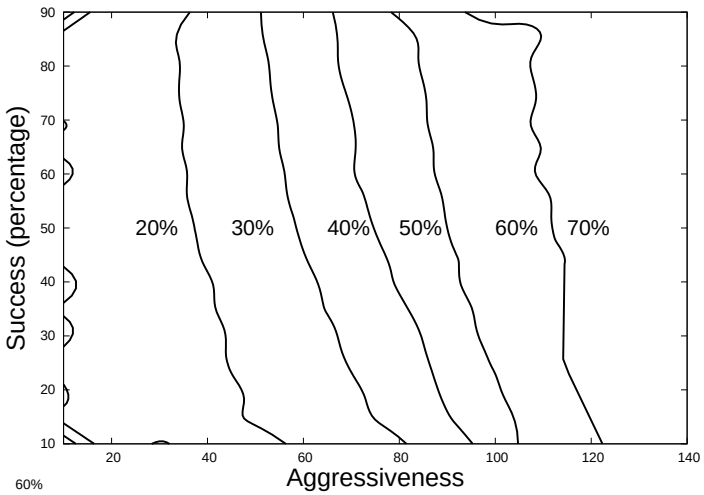
4.4 Modeling Acquisition

In §4.3.3 we examined the efficacy of the detection mechanisms purely as detectors: for a fixed false positive rate, we calculated the effective true positive rate. In this section, we use the detection surface in Figure 4.2 to examine the impact of alarms on *acquisition attacks*. We evaluate the efficacy of the detection surface by building a mathematical model for attacker payoff during an acquisition attack. Applying this model to the surface, we can determine how many hosts an attacker can expect to take over, and from these results determine how effective an alarm has to be in order to keep attackers from taking over hosts.

This section is divided as follows: §4.4.1 describes our model for acquisition attacks. §4.4.2 compares alarm efficiency using our payoff function. §4.4.3 considers the problem of alarm evaluation from a different perspective — instead of calculating efficiency in terms of true and false positives, we determine the minimum false positive rate required to counter an attacker.

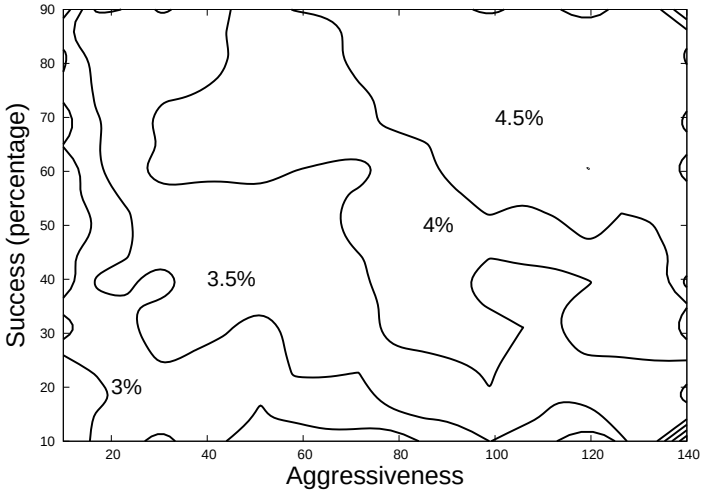


(a) Total graph size v

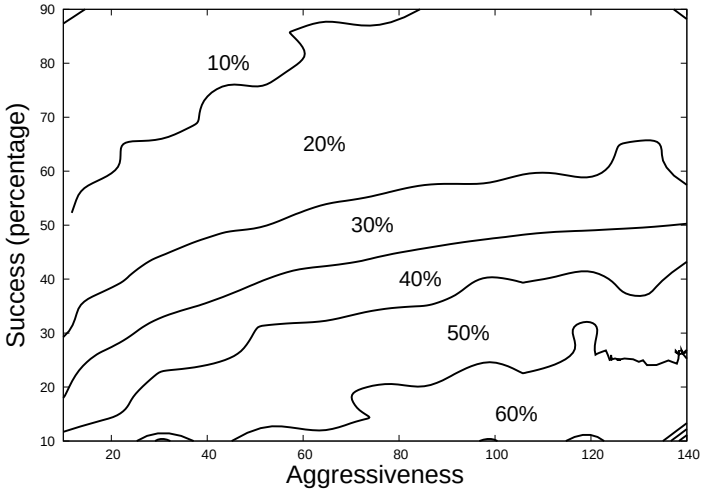


(b) Largest component size c

Figure 4.3: Detection surfaces ($\mathcal{P}^x_{\mathsf{det}}(a, s)$, as a percentage) for individual alarms.



(c) Server entropy h



(d) Maximum failed connection run r

Figure 4.3 continued.

4.4.1 Acquisition Payoff Model

We define an acquisition attack as a game between two parties who are competing for ownership of a single network. The two parties in the game are the *attacker*, who attempts to communicate with hosts on the network, and the *defender*, who attempts to prevent takeovers of hosts on the network. The game is broken into multiple *attempts*, during each of which the attacker uses a single *bot* to communicate with multiple addresses within the network using a *hit list* acquired previous to the attack.

In each attempt, the attacker communicates with some number of addresses (specified by the attacker's a), each of which has s chance of succeeding — for the purposes of the simulations, a successful attack is one that communicates with a vulnerable host at that addresses, and a failed attack is one that does not. The *payoff* of an attempt, $\mathcal{H}_{\mathsf{acq}}$, is the measure of the expected number of hosts with which the attacker communicates during an attempt. We assume that if an attacker talks with a host, the attacker has taken the host over.

The game is zero-sum: the goal of the defender is to minimize $\mathcal{H}_{\mathsf{acq}}$, the goal of the attacker to maximize the same. To do so, the defender deploys multiple alarms x each of which has a different detection surface. The probability of detecting a particular attempt is $\mathcal{P}^{x}_{\mathsf{det}}(a, s)$. If the defender detects an attacker during an attempt, then the defender recovers all of the hosts the attacker has communicated with using that particular bot during the game (*i.e.*, if the same bot is used during multiple attempts, the defender recovers *all* the hosts taken over by that bot during the attempt that the

bot was detected and all previous attempts using that bot).

Let owned be a random variable indicating the number of hosts taken over, and let $\mathsf{alarmed}$ be the event that the bot is detected. Below, we assume that the probability of detection in each attempt is independent. If such is the case, then we can derive the payoff for an attack for an alarm x as:

$$\begin{aligned}
&\mathcal{H}^x_{\mathsf{acq}}(a,s,k) \\
&\quad = \mathbb{E}\left[\mathsf{owned}\right] \\
&\quad = \mathbb{P}\left[\mathsf{alarmed}\right]\mathbb{E}\left[\mathsf{owned} \mid \mathsf{alarmed}\right] + \mathbb{P}\left[\neg\mathsf{alarmed}\right]\mathbb{E}\left[\mathsf{owned} \mid \neg\mathsf{alarmed}\right] \\
&\quad = (1-\mathcal{P}^x_{\mathsf{det}}(a,s))^k(ask) \qquad (4.7)
\end{aligned}$$

The last step follows by taking $\mathbb{E}\left[\mathsf{owned} \mid \mathsf{alarmed}\right] = 0$, since we presume that if the defender detects an attacker during an attempt, then the defender recovers *all* of the hosts the attacker has communicated with using that particular bot. Note that

$$\mathcal{H}^x_{\mathsf{acq}}(a,s,k+1) - \mathcal{H}^x_{\mathsf{acq}}(a,s,k) = (1-\mathcal{P}^x_{\mathsf{det}}(a,s))^k\, as[1-\mathcal{P}^x_{\mathsf{det}}(a,s)(1+k)] \qquad (4.8)$$

As such, it is fruitful for the attacker to use the same bot for multiple attempts (*i.e.*, increment k) so long as $\mathcal{H}_{\mathsf{acq}}(a,s,k+1) - \mathcal{H}_{\mathsf{acq}}(a,s,k) > 0$ or, in other words,

$$k < \frac{1-\mathcal{P}^x_{\mathsf{det}}(a,s)}{\mathcal{P}^x_{\mathsf{det}}(a,s)} \qquad (4.9)$$

Figure 4.4 plots the payoff over the observed attack space using (4.7) with the maximum k satisfying (4.9). As this figure shows, aggressive attacks have

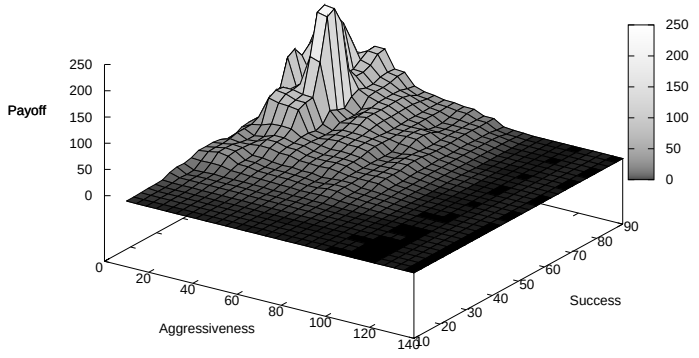


Figure 4.4: Plot of the payoff for acquisition attacks over the OAS.

a minimal payoff, a result that can be expected based on Figure 4.2. Above approximately $a \geq 80$, the attacker is consistently identified and stopped regardless of their success rate.

This behavior is the result of the interaction of two detectors: c and r. As s increases, the probability of the attacker combining previously separate components of the protocol graph increases, increasing the likelihood of detection by the c alarm. As the attacker's success rate decreases, he is more likely to generate a sufficiently long failed connection run to trigger the r detector. The other detectors will identify attackers, however their effectiveness is limited for attacks that are this subtle — an attacker who does disrupt v or h will already have disrupted d.

4.4.2 Calculating Alarm Efficiency

We can use (4.7) to also calculate a comparative efficiency metric for an alarm. The volume under the surface specified by (4.7) is the ability of the attacker to take over hosts in the presence of a particular alarm. The *efficiency* of an alarm x is therefore the indicator of how much x reduces an attacker's payoff. We can express alarm efficiency as the ratio between the number of hosts an attacker expects to take over in the presence of an alarm, and the number of hosts the attacker can take over in that alarm's absence.

$$\mathcal{E}^x_{\mathsf{acq}} = 1 - \frac{\sum_{a\in(0,\theta_d)} \sum_{s\in(0,1]} \mathcal{H}^x_{\mathsf{acq}}(a, s, k^x_{\max}(a, s))}{\sum_{a\in(0,\theta_d)} \sum_{s\in(0,1]} \mathcal{H}^{\varnothing}_{\mathsf{acq}}(a, s, k^x_{\max}(a, s))} \tag{4.10}$$

where $k^x_{\max}(a, s) = \max_k \mathcal{H}^x_{\mathsf{acq}}(a, s, k)$ (i.e., the maximum k satisfying (4.9)) and where $\mathcal{H}^{\varnothing}_{\mathsf{acq}}(a, s, k) = ask$ is the payoff for an acquisition attack if no alarm is available. The subtraction in (4.10) is included simply to provide an intuitive progression for efficiency: if $\mathcal{E}$ is greater for alarm A than alarm B, then A is a better alarm than B. Based on (4.10), we can calculate an efficiency of 0.14 for v, 0.0099 for h, 0.73 for c and 0.22 for TRW. The effectiveness of the combined alarm is 0.80.

The most interesting result from these calculations is the relatively low efficiency of r as an alarm. Recall from Figure 4.3 that r has a very high true positive rate. However, because the detection mechanism relies on attacker failures, it is better at detecting attacks which have a relatively low s. r detection is therefore very good at detecting attacks with low payoff.

We expect that the comparative efficiency of these alarms will differ from

one protocol to the next. v, c and h are affected by the aggregate traffic for one protocol, *e.g.*, the total number of hosts using a particular protocol. Conversely, r relies exclusively on per-host behavior. Consequently, using protocols with more clients or servers (such as HTTP) should result in less v and c efficiency, while r should have the same efficiency as on SSH.

4.4.3 Determining a Minimum False Positive Rate

As Figure 4.4 implies, even with all the detection mechanisms operating, attackers can still acquire a high rate of return with a sufficiently subtle hit-list attack. In this section, we will now address the question of detection from a different perspective: how high a false positive rate do we have to tolerate in order to prevent the attacker from seriously compromising the monitored network?

To do so, we invert (4.7) so that instead of calculating the attacker's payoff as a function of detectability, we calculate the probability of detection as a function of payoff. Solving for $\mathcal{P}^x_{\mathsf{det}}(a,s)$ in (4.7) yields

$$\mathcal{P}^x_{\mathsf{det}}(a,s) = 1 - \sqrt[k]{\frac{\mathcal{H}^x_{\mathsf{acq}}(a,s,k)}{ask}} \tag{4.11}$$

Suppose the defender wishes to minimize $\mathcal{P}^x_{\mathsf{det}}(a,s)$ (and hence also the false alarm rate) while restricting $\mathcal{H}^x_{\mathsf{acq}}(a,s,k) \leq 1$, and so the attacker wishes to maximize $\mathcal{P}^x_{\mathsf{det}}(a,s)$ in order to achieve $\mathcal{H}^x_{\mathsf{acq}}(a,s,k) = 1$. The attacker does so by choosing k so as to minimize $(ask)^{-1/k}$, for any a and s.

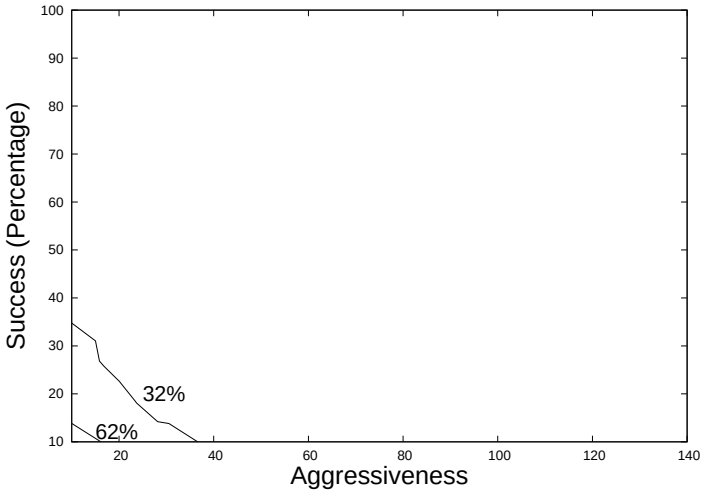
Using this strategy, we calculate the detection probability required to identify and stop attackers at points within the OAS. To calculate the result-

ing detection thresholds for each alarm, we use our simulated attacks with parameters a and s to calculate the threshold needed to filter off $\mathcal{P}^x_{\mathsf{det}}(a,s)$ of the attacks when overlaid on our training data.

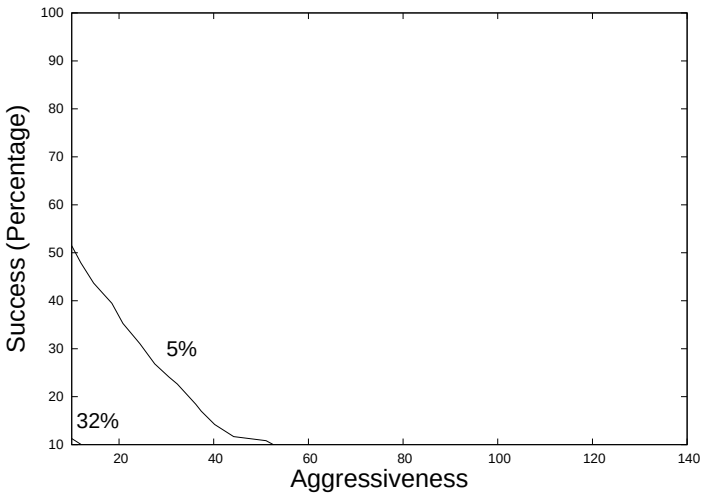
The results of these runs are given in Figure 4.5. These figures are contour plots over the OAS as before. However, the contours for the figure are the false positive rates that would result from this analysis. For the v, c and h detectors, these values are calculated using (4.2). For r this value is calculated by using (4.4).

[hpt] As Figure 4.5 indicates, anomaly detection systems that are capable of defending against subtle attacks will require extremely high false positive rates. Recall that our measurement system conducts a test every 30s; for every 1% false positive rate we accept, we pay 10 alerts per eight-hour shift. As such, this figure indicates that the false positive rates for building systems that can limit the attacker to $\mathcal{H}^x_{\mathsf{acq}}(a,s,k) \leq 1$ are much higher than we can consider accepting.

One way to avoid such high false positive rates would be to not place such a stringent limit of $\mathcal{H}^x_{\mathsf{acq}}(a,s,k) \leq 1$. For example, if the defender insists on a near-zero false positive rate, we can determine if there is a higher threshold for the payoff that can accommodate this rate, such as $\mathcal{H}^x_{\mathsf{acq}}(a,s,k) \leq 5$. Figure 4.6 shows this for the c alarm, for $\mathcal{H}^c_{\mathsf{acq}}(a,s,k) \in \{5,10,15\}$. Specifically, each contour line shows the values of a and s for which $\mathcal{H}^c_{\mathsf{acq}}(a,s,k)$ can be limited to at most the specified value, using a threshold $\theta_c = \mu_C + 3.5\sigma_C$, which is large enough to ensure a false positive rate very close to zero. As this figure shows, the defender can effectively impose an upper limit on the attacker's payoff, but unfortunately this limit

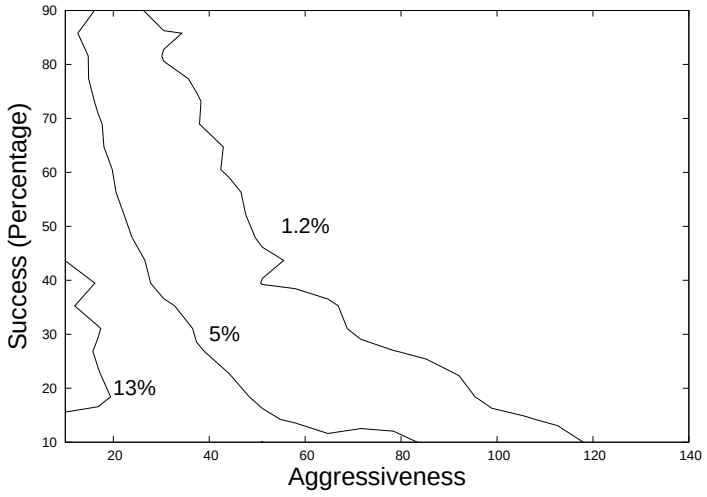


(a) Total graph size v

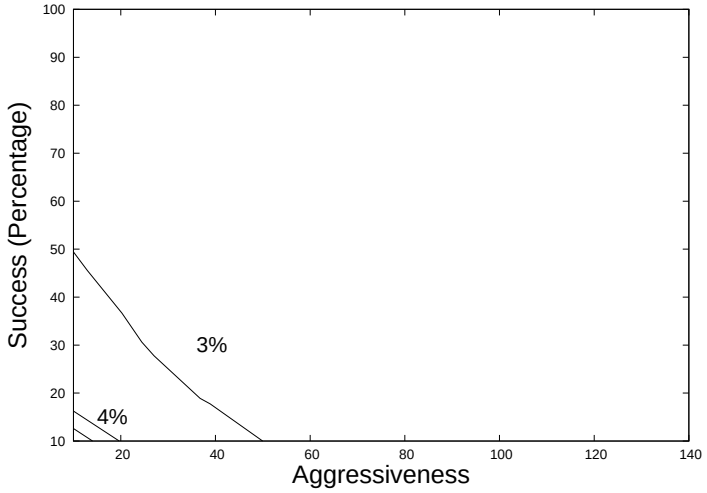


(b) Largest component size c

Figure 4.5: False positive rates required to limit expected hosts compromised to 1.



(c) Server entropy h



(d) Maximum failed connection run r

Figure 4.5 continued.

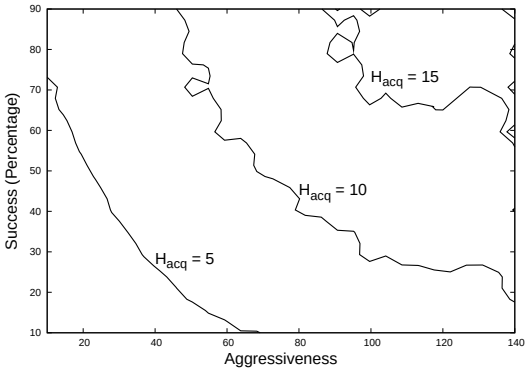


Figure 4.6: Values of a and s for which $\mathcal{H}^c_{\mathsf{acq}}(a, s, k)$ can be limited to at most the specified value, using a threshold $\theta_c = \mu_C + 3.5\sigma_C$.

must be rather large ($\mathcal{H}^c_{\mathsf{acq}}(a, s, k) = 15$) in order to cover the majority of the attack space.

From Figures 4.5 and 4.6, we conclude that in order for an anomaly detection system to be a viable deterrent to host compromise, it must either use finer resolution data than NetFlow, or it must develop mechanisms for coping with a high false positive rate. These mechanisms may include an explicit upper limit on what a NIDS can identify, or host-based intrusion detection compensating for the limits of the NIDS.

4.5 Modeling Reconnaissance

In this section, we develop an alternative attack scenario based around a different measure of value. In this scenario, to which we refer as *reconnaissance*, the attacker scouts out the network with his bots. In each attack, he communicates with addresses to simply determine the presence of hosts at

certain addresses. The reconnaissance scenario differs from the acquisition scenario by the attacker's knowledge and goals. Specifically, the attacker's goal is to contact as many addresses as possible within a short period. To do so, the attacker uses a *chaff hit list* consisting of hosts that the attacker already knows about, and a target space of addresses to probe. The chaff hit list reduces the attacker's probability of detection by lowering his failure rate. However, it also reduces the attacker's payoff by requiring him to "sacrifice" a certain number of targets every round.

Let $\mathsf{alarmed} = i$ be the event that the bot is detected at the end of attempt i (and before attempt $i + 1$); as before, an attempt is comprised of contacting a addresses with success rate (in this case, owing to the chaff hit list) of s. Let $\mathsf{scanned}$ denote a random variable indicating the number of scans that one bot performs successfully (i.e., determines whether the scanned address has a listening service or not), not counting the "chaff" that it introduces to avoid detection. Note that we suppose that the number of listening services the bot finds is sufficiently small that it does not relieve the bot from introducing a fraction s of chaff scans. We also presume that the probability the bot is detected in each attempt is independent.

$$
\begin{aligned}
\mathcal{H}^x_{\mathsf{rec}}(a,s) &= \mathbb{E}\left[\mathsf{scanned}\right] \\
&= \sum_{i=1}^{\infty} \mathbb{P}\left[\mathsf{alarmed} = i\right] \mathbb{E}\left[\mathsf{scanned} \mid \mathsf{alarmed} = i\right] \\
&= \sum_{i=1}^{\infty} \left((1 - \mathcal{P}^x_{\mathsf{det}}(a,s))^{i-1} \mathcal{P}^x_{\mathsf{det}}(a,s)\right) (ia(1-s)) \\
&= a(1-s) \frac{\mathcal{P}^x_{\mathsf{det}}(a,s)}{1 - \mathcal{P}^x_{\mathsf{det}}(a,s)} \sum_{i=1}^{\infty} i(1 - \mathcal{P}^x_{\mathsf{det}}(a,s))^i \\
&= a(1-s) \frac{\mathcal{P}^x_{\mathsf{det}}(a,s)}{1 - \mathcal{P}^x_{\mathsf{det}}(a,s)} \frac{1 - \mathcal{P}^x_{\mathsf{det}}(a,s)}{\mathcal{P}^x_{\mathsf{det}}(a,s)^2} \\
&= a(1-s) \frac{1}{\mathcal{P}^x_{\mathsf{det}}(a,s)} \qquad (4.12)
\end{aligned}
$$

Applying (4.12) to the detection matrix over our OAS results in the payoff plot shown in Figure 4.7. This figure plots the aggregate payoff over the OAS for reconnaissance. Of particular note with this result is that it demonstrates that a sufficiently motivated and subtle attacker can scan a network by subtly exploiting attacks with high s rates. In this case, the attacker can slowly scan the network for an extended period — the observed peak at the $a = 20$ segment of the graph implies that the attacker scans for 25 minutes before being detected.

However, the attacker can achieve just as effective results by aggressively scanning the network. Recall that the effective aggressiveness of the attacker is bound by θ_d to less than 150 nodes. In the reconnaissance scenario, the attacker faces no penalty for scanning at a higher aggressiveness rate, since the defender can only block an address. Consequently, this plot can continue out to whatever the practical upper limit for a is, a result which would

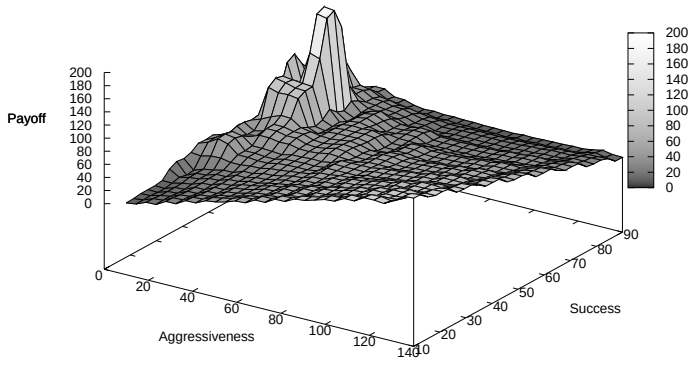


Figure 4.7: Plot of the payoff for reconnaissance attacks over the OAS.

correspond to the scanning we observe right now.

4.6 Conclusion

In this chapter we have developed a new method for evaluating the performance of alarms based on an observable attack space, specifically the view of a harvesting or scanning attack that is available in flow logs that lack payload data. Our approach complements ROC-based analysis by enabling the creation of detection surfaces — models of an alarm's ability to detect different attacks. Moreover, we augment this analysis with a payoff-based metric. By incorporating payoffs, we are better able to characterize the deterrence offered by an alarm. In particular, instead of describing the detection of a system in terms of pure false positive and false negative rates,

we are able to use payoff functions to calculate the gain that an attacker can expect from a certain type of attack. This also enables us to determine how high a false positive rate we must endure in order to limit the attacker's payoff to a target value.

Several useful and, in some cases, discouraging results fall out of our analysis techniques as applied to SSH traffic observed on a very large network. For example, in §4.3.3 our analysis elucidated the complementary capabilities of detection using the size c of the largest component of a protocol graph [13] and the TRW scan detector r [26]. Consequently, there is good reason to use both simultaneously. Moreover, we showed that these detectors significantly outperform the server address entropy detector h, the graph-size detector v, and the degree-based detector d, for the stealthy attacks that form our observable attack space. That said, using our payoff analysis for acquisition attacks, we showed in §4.4.2 that r detection is primarily effective at detecting acquisition attacks with low payoff for the attacker, and so its utility for acquisition attacks is less compelling. In addition, we showed in §4.4.3 that in order to severely limit the attacker's acquisitions, the false positive rates that would need to be endured by any of the detectors we considered would be significant and, for a network of the size we studied, impractical. Consequently, we showed how to derive the payoff limits that would enable near-zero false positive rates for an alarm.

Chapter 5

Conclusions

"Am I being scanned?"

–Querying on this string to Google groups returns 123,000 messages

In this dissertation, we have developed a novel anomaly detection method which uses protocol graphs to identify subtle attacks. By using a two-stage filtering method, we have been able to train this system on heavily polluted traffic, in particular SSH logs. Finally, we have developed a new evaluation approach to compare detection methods, and using this approach we have shown how our detection method compares to other scan and harvesting detection methods.

Several open research directions are evident from this work. As originally intended, this dissertation would have focused more extensively on the use of protocol graphs to describe traffic. However, the exigencies of working on a live dataset led to problems in making *any* anomaly detection system

viable. As a consequence, there are a large body of graph-based description problems left to be addressed. The anomaly detection methodology of Chapter 2 is the beginning of an integrated approach to anomaly detection that uses graph attributes both for detection and attacker identification. A richer vocabulary of graph attributes is available, and we have already identified several qualities that may result in deeper insights into attackers and normal behavior.

The most interesting problem that this dissertation raises, however, is about evaluating and reacting to attacks. As Chapter 3 shows, while we can remove attacks to train the IDS, the fact remains that the network is attacked (more precisely, *scanned*) continuously. The majority of attacks we observe in our dataset our failures — while the system *is* being attacked, there is little compelling evidence that the attacking IP addresses actually communicated with anyone within the observed network, and consequently these attacks may warrant no reaction.

In particular, our work in Chapter 4 treats the problem of IDS construction as a design challenge to the attacker — a question that must be extended out more thoroughly. Our initial work shows that simple IDS will likely remain insufficient, at least at the scale of networks we have observed. However, further work must be done to model attacker choices, to see if there are other methods an attacker can exploit to systematically take over a network.

In this thesis, we modeled attackers who treat all IP addresses as either known (on the hit list) or unknown (off of the list). However, there are other models of attacker knowledge which are critical for future eval-

uations. Weaver *et al.* [66] outline a variety of means by which a worm can spread outside of blind scanning, including hit-list and topological attacks. Topological attacks are a particularly interesting threat to protocol graph detection, because in this scenario a bot attacks its neighbors, rather than blindly scanning. These attacks would therefore reflect the same social structures that protocol graphs model.

The attackers we have observed in this work benefit from an industrialized technology. At the time of this writing, botnet technology has been available for approximately a decade, and attackers have developed new attacks and methods based around their ability to expend hundreds, if not thousands of hosts in order to achieve their goals. As a consequence, the question we now face is not whether we are being attacked — we are, continuously — but what we can do to mitigate the impact of these attacks.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1975.

[2] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*, pages 171–180, 2000.

[3] E. Alata, V. Nicomette, M. Kaaniche, M. Dacier, and M. Herrb. Lessons learned from the deployment of a high-interaction honeypot. In *Proceedings of the 2006 European Dependable Computing Conference*, 2006.

[4] S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis. Defending against hitlist worms using network address space randomization. In *WORM '05: Proceedings of the 2005 ACM Workshop on Rapid Malcode*, pages 30–40, 2005.

[5] S. Axelsson. The base rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security*, 3(3):186–205, 2000.

[6] D. Barbará, J. Couto, S. Jajodia, and N. Wu. ADAM: a testbed for ex-

ploring the use of data mining in intrusion detection. *Record of the ACM Special Interest Group on Management of Data (SIGMOD)*, 30(4):15–24, 2001.

[7] D. Barbará, Y. Li, J. Couto, J. Lin, and S. Jajodia. Bootstrapping a data mining intrusion detection system. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, 2003.

[8] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *Proceedings of the 9th World Wide Web Conference*, pages 309–320, 2000.

[9] A. Cárdenas, J. Baras, and K. Seamon. A framework for evaluation of intrusion detection systems. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[10] S. Chen and Y. Tang. Slowing down Internet worms. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 312–319, March 2004.

[11] K. Claffy, H. Braun, and G. Polyzos. A parameterizable methodology for internet traffic flow profiling. *IEEE Journal of Selected Areas in Communications*, 13(8):1481–1494, 1995.

[12] M. Collins, C. Gates, and G. Kataria. A model for opportunistic network exploits: The case of P2P worms. In *Proceedings of the 2006 Workshop on Economics and Information Security*, 2006.

[13] M. P. Collins and M. K. Reiter. Hit-list worm detection and bot iden-

tification in large networks using protocol graphs. In *Proceedings of the 2007 Symposium on Recent Advances in Intrusion Detection*, 2007.

[14] M. P. Collins and M. K. Reiter. Anomaly detection amidst constant anomalies: Training IDS on constantly attacked data. Technical Report CMU-CYLAB-08-006, Carnegie Mellon University, CyLab, 2008.

[15] D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[16] D. Ellis, J. Aiken, A. McLeod, D. Keppler, and P. Amman. Graph-based worm detection on operational enterprise networks. Technical Report MTR-06W0000035, MITRE Corporation, April 2006.

[17] L. Ertoz, E. Eilertson, A. Lazarevic, P. Tan, J. Srivastava, V. Kumar, and P. Dokas. *Next Generation Data Mining*, chapter 3. MIT Press, 2004.

[18] F. Freiling, T. Holz, and G. Wicherski. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In *Proceedings of the 2005 European Symposium on Research in Computer Security*, 2005.

[19] J. Gaffney and J. Ulvila. Evaluation of intrusion detectors: a decision theory approach. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[20] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23:319–344, September 1991.

[21] V. Garg, V. Yegneswaran, and P. Barford. Improving NIDS performance through hardware-based connection filtering. In *Proceedings of the 2006 International Conference on Communications*, 2006.

[22] C. Gates, J. McNutt, J. Kadane, and M. Kellner. Detecting scans at the ISP level. Technical Report SEI-2006-TR-005, Software Engineering Institute, 2006.

[23] C. Gates, J. McNutt, J. Kadane, and M. Kellner. Scan detection on very large networks using logistic regression modeling. In *Proceedings of the IEEE Symposium on Computers and Communications*, June 2006.

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

[25] G. Gu, M. Sharif, X. Qin, D. Dagon, W. Lee, and G. Riley. Worm detection, early warning and response based on local victim information. In *Proceedings of the 2004 Annual Computer Security Applications Conference*, 2004.

[26] 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.

[27] R. Maxion K. Killourhy and K. Tan. A defense-centric taxonomy based on attack manifestations. In *Proceedings of the 2004 Conference on Dependable Systems and Networks (DSN)*, 2004.

[28] M. Kang, J. Cabllero, and D. Song. Distributed evasive scan techniques

and countermeasures. In *Proceedings of the 2007 Conference on the Detection of Intrusions, Malware and Vulnerability Assessment*, 2007.

[29] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: multilevel traffic classification in the dark. In *Proceedings of ACM SIGCOMM*, pages 229–240, 2005.

[30] E. Kreyszig. *Advanced Engineering Mathematics, 9th Edition*. J. Wiley and Sons, 2005.

[31] A. Kumar, V. Paxson, and N. Weaver. Exploiting underlying structure for detailed reconstruction of an Internet scale event. In *Proceedings of the ACM Internet Measurement Conference*, October 2005.

[32] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. In *Proceedings of ACM SIGCOMM*, pages 217–228, 2005.

[33] W. Lee, S. Stolfo, P. Chan, E. Eskin, W. Fan, M. Miller, and J. Hershkop, S.and Zhang. Real time data mining-based intrusion detection. In *Proceedings of the 2001 DARPA Information Survivability Conference and Exposition*, 2001.

[34] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham, and M. Zissman. Evaluating intrusion detection systems: the 1998 DARPA off-line intrusion detection evaluation. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, 2000.

[35] T. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalali, and P. Neu-

man. A real-time intrusion-detection expert system (IDES). Technical Report Project 6784, CSL, SRI International, 1992.

[36] M. Mahoney and P. Chan. Learning rules for anomaly detection of hostile network trafic. In *Proceedings of the 2003 IEEE International Conference on Data Mining*, 2003.

[37] R. Maxion and K. Tan. Benchmarking anomaly-based detection systems. In *Proceedings of the 2000 Conference on Dependable Systems and Networks (DSN)*, 2000.

[38] J. McHugh. Testing intrusion detection systems: a critique of the 1998 and 1998 DARPA intrusion detection system evaluations as performed by lincoln laboratory. *IEEE Transactions on Information and Systems Security*, 3(4):262–294, 2000.

[39] J. Mirkovic and P. Reiher. D-ward: A source-end defense against flooding denial-of-service attacks. *IEEE Transactions on Dependable and Secure Computing*, 2(3):216–232, 2005.

[40] D. Moore, C. Shannon, G. Voelker, and S. Savage. Network telescopes. Technical report, CAIDA, 2003.

[41] T. Moore and R. Clayton. Examining the impact of website take-down or phishing. In *Proceedings of the 2007 eCrime Researchers' Summit*, 2007.

[42] S. Northcutt. *Network Intrusion Detection: An Analyst's Handbook*. New Riders, 1999.

[43] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson.

Characteristics of internet background radiation. In *Proceedings of the 2004 Internet Measurement Conference*, 2004.

[44] V. Paxson. Bro: A system for detection network intruders in real time. In *Proceedings of the 2008 Usenix Security Symposium*, 1998.

[45] K. Pentikousis and H. Badr. Quantifying the deployment of TCP options, a comparative study. *IEEE Communications Letters*, 8(10):647–649, 2004.

[46] P. Porras and P. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*. NIST, 1997.

[47] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. A measurement study of the BitTorrent peer-to-peer file-sharing system. Technical Report PDS-2004-007, Delft University of Technology, April 2004.

[48] A. Ramachandran, N. Feamster, and D. Dagon. Revealing botnet membership using DNSBL counter-intelligence. In *Proceedings of the 2006 USENIX Workshop on Steps for Reducing Unwanted Traffic on the Internet (SRUTI)*, 2006.

[49] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing*, 6(1), 2002.

[50] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002*, 2002.

[51] S. Schechter, J. Jung, and A. Berger. Fast detection of scanning worm infections. In *7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 59–81, September 2004.

[52] V. Sekar, Y. Xie, M. K. Reiter, and H. Zhang. A multi-resolution approach to worm detection and containment. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks*, pages 189–198, June 2006.

[53] U. Shankar and V. Paxson. Active mapping: Resisting NIDS evasion without altering traffic. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 44. IEEE Computer Society, 2003.

[54] C. Shannon and D. Moore. The spread of the Witty worm. *IEEE Security and Privacy*, 2(4):46–50, 2004.

[55] S. Shapiro and M. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52:591–611, 1965.

[56] A. Soule, K. Salamatian, and N. Taft. Combining filtering and statistical methods for anomaly detection. In *Proceedings of the 2005 Internet Measurement Conference*, 2005.

[57] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, August 2002.

[58] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS – A graph-based intrusion detection system for large networks. In *Proceed-*

ings of the 19th National Information Systems Security Conference, pages 361–370, 1996.

[59] R. Stevens. *TCP/IP Illustrated*. Addison Wesley Longman, Inc., 1994.

[60] S. Stolfo, W. Fan, Wl. Lee, A. Prodromidis, and P. Chan. Cost-based modeling for fraud and intrusion detection: Results from the jam project. In *Proceedings of the 2000 DARPA Information Survivability Conference and Exposition*, 2000.

[61] S. J. Stolfo, S. Hershkop, C. Hu, W. Li, O. Nimeskern, and K. Wang. Behavior-based modeling and its application to email analysis. *ACM Transactions on Internet Technology*, 6(2):187–221, May 2006.

[62] K. Tan and R. Maxion. The effects of algorithmic diversity on anomaly detector performance. In *Proceedings of the 2005 Conference on Dependable Systems and Networks (DSN)*, 2005.

[63] R. E. Tarjan. *Data Structures in Network Algorithms*, volume 44 of *Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, 1983.

[64] J. Twycross and M. W. Williamson. Implementing and testing a virus throttle. In *Proceedings of the 12th USENIX Security Symposium*, pages 285–294, August 2003.

[65] S. Venkataraman, J. Caballero, D. Song, A. Blum, and J.Yates. Black box anomaly detection: is it utopian? In *Proceedings of the 2006 HotNets Workshop*, 2006.

[66] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A taxonomy

of computer worms. In *Proceedings of the 2003 Workshop on Rapid Malcode (WORM)*, 2003.

[67] N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *Proceedings of the 2004 USENIX Security Symposium*, 2004.

[68] S. Webster, R. Lippmann, and M. Zissman. Experience using active and passive mapping for network situational awareness. In *Proceedings of the 2006 IEEE International Symposium on Network Computing and Applications*, 2006.

[69] J. Wu, S. Vangala, L. Gao, and K. Kwiat. An efficient architecture and algorithm for detecting worms with various scan techniques. In *Proceedings of the 2004 Network and Distributed Systems Security Symposium*, 2004.

[70] V. Yegneswaran, P. Barford, and J. Ullrich. Internet intrusions: Global characteristics and prevalence. In *Proceedings of the 2003 ACM SIGMETRICS Conference*, 2003.

[71] C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and early warning for Internet worms. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 190–199, 2003.