On the Use of Context in Network Intrusion Detection Systems

Jayanth Kumar Kannan

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-110
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-110.html

August 9, 2009

On The Use of Context in Network Intrusion Detection Systems

by

Jayanthkumar Kannan

B.Tech. (Indian Institute of Technology, Madras) 2002
M.S. (University of California, Berkeley) 2007

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Ion Stoica, Chair
Professor Scott Shenker
Professor Vern Paxson
Professor John Chuang

Fall 2009

The dissertation of Jayanthkumar Kannan is approved.

Chair	Date
	Date
	Date
	Date

University of California, Berkeley

On The Use of Context in Network Intrusion Detection Systems

by

Jayanthkumar Kannan

Abstract

On The Use of Context in Network Intrusion Detection Systems

by

Jayanthkumar Kannan

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

This thesis examines frameworks and mechanisms for building network intrusion detection systems. These systems perform a variety of complex analysis in order to enforce security policies, and such enforcement requires contextual information from several sources. In this thesis, we examine three such sources of context. First, we propose semi-automatic mechanisms that can be used in order to understand how application traffic manifests in the network; such mechanisms are necessary to incorporate application semantics into security policy enforcement. Second, we analyze the effectiveness of information exchange amongst multiple sites in containing a fast spreading worm. Third, we propose a framework that helps a network security system gain access to encryped network traffic that is typically decipherable only by the end-host, while at the same time, respecting confidentiality constraints on sensitive content embedded in network traffic.

Professor Ion Stoica
Dissertation Committee Chair

Contents

Contents **i**

List of Figures **iv**

List of Tables **v**

1 Introduction **1**

1.1 Need for Context 3

1.2 Sources of Context 4

1.3 Contributions and Summary of results in the thesis 4

2 Tools for Session Context Analysis **8**

2.1 Session Structure Inference: A Bird's-Eye View 9

2.2 Background 11

2.3 Related Work 13

2.4 Session Extraction 15

2.4.1 Types of Sessions 15

2.4.2 Extracting Homogeneous Sessions 16

2.4.3 Extracting Mixed Sessions 16

2.4.4 Discussion 23

2.5 Structure Abstraction 24

2.5.1 Representation of Session Descriptors 24

2.5.2 Abstraction Framework 26

2.5.3 Coverage Phase 27

2.5.4 Generalization Phase 28

2.5.5 Coverage Curves 29

2.6 Results . . . 30
2.6.1 Parameter Settings . . . 30
2.6.2 Empirically Observed Session Structure . . . 33
2.6.3 Finding Attacks Using Anomaly Detection . . . 38
2.7 Conclusion . . . 39

3 Cooperative Worm Containment **40**
3.1 Problem Setting and Methodology . . . 42
3.1.1 Evaluation Methodology . . . 43
3.2 Modeling Cooperation . . . 43
3.2.1 Discussion . . . 47
3.3 Cost Of Decentralization . . . 47
3.3.1 Parameters . . . 47
3.3.2 Analyzing Detection and Filtering . . . 49
3.3.3 Analyzing Signaling . . . 51
3.3.4 Numerical Solutions . . . 55
3.4 Cost Of Partial Deployment . . . 58
3.4.1 Baseline Solution . . . 59
3.4.2 Rerouting . . . 59
3.4.3 Numerical Results . . . 62
3.5 Cost Of Malice . . . 62
3.5.1 Analyzing Cooperation under Malice . . . 64
3.5.2 Numerical Results . . . 66
3.6 Limitations of Modeling . . . 66
3.7 Related Work . . . 70
3.8 Conclusion . . . 72

4 Hiding Context from Admins **73**
4.1 Introduction . . . 73
4.2 Three Motivating Scenarios . . . 79
4.3 Design Goals . . . 82
4.4 Design . . . 84
4.4.1 Off-Path Architecture . . . 85
4.4.2 Host Software: Architecture . . . 86

4.5 Implementation . . . 93
4.5.1 DC Layer . . . 93
4.5.2 Password Obfuscator . . . 94
4.5.3 Interaction with Honeypot . . . 96
4.6 Security Analysis of Cloak . . . 97
4.6.1 Confidentiality Guarantee . . . 99
4.6.2 Fidelity Evaluation: Known Attacks . . . 100
4.6.3 Fidelity Evaluation: Qualitative Analysis For Unknown Attacks . . . 102
4.6.4 Attacks against Cloak . . . 108
4.7 Performance Evaluation . . . 109
4.8 Limitations . . . 111
4.9 Related Work . . . 112
4.10 Conclusion . . . 114

5 Conclusion and Future Work **115**
5.1 Tools for Session Structure Inference: . . . 115
5.2 On the Use of Global Context for Cooperative Worm Containment . . . 116
5.3 Off-path Architecture for Intrusion Prevention . . . 117
5.4 Future Work . . . 117

Bibliography **119**

List of Figures

2.1 The Exact DFA for FTP 24

2.2 Abstract Session Descriptor for FTP 25

2.3 Overview of Structure Abstraction Framework 26

2.4 **FTP** Coverage Curve (A), DFAs: 4 edges (B), 4 edges (C), 6 edges (D), 8 edges (E), 10 edges (F), 18 edges (G) 32

2.5 **Timbuktu** Coverage Curve (A) and DFAs: 4 edges (B), 18 edges (C) . . . 34

2.6 **HTTP** DFA with 30 edges 34

2.7 Session Illustrating Anomalous Behavior 37

3.1 (a) Decentralized Cooperation (b) State Transition Diagram 44

3.2 (a) Temporal Dynamics: $\lambda = 2.0$ (with implicit signaling) (b) Containment Vs. Size of vulnerable population (c) Containment Vs. Scanning Rate (d) Containment Vs. Number of Initial Seeds (e) Containment Vs. Time to detection (f) Containment Vs. Rate of Explicit Signaling 56

3.3 (a) Rerouting Mechanism (b) Containment in the Partial Deployment Scenario (b) Containment at very low deployment levels 61

3.4 (a) Containment Vs. Threshold T (b) Containment Vs. Non-Uniformity (c) Containment Vs. Network Delays 65

3.5 (a) Containment Vs. Constant time to detection (b) Containment Vs. Time To Detection (realistic environment) 67

4.1 Apache HTTPS server. 75

4.2 Cloak. 85

4.3 Host Software Architecture of Cloak. 87

List of Tables

3.1 Default Setting of Parameters in our Model 48

Chapter 1

Introduction

There are a wide range of security mechanisms implemented in today's networks in order to achieve a variety of security objectives. These objectives include preventing the leakage of sensitive data from within the enterprise, controlling which applications are allowed to send traffic over the network, avoiding the compromise of internal machines by malicious exploits, and containing already compromised internal machines.

In all these cases, the security mechanism requires a decision to be made with respect to a single network packet: should it be allowed or not?

However, this decision generally requires significant *context* to accurately infer the intent of the packet. By context, we refer to any information required by the mechanism to make the decision apart from the current packet itself. For instance, in order to filter out known exploits, the administrator typically specifies a signature as a sequence of packets representing the exploit. Implementing the signature matching correctly in this case requires maintaining a history of packets seen in the past.

This thesis proposes the following mechanisms and architectures to help infer such context from various sources:

- One important source of information about the intent of a particular network packet

comes from connections that relate to the same high-level task as the packet at hand. For example, in order to identify the first packet on a FTP data transfer connection as a data transfer packet, it is necessary to know the context setup on the control connection preceding the data transfer itself. We propose *semi-automatic mechanisms* for inferring such *connection-level structure* that manifest in network traffic. By semi-automatic, we mean that the mechanism is guided by a human in its inference process (such human input is necessary for reliable results).

- One way to help deal with the threat of fast zero-day worms is to obtain *global context* about anomalous network activity at multiple Internet sites. We use analytical modeling to discuss fundamental limitations in the exchange of such global context across multiple sites in dealing with unknown fast worm attacks.

- The downside of security mechanisms relying on sophisticated packet inspection to gain context is that sensitive data embedded in network packets may be exposed to those in charge of the security appliance. For instance, passwords used for authenticating users are often present in network traffic. We discuss an *off-path architecture* for deploying network security appliances that can be used to prevent leakage of such sensitive data in a flexible fashion. The main difference from the current architecture is that the security appliance is moved away from the network path of incoming and outgoing packets, and that software running at the host is responsible for conveying packets to the security appliance.

The rest of this chapter is organized as follows. First, we discuss the wide variety of security objectives that an administrator wishes to achieve, and discuss how knowledge of context is useful in implementing these objectives. Then, we will discuss the specific contributions made by this thesis towards providing various forms of such context.

1.1 Need for Context

The traditional objective of security mechanisms is to control access to confidential information using computer programs. In the context of the network, one way to help achieve this objective of *preventing the leakage of sensitive information* is by *application-level access control*. This limits the set of services that an user is allowed to access over the network. This mechanism is achieved by configuring rules at a firewall that list which connections should be permitted.

For implementing access control correctly, a great deal of context is required to make the decision of whether to admit a particular packet. For example, if the administrator wishes to permit access to FTP servers, an incoming TCP SYN packet should be allowed if it corresponds to a transfer admitted by the protocol, and otherwise rejected. This requires context about the other packets preceding the current packet on which the decision has to be made.

A more recent security objective is to prevent the *mis-use* of internal computer resources (bandwidth, compute time) for various activities (such as sending spam, compromising other machines, mounting DDoS attacks). This objective has become necessary given the fact that it is possible for a remote untrusted party to easily compromise internal computers over the Internet. Further, such mis-use can lead to liability issues as well if internal computers are used to compromise remote machines belonging to other organizations. Such misuse is prevented by dropping packets that may possible lead to an internal machine being exploited. Another mechanism may limit the packets sent by a compromised host. In either case, significant amount of context is typically required in making this decision. These mechanisms require analysis to determine the exact protocol state corresponding to the communication, and to determine whether any deviation from the expected state is anomalous.

1.2 Sources of Context

Both the kinds of security mechanisms discussed above are most useful when implemented in an online fashion. This requires the security mechanism to make its decision on a per-packet basis: would allowing this packet violate the particular policy that the administrator wishes to achieve?

However, even for the simplest of these functionality, access control, a great deal of context is required to make this decision accurately. More generally, the nature of the context required can be classified into three primary sources:

- **Intra-connection Context:** This context relates to packets seen on the connection before the current packet. For instance, this context helps determine the application involved and the particulars of the intended access (*e.g.,* the URL accessed, or an attachment to an email). This context is usually achieved by relying on deep packet inspection; this refers to technologies that analyze the packet at the application layer using protocol knowledge.

- **Inter-connection Context:** This context relates to connections seen before the current connection that help infer the intent of the particular packet. For example, the port number negotiated over the FTP protocol helps identify the connection attempt as an attempt to transfer the file.

- **Global context:** This context relates to packets seen at other locations in the network that may offer clues to the intent of the current packet. This is typically most useful for detecting large scale compromises (such as botnets or worms) where similar anomalous activity is observed at several Internet locations.

1.3 Contributions and Summary of results in the thesis

- **Tools for Session Structure Inference:** To reduce the overhead of reverse-

engineering of inter-connection context, this thesis discusses semi-automatic mechanisms that can mine a trace of network connection logs to infer *session* structure.

A session structure refers to a sequence of connections that follow each other in succession and typically reflect one single intent. For instance, a browsing session over HTTP may be followed by a FTP control connection and a sequence of FTP data transfers.

It is important for network devices to be able to recognize such session structures in application traffic since various network decisions typically apply at the granularity of sessions, not individual connections. However, it is increasingly difficult for devices to keep up with the huge variety of Internet applications; typically it takes constant reverse-engineering effort on the part of the programmers of these network devices to ensure that the recognition is up to date.

Our mechanisms helps an analyst mine such structures out of a trace using causality tracking mechanisms that rely on timing information. We do not require any application-specific knowledge; the analyst can apply our tool to a network trace and simply confirm the detected session structures. Given the variety of network devices that need to recognize application session patterns (such as NATs, application-level firewalls, QoS mechanisms), we expect our mechanisms will be widely useful to programmers of network devices in deciphering inter-connection context.

- **Cooperative Worm Containment:** The advantage in obtaining global context to combat unknown worm attacks is the visibility across a large number of networks. Such cooperative containment is becoming increasing useful for two reasons.

  First, due to the imperfect and error-prone anomaly detection techniques that are used to identify worms, it is beneficial for sites to cooperate with one another to ensure that worm attacks are neither missed nor falsely signalled. Second, such cooperation is useful since a site can infer a worm attack before its own site is infected. Thus it can prevent the infection of any internal machines by installing filters on incoming traffic.

Despite the usefulness of cooperative containment, its effectiveness is not fully understood; most existing literature has analyzed the effectiveness in conjunction with specific schemes for detection and cooperation. We aim to remedy this by studying the containment offered by cooperation by modeling the detection and information exchange schemes as black-boxes. Our study is based on analytical techniques and simulations that offer a complete understanding of cooperative worm containment and the regime under which it can perform well.

- **Off-path Architecture for Intrusion Prevention:** In the typical deployment of a intrusion prevention device as a network-level middle-box, network traffic is typically exposed completely to the administrator in the control of the middle-box. This entails significant risk since sensitive information embedded in network traffic is exposed to admins who may not necessarily be trustworthy. In order to address this issue, we propose a novel off-path architecture that relies on host software to furnish a desensitized version of network traffic to a in-network appliance.

  This architecture offers a flexible alternative to the all-or-nothing situation prevailing today. Today's intrusion detection architectures offer only two choices. In some networks, due to the sensitivity of the information in network traffic, security is completely foregone and security devices simply skip over the encrypted portion of the traffic. This exposes internal machines to attacks over these encrypted channels. On the other hand, some networks choose to place security above confidentiality, and simply disallow encryption; this vests tremendous trust in the administrator of the network device.

  Our architecture offers considerable flexibility as compared to the status quo. Networks can choose to trade-off between security and confidentiality at a finer granularity by specifying exactly which information should be considered sensitive. Thus, the portion of the network traffic which is not deemed sensitive can be analyzed by the security device, while rest can be kept secret from the administrator of the security device. Given the increasing need for end-to-end encryption to defend against eavesdroppers and the growing need for sanitizating incoming traffic, we believe our

architecture will be useful to achieve finer trade-offs between these two important requirements in today's networks.

Chapter 2

Tools for Session Context Analysis

We begin this chapter by explaining our notion of a session, and then explain how the knowledge of the typical session structures that manifest in network traffic may be useful for implementing network security mechanisms. We then discuss the design and implementation of a tool that helps an analyst mine such session structure descriptions from a connection trace.

We will use the term *session* to denote a group of connections associated with a single network task, where by "task" we mean the activity that emanates from an external event that serves as the *causal origin* of the connections. Thus, a "task" in this context can be quite abstract, such as a user deciding to process their email or conducting an interactive login to a remote host. It will however often be convenient or apt for us to refer to a session using the layer-7 notion of "application" as a proxy for these abstract tasks. For example, we will refer to an "FTP session" rather than a "user transfering files session" because from a network trace perspective what we directly see is an instance of a TCP port 21 connection (associated with the FTP control channel) and perhaps subsequent port 20 or ephemeral port connections (data transfers directed by the control channel).

An understanding of the qualitative session structures that manifest in network traffic may be useful for network administrators in two ways. First, the database of structures

that can manifest due to legitimate reasons offers a baseline against which any suspicious sessions can be matched to see if it could have arisen due to any typical network application usage. Some of our preliminary efforts in this direction indicate that with some help from a human analyst this might be possible. Second, such understanding also helps an administrator frame access control policies in terms of allowing TCP 4-tuples such that users in her network can use applications of their interest. Apart from the use of our tool in the security context, we believe that it would also be useful to researchers who may benefit from working with higher-level abstractions of network traffic. For researchers, these abstractions promise to aid with traffic characterization and monitoring and constructing source models for synthesizing network traffic.

We now present a bird's eye-view of our tool in the following section.

2.1 Session Structure Inference: A Bird's-Eye View

The aim of our algorithms is to mine a connection-level trace to derive such abstract descriptions of the session-structure for the different applications present in the trace. These algorithms work in general terms, without requiring any *a priori* knowledge about a particular type of application. The procedure operates in a *semi-automated* fashion: we aim to provide an analyst with high-quality information reflecting different possible abstractions (*descriptors*) of an application's session structure. We express these descriptors as regular expressions, or, equivalently, deterministic finite automata (DFAs), which capture the order, type, and directionality of the connections that comprise a session, but not their interarrival timing. The different abstractions provided trade off economy-of-expression versus more detailed fidelity to the observed data.

Application sessions manifesting in network traffic can have different basic natures. Some applications have an explicitly defined session structure such as FTP sessions that consist of a FTP control connection followed by a number of data transfer connections. Other sessions reflect a looser structure that arises from how end-user software, or the end user themselves, drives network access in order to perform a task. For example, for sessions

beginning with an SMTP connection, we find a plethora of additional activity reflecting the fact that transferring email can be invoked in a variety of contexts: by mail servers (some of which initiate Ident connections back when contacted with incoming mail), by mail clients (such as Thunderbird or Outlook, which may be configured to establish SMTP connections along with POP3/IMAP4 connections for retrieving email), or by users (who may read their mail and browse alongside, leading to HTTP connections). Still other sessions arise from hostile activity, some of which runs counter to the session structures exhibited by benign traffic. While we do not aim to identify these in a comprehensive fashion, we note that our structural inference can provide a baseline for applying anomaly detection techniques for detecting some forms of attacks.

Our method for discovering session structures uses as input a connection-level traffic trace. The method operates in two stages: *Session Extraction* and *Structure Abstraction*.

Session Extraction is a statistical algorithm that reduces a stream of connections down to a stream of sessions. We base it on modeling the temporal characteristics of session arrivals, using the important observation that, in comparison with connections from different (and hence independent) sessions, connections belonging to the same session tend to occur "close" to one another. An important limitation of our current work is that we only capture sessions between a single pair of hosts, whereas many forms of sessions (e.g., Web surfing) naturally include a local host's interactions with multiple remote hosts. In the development of our approach, we note at several points ways in which it might be generalized in this fashion.

Structure Abstraction operates on the set of sessions extracted for a given application and attempts to infer succinct session descriptors that capture (at varying levels of abstraction) the structure of sessions typically generated by the application. We aim for this step to provide both economy of description and insight by suggesting apt abstractions. To do so, the inference framework includes a set of generalization rules that simplify or transform the raw descriptions inferred directly from a trace. We present these generalized descriptors to the analyst in terms of *complexity-coverage curves*: the analyst can opt for complex representations for more precise coverage of the set of sessions empirically observed, or for simpler, more abstract representations that may better capture "the heart of the matter".

We found that often the curves exhibit inflection point "sweet spots" that mark qualitative transitions between adding complexity and gaining more coverage.

We evaluated our scheme using a month's trace collected at the border of the Lawerence Berkeley National Laboratory, which on average saw about 2,700,000 connections each day. We used the first half of the trace to develop and calibrate our methods, then applying them to infer descriptors for about 40 different applications in the second half of the trace. These include well-known ones such as content-transfer (*e.g.,* SMTP, FTP, HTTP), remote access (*e.g.,* SSH, Telnet, remote exec), database (*e.g.,* OracleSQL, MySQL), peer-to-peer (*e.g.,* BitTorrent), and mapping and authentication (*e.g.,* RPC portmapper, LDAP, Kerberos). We also encountered applications little discussed in the literature: remote desktop (*e.g.,* Timbuktu, Groove), engineering/scientific (*e.g.,* ProEngineer, Legato, GridFTP), and lesser-known peer-to-peer (*e.g.,* KCEasy). When possible, we validated our inferences using published protocol specifications, but for applications whose public documentation does not specify an overall session structure, often we could only assess the plausibility of the inferences based on what we could determine about the application's operation.

The outline for the rest of the chapter is as follows. We begin with some background in Section 2.2 and discuss related work in Section 2.3. We develop our session extraction mechanism in Section 2.4 and our structure abstraction mechanism in Section 2.5. In Section 2.6 we present an evaluation of these mechanisms along with some preliminary results on detecting anomalous traffic. We then conclude in Section 2.7.

2.2 Background

We first frame the problem setting by specifying the inputs processed by our algorithms and the terminology we will use in their development.

Traffic Characteristics Inputs. Our algorithm works using connection-level information obtained from passive network monitoring. While in principle our approach could work applied to monitoring inside a site or within a backbone, our development has been

in the context of traces recorded at a site's border. For every TCP connection, the monitor records the IP addresses of the local and remote hosts, direction (incoming/outgoing), timing information (start time, duration), and connection status (whether successfully established). One could generate similar information for non-TCP flows, but to date we have evaluated our mechanism only for TCP traffic, using traces of SYN, FIN and RST packets.

Terminology. We denote a connection C by the tuple (*proto, dir, remote-host, local-host, start-time, duration*). *proto* specifies the service associated with the destination port X of the connection (*e.g.,* FTP or HTTP), or, if X does not have a specific service associated with it, we use "*priv-X*" for ports $X < 1024$ (usually only available to privileged processes) and "*other-X*" for $X \geq 1024$. For these latter, in some contexts we will use "*ephemeral*" to indicate that it routinely varies. *dir* takes the value "**in**" or "**out**" indicating the connection was incoming (initiated by the remote host) or outgoing (locally initiated). *remote-host/local-host* is the IP address of the remote/local host, and *start-time* and *duration* denote the beginning times and duration, respectively.

We define the *type* $T(C)$ of a connection C as the tuple (*proto, dir*). We allow some fields to be absent in our tuple notation for connections; the value of such omitted fields will be clear from context.

We define a *session* as a sequence of connections $S = (C_1, C_2, \ldots, C_n)$ that have a common causal origin. In this work, we only consider sessions that involve a single local-host and single remote-host, so implicitly all connections involve only these two hosts. We define the *application*, $A(S)$, associated with a session S as the type of the first connection C_1, *i.e.,* $T(C_1)$. This is imperfect since different applications might manifest with the same type of initial connection, or the same application with different initial connections, but in the absence of ground truth regarding the true user/application sessions in a traffic stream, it strikes us as arguably a reasonable approximation.

We say that a session S *belongs* to the *session type* $ST(S) = (T_1, \ldots, T_n)$, if for all $i \leq n$, $T_i = T(C_i)$. Thus, $A(S)$ categorizes which application a session S belongs to, while

$ST(S)$ captures the structure of the entire session in terms of the *proto*'s of each connection and their directionality.

DFA Visualization. In our DFA visualizations (produced by FSA utilities software [22] and Graphviz visualization software [23]), we label the start-state as 0, and distinguish between accepting states and non-accepting states with shading. We label edges with the type of the corresponding connection and a direction tag, "_in" or "_out" indicating incoming/outgoing connections respectively. Finally, we use the thickness of the edge to visually convey how often each edge occurred in the trace.

2.3 Related Work

The main area of previous work related to our effort consists of studies characterizing network flows at various granularities: packets, connections, and sessions.

Early Internet traffic studies focused primarily on the dynamics of individual packets in terms of comprising flows [26, 25, 12] and with regard to correlational structure [21], especially self-similarity [32]. Subsequent work then characterized the conditions encountered by streams of packets as they traversed the Internet [10, 50, 81]. These studies were accompanied by other early ones that analyzed traffic in terms of connection properties. These included measurements of connection characteristics as seen at different sites [18], dynamics seen within connections [36,45], transport behavior [49], structural contribution to self-similarity [72], connection performance limitations [80], and numerous characterizations of different applications (*e.g.,* [48, 15] for older ones).

Studies of both packet and connection dynamics have continued to expand into a large body of subsequent work. However, the literature examining application session structure in abstract terms (that is, not simply the structure of a particular application such as Web access or streaming audio/video) has been much more limited. Particularly relevant for our work is Paxson and Floyd's characterization of the connection-level and session-level behavior of applications [52]. This work studied the wide-area TCP arrival processes

at differing levels of granularity: packets, connections, and sessions. The authors found that *session* arrivals were generally well-modeled by a Poisson process with hourly rates, even though individual *connection* arrivals were not. More recently, Nuzman *et al.* [44] studied the arrival characteristics of HTTP connections, finding that when these connections are suitably aggregated into sessions, the sessions indeed reflect a Poisson process. These observations form the basis for our Session Extraction algorithm. Characteristics of Web traffic, such as the prevalence of pipelined and persistent HTTP connections, have been studied by analyzing HTTP connections that correspond to requests for a single Web page and its embedded objects [59]. Our work focuses on higher-level patterns of connections (possibly on different ports) exhibited by protocols, such as HTTP, and does not deal with specific application-level characteristics.

In addition, our work is similar to spirit to recent work that aims to automatically infer application-level packet structure. Machine-learning based techniques [38, 34] have been used to identify the application that a particular flow belongs to *without* relying on port numbers; our work is concerned with inferring the application's internal *session structure* and simply uses port numbers to identify applications. Protocol Informatics [54] uses sequence analysis techniques from bioinformatics to identify protocol fields in unknown or poorly documented applications. RolePlayer [16] uses byte-stream alignment algorithms, along with knowledge of a few low-level syntactic conventions, to infer protocol formats to enable RolePlayer to cheaply emulate the application-level behavior of a previously seen client or server. Our work is complementary to these efforts in that we focus on higher-level abstractions of sessions, while they are geared towards more fine-grained characterizations of traffic.

In abstract terms, our work is about discovering and characterizing *causality* in network traffic: which network activities are due to previously seen activities. This theme has been pursued by several lines of work in intrusion detection. First, host-based schemes have related network traffic received by a host with subsequent code executed by the host [14,13,42]. In addition, other work has tracked causality in network traffic in terms of an attacker moving among a set of nodes [62], identifying which hosts have infected other hosts [75,31], and

detecting "stepping stones" whereby attackers relay their traffic through previously compromised machines to obscure their identity while attacking other machines [63,82,78,67,9]. Our work differs from these in that, first, we aim to discover overall patterns of activity rather than detect individual instances; and, second, we do not have to contend with an adversary motivated to thwart our analysis. As a consequence, we can tolerate a greater degree of statistical uncertainty and decision errors. Furthermore, because our framework relies on little more than the properties of Poisson arrival process, we can establish a bounded false positive rate in terms of how often we report incorrect sessions.

2.4 Session Extraction

Our discovery process begins with *Session Extraction*: reducing a stream of connections to a stream of application-level sessions, where each session comprises a sequence of connections. We first detail different types of session structures (including homogeneous and mixed sessions), then describe how we extract homogeneous sessions, and finally how we extract mixed sessions.

2.4.1 Types of Sessions

The simplest possible session structure is a lone connection by itself, which we term a *singleton*. Next in complexity comes sessions consisting of consecutive invocations of the same application protocol, all with the same directionality, which we term *homogeneous sessions*. Last in complexity for the types of sessions we tackle in this work are sessions involving different connection types as well as varying directionality, *mixed sessions*. (More complex still are sessions involving multiple remote hosts. Extracting these remains for future work, though we have some preliminary results indicating its likely feasibility [28].)

Different applications vary widely in the prevalence they exhibit for each of these types of session structure. For example, the algorithm we develop classifies 11% of LDAP client sessions as singletons and 88% as homogeneous sessions (with 1% being mixed sessions);

for SSH client sessions, 80% singletons vs. 18% homogeneous sessions; and for Grid FTP client sessions, 58% vs. 0% (with 42% being mixed sessions). For about half of the 40+ applications we examined in our trace, simple forms of sessions dominate, while for the rest, sessions often involve somewhat more complex structure. In addition, for some protocols (Web surfing, peer-to-peer) many sessions involve multiple remote hosts, which our present structure abstraction does not aim to capture. Also, as our results later show, for applications that only rarely exhibit mixed sessions, sometimes these are sessions of particular potential interest to an analyst.

2.4.2 Extracting Homogeneous Sessions

Consider an algorithm that processes a stream of connection arrivals $C_1, \ldots$ in an online fashion. On observing a new connection C_i, the algorithm must decide whether: (a) C_i is part of a current session, or (b) C_i represents the beginning of a new session.

We first employ a simple heuristic to identify homogeneous sessions that, as explained before, consist of consecutive invocations of the same application protocol. For example, a user may invoke their mail client to send multiple mails in a single sitting, leading to consecutive SMTP instantiations. The same holds for Web browsing as reported in [44], which shows that one can capture HTTP sessions by simply considering HTTP connections less than a time T_{aggreg} apart as part of the same session. Their work found $\mathsf{T}_{aggreg} \approx$ 100 secs as suitable. We generalize this *aggregation rule* as follows. For a connection C_i, if we already have an existing active session $S_j = (C_1^j, \ldots, C_n^j)$ between the same pair of hosts and involving the same protocol and direction (*i.e.,* $A(S_j) \equiv T(C_1^j) = T(C_i)$) , and for which the most recently seen connection of S_j, C_n^j, arrived less than T_{aggreg} in the past from C_i's arrival, then we consider C_i part of S_j.

2.4.3 Extracting Mixed Sessions

The aggregation rule does not help for connections either involving different protocols, or somewhat further apart. The former are particularly interesting, as these potentially

reflect mixed sessions. In this case we attempt to assess possible *causality*, as follows. If C_i's arrival is indeed part of an ongoing session $S_k = (C_1^k, \ldots, C_m^k)$, then we can consider C_i as "triggered" (caused) by the connection C_1^k; C_1^k represents the start of the session S_k, and thus serves as a representative of the event that led to the initiation of the session.

We base our approach on the observation that if C_i is a "triggered connection", *i.e.,* causally related to S_k, then the arrival of C_i is likely to be "closer" to S_k, in comparison to the case where C_i is a "normal connection" (no causal connection to S_k). We now proceed to develop a more formal statement of this intuition by framing the problem in terms of hypothesis testing, and then explain a statistical detection algorithm that has a bounded false positive rate.

Modeling Normal Traffic. We can frame the problem of distinguishing between triggered and normal connections in terms of hypothesis testing [56]. In this formulation, we use the arrival time of a connection to choose between two hypotheses: the *null* hypothesis that the connection is normal, versus the *alternative* hypothesis that the connection is triggered.

Modeling the alternative hypothesis requires capturing the statistical dependence of the arrival time of the triggered connection on the arrival time of the triggering connection, but this may depend on the semantics of the specific application session itself. Fortunately, as we will show shortly, the null hypothesis is easier to model. Therefore our abstraction algorithm discovers likely causality, and hence sessions, by building a model for the arrival characteristics of untriggered (normal) connections. We then identify connections whose arrivals deviate from this model as triggered connections.

At the heart of our approach lies the empirical observation that the arrival of user-initiated sessions is generally well-modeled as a Poisson process, stationary over time scales of an hour [52, 44]. Here, *user-initiated* means sessions instigated by human activity rather than machine activity with a correlational structure (such as periodic daemons). Note that although the model of the arrival process can be relaxed from a Poisson process to a renewal

process, the Poisson assumption makes the false positive analysis simpler (as will be seen in Section 2.4.3).

To fit within this framework, we estimate rates for each type of session using a sliding window of duration $\mathsf{T}_{rate} = 1$ hr. Our model for session arrivals views the activity of local hosts as independent from that of other local hosts, and, further, independent of sessions of other types involving the same local host. We therefore maintain separate notions of session rates for the different types of applications in which each host participates. Finally, our definition of session type also includes directionality, accounting for the difference in the arrival characteristics of clients and servers.

Causality Detection Algorithm. We now describe a statistical test that identifies triggered connections by using our model of normal traffic.

Consider the arrival of two connections C_i, C_j with types denoted T_i, T_j. Assume that C_i is the connection at the start of an ongoing session S_1. Let us provisionally assume that C_j marks the beginning of a new, separate session, S_2. Denote the arrival rates of these session types as λ_1 and λ_2. Let the interarrival time between C_i and C_j be x. Now consider the alternative that C_i triggered C_j. In this case, in general we presume to find x significantly lower than the case when C_i and C_j are unrelated; we base this presumption on the expectation that connections due to a common origin will tend to come somewhat close together. Our strategy is therefore to estimate the probability P of observing an interarrival x for the null hypothesis of the connections being unrelated, and to deduce that C_i triggered C_j, and therefore C_j belongs to S_1, if P is less than a confidence threshold α.

Let T_1 and T_2 be two sessions whose arrivals follow independent Poisson processes with rates λ_1 and λ_2 respectively. Consider the event of an arrival of T_1 followed no later than x seconds by an arrival of T_2. We denote by $P[T_1, T_2, x]$, the expected number of such events per one unit of time. Given such a formulation, our causality detection algorithm proceeds as follows. We first categorize connections into different types; estimate rates for each of these types; and then use these computed rates along with the threshold α to detect

triggers. More specifically, on the arrival of a connection C (either incoming or outgoing) of type T involving a local host L, we perform the following actions:

- Let the sessions observed at L in the previous $\mathsf{T}_{trigger}$ seconds be $S_1, S_2, \ldots, S_n$, where $\mathsf{T}_{trigger}$ is a threshold specifying the maximum interval that can separate a triggered connection from the most recent activity in the session. In our study, we set $\mathsf{T}_{trigger}$ to be 500 sec.

- If any session S_i (a) has the same type as C, (b) involves the same remote host, and (c) had a connection arrival within a time window T_{aggreg} of C, then we add C to the most recent such S_i, and we are done. This is the simple aggregation heuristic discussed above.

- Estimate the rate of connection arrivals at L for each session type within the past T_{rate} seconds (3600 sec in our study). We form our estimate as simply the average interarrival time between sessions of each type, over the window size T_{rate}.

- For $1 \leq i \leq n$, compute $P[T_i, T, x_i]$, for x_i the interval between the arrival of S_i and C. Note that at this point C differs in type from S_i.

- If $P[T_i, T, x_i] < \alpha$ and C and S_i involve the same remote host, then add C to S_i. (We conceptually defer the test for the same remote host to this point because when expanding our work to discover sessions involving multiple remote hosts, it is at this point that we will modify the inference algorithm.) Also, note that if $P[T_i, T, x_i] < \alpha$ holds for multiple T_i, then C is added to *all* such sessions S_i.

- If the probability test does not identify C as belonging to any ongoing session, then C is considered to be the first connection of a new session S_{n+1}.

Naturally, the performance of the above algorithm depends critically on the parameter α. Too low a value may mean we miss certain causal links (false negatives); too high may lead to falsely aggregating unrelated connections (false positives). The false negative rate is difficult to characterize analytically due to the lack of a statistical model for the arrival

of triggered connections, so we rely on empirical analysis for evaluating it. Our choice for $P[T_1, T_2, x]$, however, allows us to provide a bound on false positives; see below.

False Positives. In this section, we establish an upper bound for $P[T_1, T_2, x]$, where T_1 and T_2 are two different types of sessions, and then use it to upper-bound the false positive rate (i.e., the number of false positives per second) in the presence of M different session types.

Theorem 1. *Let λ_1 and λ_2 be the arrival rate for sessions of type T_1 and T_2. Then, $P[T_1, T_2, x]$, the expected number of events where an arrival of type T_1 is followed by an arrival of type T_2 within time x, is $P[T_1, T_2, x] \leqslant \lambda_1 \lambda_2 x$.*

We present the proof for this below. Note that we also verified Theorem 1 using Monte Carlo simulations [28]. We prove Theorem 1 in two steps: First, we consider the case of a single Poisson process with rate λ, and compute the probability that it has two arrivals within x time units of each other in the time interval $[0, \tau]$. Then, we generalize this to the scenario where there are two independent Poisson processes with different rates.

Lemma 1. *Let $\{X_i, i \geqslant 1\}$ be the interarrival times of the Poisson process $\{N(t), t \geqslant 0\}$ with an arrival rate λ. Then, for some $\tau > 0$*

$$\Pr[X_i < x \quad \text{for some } i \leqslant N(\tau)] \leqslant \lambda^2 \tau x. \tag{2.1}$$

Proof. Conditional on $N(\tau) = n$, for any given $i \leqslant n$ we can write

$$\Pr[X_i > x | N(\tau) = n] = \left(\frac{\tau - x}{\tau}\right)^n = \left(1 - \frac{x}{\tau}\right)^n.$$

This holds since the conditional distribution for arrival epochs follows the order statistics for a joint uniform distribution of n random variables [56]. Therefore

$$\Pr[X_i < x | N(\tau) = n] = 1 - \left(1 - \frac{x}{\tau}\right)^n \tag{2.2}$$

$$\leqslant 1 - \left(1 - n\frac{x}{\tau}\right) = \frac{nx}{\tau}, \tag{2.3}$$

where (2.3) follows since the higher order terms in the Binomial expansion satisfy

$$\binom{n}{2}\left(\frac{x}{\tau}\right)^2 - \binom{n}{3}\left(\frac{x}{\tau}\right)^3 + \cdots \geqslant 0.$$

Since the time for the first arrival is identically distributed with the i^{th} interarrival time,

$$\Pr[X_i < x | N(\tau) = N] \leq N\frac{x}{\tau} \text{ , for } 1 \leq i \leq N-1$$

Next, let us look at the probability that there exists such an interarrival time within all n arrivals.

$$\begin{aligned}
&\Pr[X_i < x \text{ for some } i \leq N(\tau) | N(\tau) = n] \\
&= \Pr\left[\bigcup_{i \leqslant n} \{X_i < x | N(\tau) = n\}\right] \\
&\leqslant \sum_{i \leqslant n} \Pr\left[X_i < x | N(\tau) = n\right] \qquad (2.4)
\end{aligned}$$

$$= (n-1)n\frac{x}{\tau}, \qquad (2.5)$$

where (2.4) follows from the union bound.

Finally, using (2.5) we find a bound for the *unconditional* probability that there exists an interarrival of size smaller than x in the entire window of τ seconds, where for some $i \leq N(\tau)$,

$$\Pr[X_i < x] = \frac{x}{\tau}(\mathrm{E}[N(\tau)])^2 = \lambda^2 \tau x \qquad (2.6)$$

where (2.6) follows since the variance of a Poisson random variable is equal to its mean. Note that the bound we find becomes tight if the average interarrival time is large compared to x, *i.e.,* $\lambda x \ll 1$. In this regime, the probability that two or more occurrences of such

interarrivals is highly unlikely. Thus, the higher order terms in the binomial expansion are negligible and the union bound is tight.

We now generalize the previous lemma to the case when there are two types of sessions, type T_1 and type T_2, seen at a given local host.

Lemma 2. *Let* $\{N_1(t), t \geqslant 0\}$ *and* $\{N_2(t), t \geqslant 0\}$ *be two independent Poisson processes with rates* λ_1 *and* λ_2 *respectively. Let* $P[N_1(\tau), N_2(\tau), x]$ *be the probability that an arrival in* $N_1(t), t \leqslant \tau$ *is followed by an arrival of* $N_2(t), t \leqslant \tau$ *within* x *time units. Then,*

$$P[N_1(\tau), N_2(\tau), x] \leqslant \lambda_1 \lambda_2 \tau x. \tag{2.7}$$

Proof. Consider the combined process consisting of arrivals from both N_1 and N_2. Since this aggregate process is Poisson with rate $(\lambda_1 + \lambda_2)$, the probability that an interarrival of less than x occurs is upper bounded by $(\lambda_1 + \lambda_2)^2 \tau x$ by Lemma 1. Among such pairs we need to count those for which the first arrival belongs to N_1 and the second arrival belongs to N_2. From the independence of the two processes,

$$\begin{aligned} P[N_1(\tau), N_2(\tau), x] &\leqslant (\lambda_1 + \lambda_2)^2 \tau x \cdot \frac{\lambda_1}{\lambda_1 + \lambda_2} \cdot \frac{\lambda_2}{\lambda_1 + \lambda_2} \\ &= \lambda_1 \lambda_2 \tau x. \end{aligned} \tag{2.8}$$

Note that Theorem 1 is a direct consequence of Lemma 2. We simply plug $\tau = 1$ since we deal with the rate of false positives, i.e., number of false positives per time unit. This is the basis for the statistical test in our session extraction approach. Next, we consider the scenario where there are $m \geqslant 2$ types of connections.

Corollary 1. *Let there be* m *types of sessions and let* α *be the threshold for reporting a connection pair. Then the rate of false positives per unit time can be upper bounded as* $m^2\alpha$.

To prove this corollary, first note that the number of different pairs of session types under consideration is m^2 (note that a session of a particular type can also be a "trigger"

for a second session of the same type). Thus, from Theorem 1, since α is an upper bound on the rate of false positives for a particular ordered pair of session types, a simple union bound gives the formula in Corollary 1.

In practice, we choose the unit of time for measuring α as an hour and we found that $\alpha = 0.1$ per hour works well. Although our proofs of these theorems assume stationarity, our results apply even otherwise (e.g., non-homogeneous Poisson processes), as long as the rate estimation algorithm adapts to the changing rates. Finally, in our experiments, we found that the number of false positives is typically lower than the worst-case bound proved above.

2.4.4 Discussion

The fact that our Session Extraction approach is statistical imposes certain constraints on our abstraction approach. That it may exhibit false negatives is not particularly disconcerting: since our focus is on discovering application behavior, usually a trace will contain several instances of a particular type of behavior, so we can abide missing some. Our extraction need not find *all* instances to be successful. Further, our Session Extraction approach may also have false positives: for instance, the Poisson assumption used in our statistical test may not hold, or, it may so happen that two connections occur close-by in time simply by coincidence. We would not want our abstraction mechanism led astray into deducing session descriptors that incorrectly incorporate elements induced by false aggregations.

Our strategy is to choose α to obtain acceptable false negative performance, and to then design our abstraction mechanism to explicitly accommodate the level of false positives this leads to (for which the analysis in the previous section also gives a useful bound). We assessed different candidate values of α by manually assessing the false negatives for 4 applications (SMTP, FTP, two Web services) present in our traces, which led us to select $\alpha = 0.1$. This setting incurs a false negative rate of less than 25% [28].

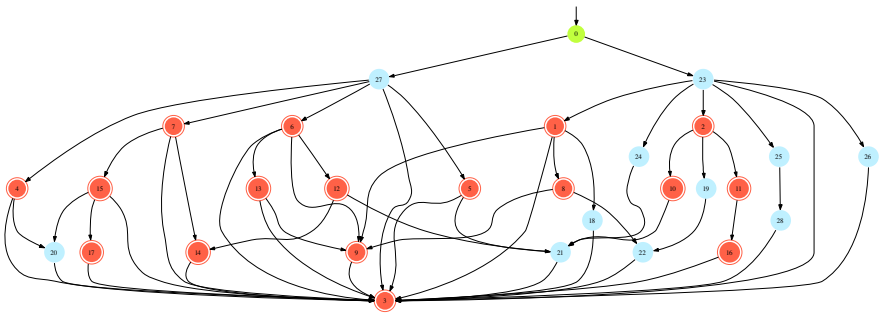


Figure 2.1. The Exact DFA for FTP

2.5 Structure Abstraction

The Structure Abstraction process aims to derive succinct descriptions for application sessions based on the set of session types reported by Session Extraction. We discuss how we represent session descriptions and then present our abstraction framework.

2.5.1 Representation of Session Descriptors

We first need to resolve "representation": what language should we use to abstractly describe the structure of sessions? We looked for a good balance between expressiveness and ease of generating abstractions from complex initial descriptions, which led us to choose regular expressions.[1] This choice was supported by our previous empirical experiences when we manually attempted to derive descriptions for different types of sessions [28]. In our discussions and figures we will often use the DFA equivalents of particular regular expressions. We also further refine this representation by labeling state transitions with probabilities, similar to the Customer Model Behavior Graphs used for characterizing workloads on websites [35].

Thus, given a set **ST** of observed session types $\mathbf{ST} = (ST_1, ST_2, \dots)$, we can capture

[1] We also experimented with using Hidden Markov Model inference techniques to abstract sessions, but found the results much harder to intuitively understand, as well as requiring computation that scales poorly with the size of the trace.

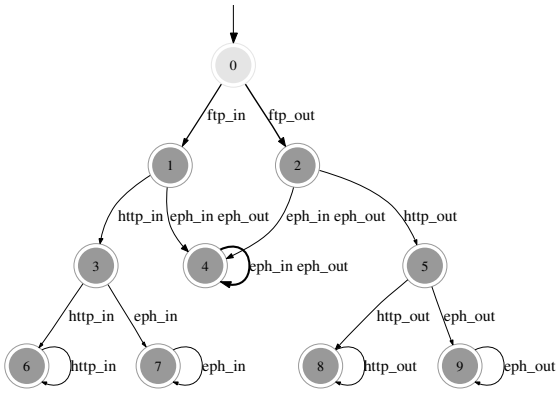


Figure 2.2. Abstract Session Descriptor for FTP

the full structural range using a regular expression that explicitly matches the entire set. Figure 2.1 shows such as complete DFA for FTP (as derived from our larger dataset). Here we have omitted the labeling because the point of the figure is simply to convey the great complexity that the full structural range can manifest. In this case, the DFA is complex (with 28 states) due to the fact that it has to *exactly* capture several FTP sessions varying in the number and the direction of data transfers.

Figure 2.2, on the other hand, shows a more "natural" (and tractable) DFA that abstracts much of the original while preserving some of its unintuitive features (the presence of HTTP transitions, for example). In the diagram, text labels are associated with the transitions to their left. A label like *http_in* indicates an inbound HTTP connection. *eph* corresponds to an ephemeral port (one that is both not well-known and changes from session to session). Also, recall that the thickness of an arc indicates its relative frequency of transition in comparison to all the other arcs in the DFA.

The goal of our structure abstraction framework is to derive such a "natural" DFA since it yields a number of benefits over the exact DFA:

Simplicity: The reduction in number of states and edges makes the DFA easier to comprehend.

Generalization: The abstracted DFA can capture a more complete set of possible structures, including some not present in **ST**. For example, in Figure 2.2 the construct

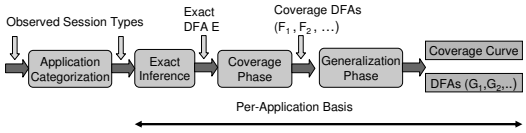


Figure 2.3. Overview of Structure Abstraction Framework

$(ftp\_in|ftp\_out)$ $(other\_in|other\_out)*$ captures an infinite set of session types; the "*" construct cannot occur in the exact DFA since it is derived from a finite trace.

Highlighting Common Behavior: Some types of sessions can exhibit quite different modes (*e.g.,* browsing sessions), and/or instances of individualized or idiosyncratic behavior (*e.g.,* login sessions that spawn many subsequent connections). By constructing abstractions that weight commonly seen elements over rare ones, we can highlight for the analyst tradeoffs between simplicity and capturing rare activity.

Minimizing False Positives: Given the statistical nature of our session extraction, the exact DFA may include false aggregations. Abstraction can help weed these out because they will tend to appear as isolated, rare structures, similar to the last item above.

2.5.2 Abstraction Framework

Figure 2.3 shows the four steps in our abstraction framework:

Application Categorization. This semi-automatic step identifies the applications in the trace and uses these to categorize the observed session types. The remaining steps operate on a per-application basis.

We lack ground truth for identifying the applications present in our trace at the granularity we desire, namely notions like "user is processing their email." Instead, we use the service associated with the first connection in a session as a proxy for the application type. To do so, we extracted a list of service ports by identifying those occurring in the trace more often than a fixed threshold, $\mathsf{T}_{service}$ (which we set to 5). We then manually analyzed this

list to determine the associated application via either entries in an extensive directory [53], or, in ambiguous cases, by inspecting packet payloads when available. Armed with the list of types of applications, we then categorize sessions based on the server port of the first connection in that session.

Exact Abstraction. This step produces an exact DFA, E, that describes the session structure of an application A based on the complete set of session types, **ST** observed for A. We construct E from the union of each of the observed session types, minimizing the DFA using the FSA toolbox [22].

Coverage Phase. This step (detailed in Section 2.5.3) emits a sequence of DFAs $F_1, F_2, \ldots$ that represent subsets of E that progressively (and greedily) account for greater and greater *coverage* (fraction of the set **ST** matched) as we add edges.

Generalization Phase. This step (detailed in Section 2.5.4) applies three generalization rules to the sequence $F_1, F_2, \ldots$ that introduce commonly useful abstractions, producing a set of generalized DFAs, $G_1, G_2, \ldots$.

At the end of this process, we present to the analyst a *coverage curve* that plots the coverage of generalized DFAs against their complexity. The analyst then uses this curve as guidance regarding which DFAs to inspect in order to understand the application's session structure at different levels of abstraction. We discuss this process in Section 2.5.5.

2.5.3 Coverage Phase

Our coverage phase aims to extract a set of DFAs that capture subsets of the observed session behavior that best trade off simplicity-of-expression (fewest states/edges) for coverage (capturing most types of observed behavior). As noted above, this helps both with keeping rare or peculiar session instances from obscuring the commonly observed patterns, and for minimizing the effects of false positives in our extraction algorithm (since these will tend to manifest as rare, peculiar sessions). This phase produces a sequence of DFAs $F_1, F_2, \ldots, F_n$ that "cover" the observed set **ST** to increasing extents. We formulate the notion of a DFA F_i's "coverage metric" with respect to a set **ST** as the fraction of session

types in **ST** accepted by F_i, weighted by the frequency with which the type occurs. Given this definition, the following greedy algorithm produces a sequence of DFAs with increasing coverage metric.

We first feed every session instance in **ST** to E, the exact DFA, accumulating a *hit count* $h(e)$ for every edge $e \in E$, *i.e.,* the number of traversals of e. Next, for each edge e we compute the *augmented hit count*, $h'(e)$, as $\sum_{\text{e' reachable from e}} h(e')$. The purpose of this augmented hit count is to capture the implicit dependencies among the edges in the DFA. The overall idea is to prioritize "upstream" edges more than "downstream" edges. We then order edges by decreasing $h'(e)$; denote this ordering by $e_1, e_2, \ldots$. Finally, we construct DFAs F_i by taking the union of all edges $e_1, \ldots, e_i$.

2.5.4 Generalization Phase

In this step we subject the coverage DFAs, $F_1, F_2, \ldots$, to a set of transformations to obtain a sequence of generalized DFAs, $G_1, G_2, \ldots$. We found 3 generalizations that worked well across about 40 applications in our trace:

Prefix Rule. If we observe a session type $ST_i = (T_1^i, \ldots, T_m^i)$ in **ST**, then consider any prefix of this type also a session type for the application. We implement this rule by marking all states of the DFA F_i as accepting states.

Invert Direction Rule. We base this rule on the observation that an application session is typically independent of the direction (inbound vs. outbound) of the originating connection. For example, if we observe $ST_i = (ftp\_in, other\_in)$, then we extend the DFA to also match $\overline{ST_i} = (ftp\_out, other\_out)$.

Counting Rule. If a DFA F_i matches aBc and aB^nc, (for $n > 1$), where a and c are individual connection types and B is a union of one-or-more connection types (*e.g.,* $other\_in|other\_out$), then we transform F_i so that the corresponding G_i matches aB^+c. Note that in Section 2.6 we find that restricting $n = 2$ provides satisfactory results.

Finally, while the order in which we apply these rules affects the generalized DFA G_i,

it can be shown that the following sequence is idempotent: Prefix Rule, Invert Direction, Counting Rule. Thus, we simply apply the rules in this sequence once and output the result as G_i. We also note that these rules have an appealing monotonicity property: if the sequence of DFAs $F_1, F_2, \ldots$ has increasing coverage metrics, the sequence $G_1, G_2, \ldots$ retains this property.

In our experiments, we chose to apply all of these rules before presenting the DFA to the analyst. (An alternative would be to leave the choice of which rules to apply to the analyst.) The Prefix Rule intuitively holds since any session may terminate mid-way due to various error conditions. In our trace, we found only one case where the Prefix Rule not hold: one mail server always originated exactly two reverse Ident connections in response to an incoming SMTP connection. The Invert Direction rule is sometimes incorrect in the sense that a particular session structure visible in sessions originated from inside (outside) may not apply for sessions originated in the other direction. We however chose to apply it since the weights included on the edges do reflect this fact to the analyst. Also, the session structure for most applications in our trace also conform to the Counting Rule (the mail server we alluded to before does not conform to this rule).

2.5.5 Coverage Curves

A coverage curve, such as that shown in Figure 2.4(A), plots the number of edges i in the generalized DFA G_i against G_i's coverage metric. Note that though we obtain F_i (the basis for G_i) by retaining exactly i edges E, G_i may have more or fewer than i edges. This is because our generalization rules can simplify the structure of the DFA by adding edges (Invert Direction and Counting Rules) or marking certain states as final states (Prefix Rule). Since our DFAs are always minimized at every step in the process, performing the minimization procedure, after such a simplification of structure, may *decrease* the number of edges. This phenomenon is what leads to the non-monotone nature of the curve, as illustrated in the dip at 8 edges in Figure 2.4(A).

A knee in the coverage curve marks a point where the coverage metric increases sharply

with the addition of a few particular edges. Such knees generally correspond to *modes*: points where adding a bit more complexity to the abstraction provides a substantially more comprehensive description. Such knees guide the analyst in choosing which DFAs merit scrutiny. In addition, the coverage curve helps the analyst deal with the problems of idiosyncratic sessions and false aggregations. Both of these typically appear towards the far right side of the coverage curve.

2.6 Results

In this section, we describe the evaluation of our scheme using 4 weeks of traces collected at the border of the Lawrence Berkeley National Laboratory, a site with about 8,000 hosts that on average participated in 2,700,000 connections each day. We used the first 2 weeks for calibration and guiding the design of heuristics in our scheme, while we present results obtained by using our calibrated scheme over the last 2 weeks.

2.6.1 Parameter Settings

Session Extraction. For setting the timing parameters T_{aggreg}, $\mathsf{T}_{trigger}$, and T_{rate}, we were guided by [52, 44]. We verified that for the corresponding values, for most applications the arrival process of sessions was Poisson-like (see [28] for detailed results). A few applications, such as *ntp*, violated Poisson grossly because they are timer-based, but in general, the imprecision of our statistical session extraction test is remedied by the way in which our structure abstraction mechanism trims rarely exhibited behavior.

We set T_{aggreg} (used in the aggregation rule) to 100 sec. We also experimented with a few other values (200 and 500 sec) suggested in [52, 44], but in the range we experimented with, we found that our final session descriptors did not vary much.

We set $\mathsf{T}_{trigger}$ (the maximum duration between the finish time of a session and the arrival time of a new connection) to 500 sec. In general, for most application sessions, $\mathsf{T}_{trigger}$ need be no greater than 100 sec. However, a few applications like FTP and Login

sessions, sometimes had long sessions in our trace. So, we conservatively set $\mathsf{T}_{trigger}$ to be 500 sec based on duration of sessions we observed in our trace. Note that the number of false positives in our mechanism is upper-bounded by the parameter α (irrespective of the value of $T_{trigger}$). Thus, despite conservatively setting $T_{trigger}$ to 500 sec, the false positive performance is still under the threshold α.

We set T_{rate} (the time duration over which rate estimates are computed) to 3,600 sec, the value over which the arrival rates were reported to be stationary in [52, 44].

We set the threshold used in our statistical test, α, to 0.1, based on our calibration of sessions for four applications. We first extracted, using our partial *a priori* knowledge of the session structure for SMTP, FTP, and two Web services (a Web proxy service, and an HTTP interface to a service running on 9303/tcp), all sessions belonging to these applications in the trace. We then examined how many of the *session types* implied by these legitimate sessions were found by our session extraction test. Except for FTP, it found all of them. For FTP, extraction generally missed long sessions consisting of several data-transfer connections, but such sessions have a very simple structure of the form (**ftp**) (**ftp-data**)*. Missing these is not a serious concern when inferring session descriptors, since our structure abstraction using its generalization rules can "fill in the gaps".

Structure Abstraction. The parameters used in structure abstraction are relatively easier to set. Regarding the semi-automatic application categorization step, since our packet captures did not include the entire packet, we had to rely on port numbers to classify connections into applications. Once this classification was done, we looked at all applications occurring at least 5 times in the 2-week trace (*i.e.,* $\mathsf{T}_{service} \geq 5$).

Apart from this parameter, there are two details worth noting. In our naïve implementation, the counting rule (inferring general positive closure) is too expensive to implement for high values of $|B|$ (the size of the repeating unit). Since the number of states s sometimes exceeds 100, we examined the impact of using much smaller values. We found that simply restricting the rule to the case of $|B| = 2$ allows us to correctly infer positive closure in each instance where we know *a priori* that it makes sense. In addition, we only feed

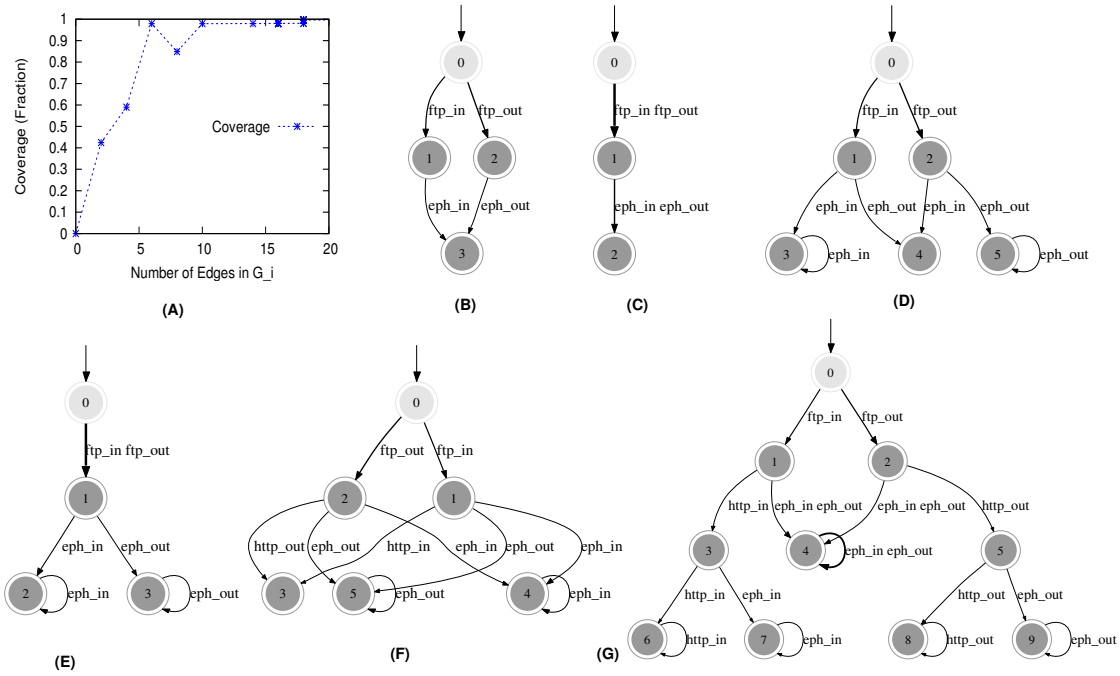


Figure 2.4. **FTP** Coverage Curve (A), DFAs: 4 edges (B), 4 edges (C), 6 edges (D), 8 edges (E), 10 edges (F), 18 edges (G)

our structure abstraction algorithm session types of length ≤ 10—otherwise, the number of states in the DFA can explode due to the exponential growth in the number of session types with increasing length. One example of such a long session is an FTP session where a control connection is followed by, say, 15 data connections. We find that omitting such sessions does not result in any loss of information in the inferred session descriptors, since our generalization rules can usually capture such session structures anyway (in this case, by allowing any number of data connections).

Role of Analyst. In obtaining the results below, the role of the analyst is to first use the coverage curve in order to select a cut-off coverage fraction, and then peruse the appropriate DFAs corresponding to that fraction. In some cases, the existence of knees in the coverage curve makes this choice easy to make, while in others (such as HTTP), the presence of a long tail implies that the analyst can choose as much detail to "explore" as much of the tail as he desires.

2.6.2 Empirically Observed Session Structure

We now turn to examining some of the session structures discovered for the applications in our trace. For session structures that can be verified by using protocol specifications, we found that our session descriptions generally capture *all* the behavior implied by the specification with no false positives. Unfortunately, many session structures arise due to empirical behavior, and for those, we can only assess their plausibility. Finally, some sessions arise for truly anomalous reasons, such as misconfigurations or attacks. These last will typically be few in number, and will not appear in the DFA unless the analyst asks for very high coverage. However, they are interesting in demonstrating the value of having a general tool that can detect causality.

FTP

Figure 2.4 shows the coverage curve (subplot A) for FTP , and DFAs representing some knees in this graph in subplots B-G. We use this as our primary example of the possibilities of our session structure discovery.

The coverage curve shows the number of edges in the generalized DFA G_i versus the coverage provided by G_i. We see a linear increase until 6 edges, after which the coverage tapers off slowly until about 100 edges (tail not shown in the figure). We now examine the various knees in this curve, showing that they usually correspond to some feature of the underlying sessions. Note that we describe the generalized DFAs corresponding to the knees in this graph as ordered by the number of edges included from the exact DFA; this does not necessarily correspond to ordering them by the number of edges in the DFA itself, since the generalization procedure can sometimes simplify the DFA considerably.

The first noteworthy point occurs at 2 edges. This is simply the DFA (not shown) that captures singleton incoming and outgoing FTP sessions. The second point occurs at 4 edges (subplot B), corresponding to the DFA capturing sessions with a single data transfer connection in the same direction as the initial control connection. Subplot C shows the next DFA of interest, which also has 4 edges (the plot shows the highest coverage exhibited by a

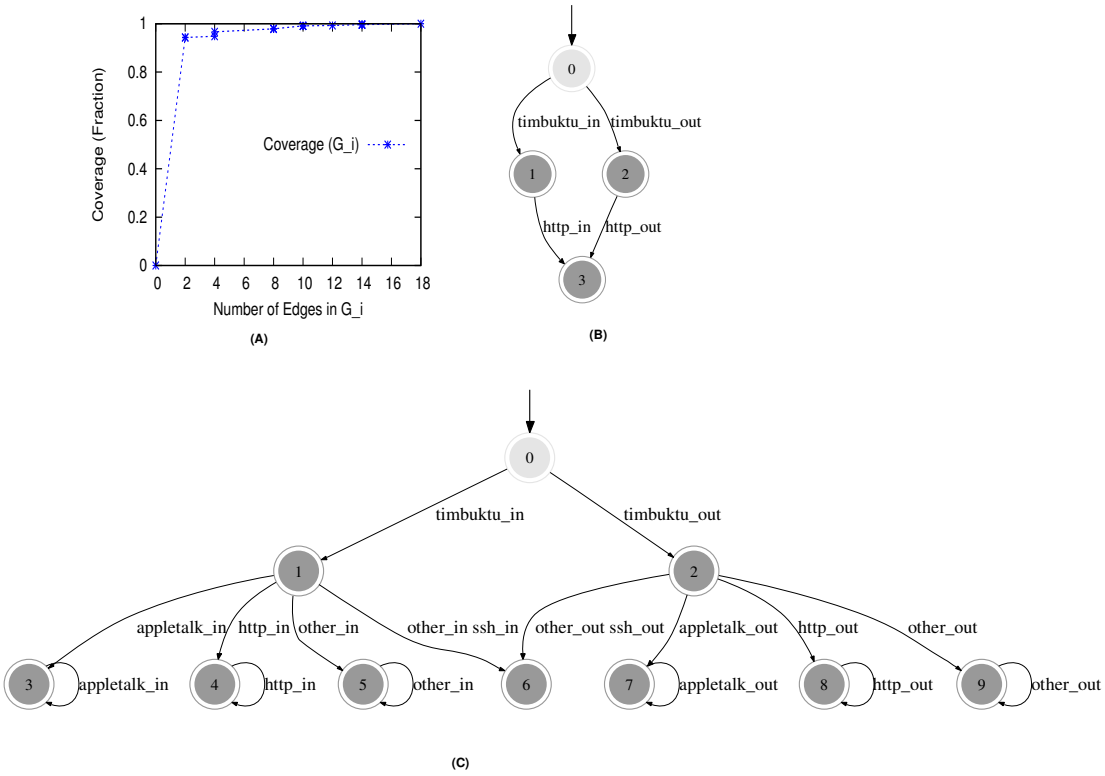


Figure 2.5. **Timbuktu** Coverage Curve (A) and DFAs: 4 edges (B), 18 edges (C)

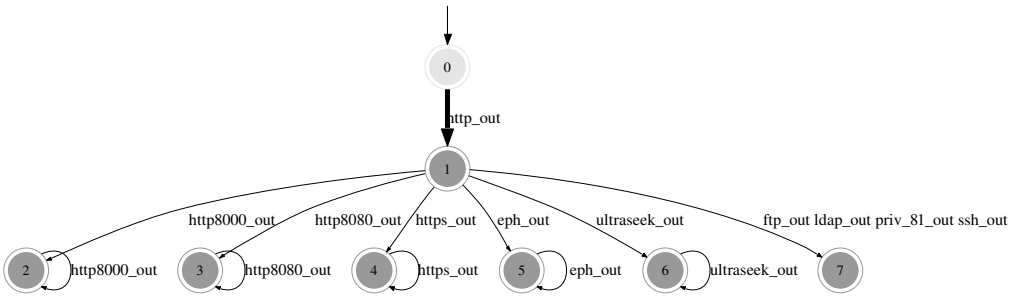


Figure 2.6. **HTTP** DFA with 30 edges

DFA of a given number of edges; so the coverage in the plot for 4 edges corresponds to this DFA). This captures the pattern $(ftp\_in|ftp\_out)(eph\_in|eph\_out)$, which allows for sessions with a single data transfer in either direction. Subplot D shows the next interesting DFA, with 8 edges, which also captures incoming (outgoing) FTP sessions with a single data transfer in the opposite direction. The DFA in Subplot E includes more edges from the exact DFA, but has fewer actual edges due to generalization of the structure of the DFA, capturing sessions with any number of data transfers in the same direction. The DFA in Subplot F (10 edges) shows how HTTP connections can occur during FTP sessions, likely due to intermingled access to Web pages with links to FTP URLs. Finally, another knee (not particularly visible in the coverage curve) occurs at 18 edges, as shown in subplot G. This DFA captures sessions with any number of FTP or HTTP transfers in either direction. This DFA has over 99% coverage, implying that it "explains" nearly all the sessions found by session extraction, and, further, captures all the characteristics of the FTP protocol specification.

Timbuktu

Figure 2.5 shows the coverage curve and two pertinent DFAs for Timbuktu [64], a Mac and Windows remote desktop application. The coverage curve (subplot A) shows a sharp knee in the beginning, gradually tapering off towards the tail. The knee corresponds to singletons, which comprise $> 90\%$ of sessions. Although the tail accounts for $< 10\%$ of the remaining sessions, it reveals interesting details. Subplot B shows the DFA with 4 edges, revealing that Timbuktu sessions may include some browsing behavior as well. Subplot C shows the DFA with 10 edges. The associated ephemeral ports likely correspond to dynamic ports negotiated in the main channel (which corresponds to the Timbuktu listening port). We also see browsing behavior reflected by associated HTTP connections, and that Timbuktu connections can occur in conjunction with the AppleTalk file sharing protocol, presumably due to users performing file transfers along with remote control software. Finally, SSH connections also occur in such sessions, suggesting that login connections of several kinds tend to occur together.

HTTP

HTTP sessions come in a number of variations. By far, the most common ($\approx 99\%$) are singleton or aggregated sessions that reflect successive retrieval of multiple pages from the same server, which the coverage curve shows as a very sharp knee very early on. However, there is also a long tail clearly visible in the coverage curve, accounting for the other 1% of sessions. To illustrate the useful information that may be gleaned from this tail, Figure 2.6 presents the HTTP DFA with 30 edges. For clarity, we show only half the edges, those corresponding to sessions begun with an outgoing HTTP connection. We chose this DFA simply to illustrate the variety of HTTP sessions; it does not correspond to any obviously visible knee in the coverage curve. Although these sorts of DFAs in general reflect remote tail behavior, for some particular hosts they can be quite prominent (*e.g.,* a server or a crawler). Highlighting such host-specific behavior for an analyst is a promising area for our future work.

The figure highlights that HTTP (port 80) can occur in conjunction with multiple connections on other ports (81, 8000, 8080, HTTPS, FTP) typically used for Web access. Likely the presence of ephemeral ports in the DFA also reflects this sort of linking, for example due to services offering HTTP interfaces that include client-side code (such as JavaScript) for retrieving additional data from the service. Connections to a port 8765 (marked "ultraseek") likely reflect an Ultraseek search engine crawler (supported by our observation that the hosts exhibiting this traffic have names that suggest indexers). We also see LDAP connections, perhaps reflecting Web services that rely on it for authentication. Finally, we see outgoing SSH connections. These may simply reflect a user's "start up" routine of opening both some Web pages and logging into a commonly visited remote host.

Deeper in the coverage curve, the DFAs also reflect AOL and other connections occurring in conjunction with HTTP, perhaps reflecting chatting and browsing alongside, and a greater variety of services offered via Web pages (*e.g.,* Oracle database access and music streaming applications that use Real Time Streaming Protocol and Windows Media).

Other Applications. We now summarize some of the more interesting additional

Figure 2.7. Session Illustrating Anomalous Behavior

application session structures that our scheme exposed in the trace, which included mail-related applications (SMTP, IMAP4, POP3), remote access (SSH, Exec, Rlogin, Telnet), database (ORACLE, MYSQL), bulk transfer (Grid FTP), Windows (NetMeeting, NETBIOS), and peer-to-peer (BitTorrent, KCEasy).

Mail-related session structures due to client-side applications (*e.g.,* ThunderBird or Outlook) exhibit session structures of the form "*smtp_in (imap4_in | imap4ssl_in | pop3_in | pop3ssl_in)** " , reflecting the different protocol clients use to send and receive email. We also see server-side sessions of the form "*smtp_in (smtp_out | ident_out)*," reflecting SMTP mail relays and Ident reverse connections.

Remote access applications typically exhibit session structures such as "*ssh_in (ident_out | X11_out*)*," due to reverse Ident connections initiated by servers, and X11 connections initated by the users back to their desktop X11 servers. We also see structures such as "*ssh_in (ssh_out | vnc_in)*," which exhibit a "stepping-stone"-like structure [63, 82], though without the goal of "laundering" traffic since the users connect back to their own originating site. Note that, in the regular expression above, VNC stands for Virtual Network Computing, a popular remote desktop protocol.

Presumably such activity reflects creating multiple login windows or transferring files to supplement the login session. We also find structures such as "*ssh_in (ftp_in | http_in)*," presumably due to user browsing behavior once logged. Finally, GridFTP is dominated by sessions with a single outbound connection followed by multiple inbound ephemeral connections (though sometimes this all occurs in the opposite direction); services such as Oracle, SQL, and NetMeeting exhibit session structures that include multiple ephemeral connections alongside the primary connection on a well-known port; and P2P applications display the session structure of the form "*app_out app_in**."

2.6.3 Finding Attacks Using Anomaly Detection

Our experimental analysis over these traces also revealed sessions exhibiting anomalous structure reflecting malicious activity. Indeed, our original goal had been to detect network attacks by finding sessions that deviate from established session structures. Our hypothesis was that such deviations would reflect either unintended misconfigurations (a host behaving as spam relay or as a Web proxy), scanning, or "phone home" connections associated with compromises. Figure 2.7 shows an example of such an attack (confirmed by the site). The event consists of an incoming *ssh* connection, which compromises the host, followed by the host visiting a Web server (presumably controlled by the attacker), an incoming port 65535 connection (likely the attacker instructing the bot software they installed on the host) and then an outbound IRC connection (presumably to a botnet).

In assessing the sessions uncovered by our extraction algorithm, we have also found anamolous sessions caused by peer-to-peer applications disallowed by the site's policy (and using non-standard ports), and an instance where machine within the site acted as a HTTP relay for Yahoo Web pages, for reasons we could not determine.

Thus, although our session extractor, being statistical, might not catch all anomalous sessions, the sessions it does catch can be of considerable operational interest. Unfortunately, we find that such activity often only manifests in the upper tail of the coverage curve. Thus, there are usually too many strange-but-benign sessions—plus structures (perhaps) falsely inferred due to the statistical limits of our detection algorithm—for upper-tail instances to serve as directly "actionable" indications of attacker activity. We do hold hope, however, that detection of peculiar sessions can serve as input into further analysis that might use additional complementary information to derive actionable conclusions. Such information might be, for example, history about the host (*e.g.,* what services it typically runs, and which remote hosts it has contacted in the past).

2.7 Conclusion

In this work we have demonstrated a statistical technique to extract application sessions from a connection-level trace of network activity, and shown how to deduce descriptors that can be used by an analyst to capture the qualitative structure of such sessions.

This chapter has only dealt with the issue of inter-connection context in the network communication between two hosts. However, for some security purposes, there is a vital need to collate context from across multiple sites; the next chapter deals with the issue of gaining global context when multiple sites cooperate with one another in order to counter fast worm attacks.

Chapter 3

Cooperative Worm Containment

In this chapter, we discuss how *global* context is useful for security monitoring. In cases where the threat spreads rapidly across various networks, there is significant advantage to be gained by exchanging information between various networks in order to identify the threat. We study this issue in the context of containing fast worms. An emerging approach towards this problem is a *decentralized* solution in which *firewalls* in various access networks *cooperate* amongst themselves by exchanging information regarding ongoing worm attacks. Thus, the main idea is to exchange *context* among multiple sites in order to detect such fast worms.

The principal advantage of such decentralized cooperative architectures is that they may be easier to deploy compared to end-host based schemes (TaintCheck [42], Vigilante [13]) that deal with a wider class of worms but incur a heavy deployment cost, schemes that may require support from routers in the core of the Internet (Wong et al [73], Shield [66], Berk et al [8]), or schemes that place the burden of analysis on a centralized infrastructure (Wormholes [70], Wu et al [74]). A decentralized design that modifies only firewalls offers, apart from deployability, superior fault-tolerance and scalability as compared to a centralized design. A centralized approach places most of the reliability and trust on a few entities, and even if implemented as a third-party overlay network, the infrastructure invested in the

overlay network has to scale along with the number of participants. Such an infrastructure may not even be feasible if a significant fraction of the Internet traffic has to be handled in the overlay network. A decentralized approach, on the other hand, distributes trust among its participants, and moreover, has the self-scaling property that the analysis load grows only as fast as the infrastructure available for analysis.

Several recent works (Hard Perimeters [68], Weaver et al [71], Anagnostakis et al [6], Nojiri et al [43]) suggest that such decentralized cooperation may be effective in containing fast worms. However, we still lack a clear understanding of the fundamental efficacy and limitations of decentralized cooperative worm containment. Towards addressing this goal, we take a generalized approach towards quantifying the effectiveness of decentralized cooperation as a paradigm for worm containment. Our methodology for studying cooperation is to first propose an abstract model of cooperation that captures, to a limited extent, the design space of different cooperative schemes. We then use a combination of analysis and simulation to study the level of containment against scanning worms in this model. In particular, we seek to answer three questions:

- *Cost of Decentralization:* Since the participating firewalls are at different points in the Internet, they can only exchange information at a finite rate, unlike the centralized case. What are the different forms of information exchange and how does its rate affect the containment?
- *Cost of Partial Deployment:* Given that all Internet firewalls may not be part of the cooperative, how does containment depend on the level of deployment in the Internet?
- *Cost of Malice:* Decentralized cooperation is also vulnerable to malicious firewalls in the cooperative that may subvert its operation. How does one deal with such malice and what is the tradeoff between containment and the level of tolerable malice?

In our abstract model of decentralized firewall-level cooperation, each *deployed* firewall employs *local detection* to detect infection within its network. Once it detects an infection, it notifies other firewalls of the ongoing worm infection in two ways: *implicit* and *explicit*

signaling. Firewalls that receive these signals are alerted about the worm attack and can then *filter* their incoming traffic based on these signals. This model integrates various known techniques for local detection (Throttling [65], Threshold Random Walk [27]) and filtering (Autograph [29], Vigilante [13], TaintCheck [42]) into a framework for cooperation.

Our results based on this model are as follows. First, decentralization does not impact containment negatively if the reproduction rate λ of a worm (defined in Section 3.3) is less than 1. In this regime, almost complete containment can be achieved without any signaling overhead. If $\lambda > 1$, signaling is required for good containment (over 95%), and the rate of signaling can be tuned to achieve a desired level of containment. We show that even with a moderate signaling rate, good containment can be achieved. Second, we study the dependence of cooperation on the level of deployment, and to improve the performance under partial deployment, we propose a scheme called *rerouting*. Our results show that even at a deployment level of 25%, cooperative containment protects over 95% of the vulnerable networks. Third, our analysis shows that one can deal with a few hundred malicious participants in an Internet-level cooperative. We use a simple thresholding mechanism to thwart malice in the system, which leverages the fact that a worm infection is wide-spread and affects several networks.

3.1 Problem Setting and Methodology

We now describe our simplified model of the Internet and the class of worms we wish to defend against. We consider the Internet to consist of access networks that are connected to their ISPs through an access link. Each network consists of several end-hosts and is monitored by a firewall on its access link. We assume that a firewall can observe all traffic to/from an end-host in its network. In this work, we mainly focus on cooperative containment of fast random scanning worms. Such worms perform a local topological scan followed by a global uniform random scan: an infected host first infects all hosts in its network and then probes external IP addresses uniformly at random to find more vulnerable hosts. Though

our analysis can also be generalized to more sophisticated worms, we primarily focus on such fast random scanning worms which includes most of the worms seen so far.

3.1.1 Evaluation Methodology

Our primary metric of success against a worm attack is the fraction of vulnerable networks that escape infection. We refer to this as the *containment* metric. We count uninfected networks instead of hosts, since eventually, either all vulnerable hosts belonging to the same network are infected or none of them are infected. We use two methods to evaluate the containment metric. In our *analytical* method, we make certain simplifying approximations in our problem setting, the most important ones being the assumption of a uniform distribution of vulnerable hosts across vulnerable networks and the assumption that network delays can be ignored. These assumptions make our analysis more tractable at the cost of ignoring real-life details. For this reason, we also present results based on a discrete-event *simulation* that quantifies the effect of non-uniform host distribution and network delays on our results. We present simulation results in two environments: one where all the assumptions used in obtaining the analytical results hold, and the other under more realistic settings (the first purely serves to validate our analysis).

3.2 Modeling Cooperation

In this section, we will define an abstract model that, to a limited extent, captures the common features present in any decentralized cooperative firewall-level containment scheme. Our model is meant to be simple and tractable to analysis, and serves to chart out the design space of decentralized cooperation.

We identify three fundamental functions in any cooperative scheme: detection, signaling, and filtering. *Detection* is the mechanism by which a firewall analyzes its own traffic to deduce that a worm attack is in progress, *signaling* is the technique by which it passes information about this worm attack to other firewalls, which then use this information

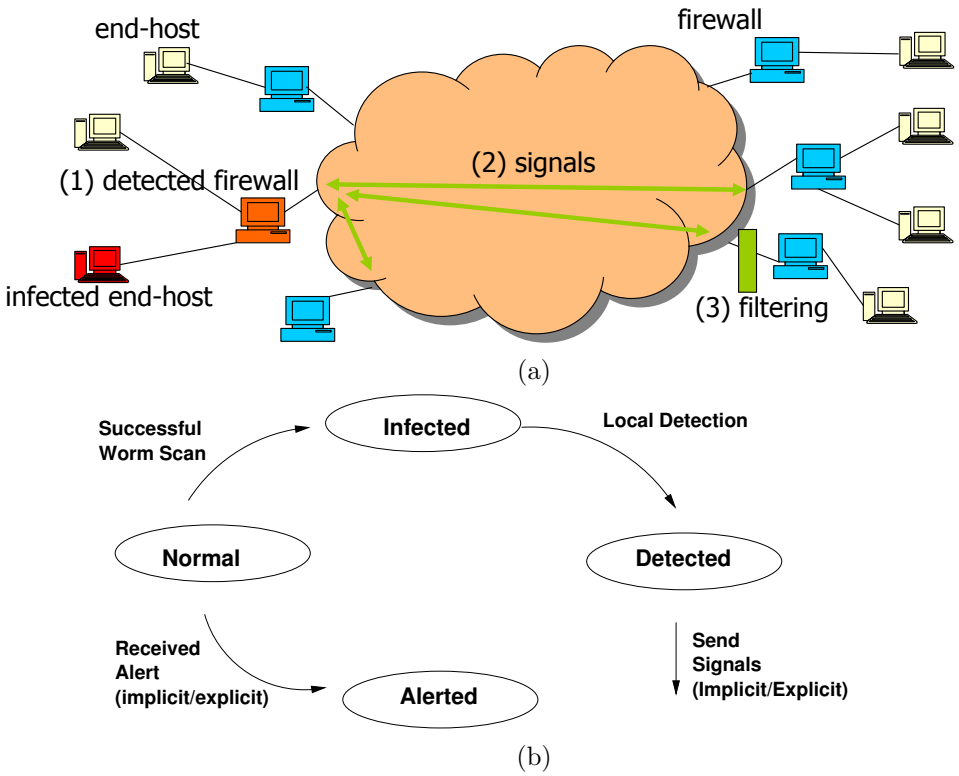


Figure 3.1. (a) Decentralized Cooperation (b) State Transition Diagram

to *filter* their own incoming traffic. The various cooperative schemes known in literature use a particular mechanism to implement each of these functions. This architecture is illustrated in Figure 3.1(a) and the corresponding state diagram of a firewall is illustrated in Figure 3.1(b). Every firewall is in one of the following four states: normal, alerted, infected, and detected. A firewall transitions into the infected state when a single host in its network is infected, and then to the detected state upon local detection. A firewall is alerted upon receiving signals from a detected firewall. Note even if a single host is infected in a network, effectively the entire network has been compromised.

We describe the operation of the protocol under two assumptions: (a) Complete Deployment: All Internet firewalls participate in our cooperative. (b) Trustworthy Participants: All firewalls operate according to our protocol. Later, in Sections 3.4 and 3.5, we will consider mechanisms to handle partial deployment and malice.

Local Detection: A firewall uses local detection to determine whether its own network is infected by analyzing *outgoing* traffic. Analyzing outgoing traffic cannot aid a firewall in

protecting its *own* network, since the characteristics of outgoing traffic change only *after* its network is infected. It can however aid in alerting other firewalls of an ongoing infection; this is the main idea of cooperation. There are two advantages in analyzing outgoing traffic as opposed to analyzing incoming traffic. A firewall can maintain characteristics of its outgoing traffic using per-host state, unlike incoming traffic which can be very noisy due to routine Internet crud. Also, a decision made using incoming traffic can potentially be influenced by external malicious hosts sending traffic to the firewall's network.

Local detection can be achieved by using existing techniques (Throttling [65], Threshold Random Walk [27]) that identify infection by analyzing various characteristics of outgoing traffic, *e.g.,* number of unique destination addresses, connection failure rate etc.

Signaling: A firewall that has detected infection within its network using local detection proceeds to notify other firewalls of its infection. These notifications include *filters* for identifying malicious packets. We assume that firewalls, upon detecting worm infection within their network, can identify such a filter (we will discuss various possible filtering mechanisms in the next paragraph). When a detected[1] firewall notifies other firewalls, these firewalls are said to be "alerted" to the attack. There are two forms of such notifications: *implicit* and *explicit*.

Implicit Signaling: A detected firewall sends implicit signals by *marking* all suspicious outgoing packets. Suspicious packets are identified using filters inferred by the filtering mechanism. Marking can be done by using bits in packet headers or by encapsulating suspicious packets within special headers. This marking serves two purposes. First, it informs the destination firewall that the packet is possibly malicious. The destination firewall can simply drop such packets or it can process them in a special manner (*e.g.,* send it to a hardened end-host for analysis). Thus, the decision to drop/process is delegated to the destination. One can also imagine these markings being processed by intermediate routers; we do not deal with that possibility here. Second, the destination firewall is also

[1]We use the term "detected firewall" to denote a firewall whose local detection scheme identifies infection within its network.

notified that the source network is infected. This enables the destination firewall to install filters before its own network gets infected.

Explicit Signaling: A detected firewall sends explicit notifications of its infection to other participating firewalls at some fixed rate E. We only consider schemes where signals are sent to randomly chosen participating firewalls. This naive method can easily be improved by, say, a publish-subscribe based system; however the security properties of such systems are harder to characterize.

In both types of signaling, note that a firewall can report only of its *own* infection; it cannot implicate other firewalls. We use a challenge-response protocol to verify that the originator of a signal is the infected firewall itself. We assume that among the list of firewalls in the cooperative is known to everyone beforehand; this list also includes a public key used for authenticating the signals. This rules out *framing* attacks in which malicious firewalls make false accusations. Such a protocol also eliminates spoofing attacks by malicious end-hosts.

Filtering: We refer to a firewall that has been alerted by a signal as an *alerted* firewall. Such a firewall installs filters and drops malicious *incoming* traffic matching these filters. These filters are gathered from the signals it receives. Filtering is also used by a detected firewall during implicit signaling: only outgoing packets identified as malicious by the filter are marked. A very simple filter could be based on port numbers: any traffic on alerted ports is dropped. More sophisticated filters can be inferred using known techniques, such as Autograph [29] which deduces signatures from packet payloads, and Vigilante [13], TaintCheck [42], which infer signatures by host-based mechanisms.

This protocol can also be extended to the case when multiple worms may be propagating. The filtering mechanism would infer filters for each propagating worm, and firewalls maintain state and perform signaling on a per-worm basis.

3.2.1 Discussion

Although not all cooperative schemes fit into our abstract model, we believe that it captures the essence of cooperation in its simplest form. We will now discuss some salient features of our model of cooperation. First, the fact that signals can be *verified* using a challenge-response mechanism makes the security properties of our model much easier to characterize. Second, our model incorporates both forms of signaling, implicit and explicit, whereas existing work has dealt mainly with explicit signaling. There are pros and cons with both approaches. Implicit signaling has lesser overhead and may be simpler to implement, but requires that marked packets be allowed through the wide-area network. Explicit signaling, as we will show, is however necessary in order to contain worms with high scanning rates. Third, our model integrates the various techniques proposed in the literature of worm defense in a framework for cooperation. Decoupling the various mechanisms used in cooperative containment also aids analysis. Finally, in this model, each firewall analyzes only its outgoing traffic; we will later show the utility of analyzing incoming traffic in the partial deployment scenario.

3.3 Cost Of Decentralization

In this section, we analyze the containment metric C (the fraction of vulnerable networks that escape infection) of our cooperative model under two cases: with and without signaling.

3.3.1 Parameters

We first introduce the assumptions and the parameters used in our analysis. In our worm model, an infected host first infects all vulnerable hosts in its own network in *zero* time, and then proceeds to probe external addresses at a fixed scan rate. The scanning rate of a fast worm is limited by access bandwidth, and hence we use the scanning rate s of a single *infected network* in our analysis. The probability of a successful probe p is the number of vulnerable hosts divided by the size of the IP address space. We denote the

N	Total number of Vulnerable Firewalls	$100,000$
n	Number of Deployed Firewalls	$100,000$
m	Number of Undeployed Firewalls	0
p	Probability of Successful Probe	0.0005
s	Scanning Rate of Worm	20000 per sec
t_d	Detection Time	0.2 secs
λ	Reproduction Rate	$phst_d = 2$
σ	Defined for Notational Convenience	$\frac{\lambda-1}{t_d}$

Table 3.1. Default Setting of Parameters in our Model

total number of vulnerable networks as N. We use the homogeneous cluster model, used for modeling Slammer victims in [69], to assume that the vulnerable hosts are uniformly distributed among all vulnerable networks. For purposes of analysis, our model of local detection is as follows: we assume that the time taken for an infected firewall to detect its infection is an exponential variable with mean t_d. This assumption is certainly not true for all local detection schemes, but considerably simplifies the analysis. Thus, our model includes several simplifications, including the uniform distribution of vulnerable hosts. We show later in Section 3.6 that the trends in our results hold even if these assumptions are not exactly true.

Our default settings for these variables (shown in Table 3.1) are as follows. We used $p = 0.0005$ corresponding to a vulnerable population of about 2 million since a widespread worm typically compromises hosts in the order of millions. We set the scanning rate s as 20000 (the average scan rate of Slammer [69], one of the fastest worms so far). Thus, these settings correspond to one of the most virulent worms seen yet. The number of firewalls N was set to 100000 based on the number of observed BGP prefixes (obtained from routeviews.org). Thus, every network has an address space of 2^{15} and about 20 vulnerable hosts. Under these settings, the worm takes 0.6 seconds to infect 99% of all vulnerable hosts, assuming no network congestion. We set $t_d = 0.2s$ in our analysis (*e.g.,* using TRW [27]) and use implicit signaling. By default, we assume all networks are deployed.

3.3.2 Analyzing Detection and Filtering

First, we analyze a simplified variant of our scheme without any signaling under complete deployment. Thus, there is no sharing of information between firewalls. In this variant, a firewall performs only local detection and filtering (once a firewall enters the detected stage, outgoing scans are marked and cannot infect any more hosts). Thus, the defense against a worm attack is that an infected host can effectively probe only until its firewall does not detect infection. The following lemma shows that such a simplified scheme is surprisingly effective under some conditions. We define $\lambda = spt_d$ where λ corresponds to the expected value of the number of successful infections by an infected network before its firewall detects infection. We refer to λ as the reproduction rate. We now state the following lemma (also proved by Staniford [61]):

Lemma 3. *If $(\lambda < 1)$, then as $N \to \infty$, the containment metric $C \to 1$. Further, for any finite N, $C \geq 1 - \frac{I_0}{(1-\lambda)N}$, where I_0 is the number of infectees at time zero.*

Proof. We prove this lemma using a differential equation model, which naturally extends to our analysis of signaling.

$$\frac{d}{dt}(n_i) = (sp)(n_i - n_d) \tag{3.1}$$

$$\frac{d}{dt}(n_d) = \frac{(n_i - n_d)}{t_d} \tag{3.2}$$

The first equation follows since the rate of increase of infected firewalls is the same as the rate of successful scans (expressed as the number of infected undetected networks times the scanning rate per network). This equation overestimates the value of $n_i(t)$, since it assumes that no scans are sent to already infected firewalls. This overestimate suffices for this analysis since we only seek to obtain an lower bound on the containment. The second equation shows that the number of detected firewalls follows the number of infected firewalls with a phase lag of t_d (since it takes time t_d for detection). Defining $f(t) = n_i(t) - n_d(t)$, where $f(t)$ is the number of scanning firewalls, gives the following equation (by subtracting

Equation (3.2) from Equation (3.1)):

$$\frac{d}{dt}(f) = (sp - 1/t_d)f$$

If $\lambda < 1$, then $(sp - 1/t_d) < 0$, and thus this corresponds to an exponential decay with a decay rate of $(1 - \lambda)/t_d$. This implies that as $n \to \infty$, $C \to 1$. One can also identify this process as a birth-death process [56] in time steps of t_d to obtain $E[I_\infty] = \frac{I_0}{1-\mu(B)} = \frac{I_0}{1-\lambda}$ (proof omitted).

□

Earlier worms like Blaster had low scan rates leading to low values of λ, and could have been controlled by simple detection and filtering even if the firewalls were deployed at the level of class B networks. Note that $\lambda = 2$ under our default settings. Surprisingly, even if $\lambda > 1$, detection and filtering still provide some containment against a *random scanning* worm, although the effectiveness degrades rapidly with λ. The reason this occurs is that as the infection proceeds, it takes longer for a random scanning worm to find uninfected hosts, and thus the local detection schemes have a greater chance of throttling the infection.

Lemma 4. *If* $(\lambda > 1)$*, assuming* $I_0 \ll N$*,* $C \geq 1 - \min_{k:k>1}(\frac{(k\lambda-1)(k+1)}{k\lambda(k-1)} - \frac{2*\log(k\lambda)}{(k-1)\lambda})$ *against a random scanning worm (k is a variational parameter used in minimization).*

Proof. The exact form of Equations (3.1,3.2) is:

$$\frac{d}{dt}(n_i) = (sp)(n_i - n_d)(n - n_i)/n \tag{3.3}$$

$$\frac{d}{dt}(n_d) = \frac{(n_i - n_d)}{t_d} \tag{3.4}$$

Observe that λ_{eff}, the effective birth rate, decreases with time as $\lambda_{\text{eff}} = \lambda(n - n_i)/n$. This reflects the fact that is harder to find vulnerable hosts as the infection proceeds. In our analysis, we use the differential equations above to model signaling until λ_{eff} decreases down

to $1/k$ from its initial value λ. We choose $k > 1$ so that $\lambda_{\text{eff}} < 1$. At this point, we use the previous lemma to bound the total number of infectees. We first modify the differential equations as:

$$\frac{d}{dn_d}(n_i) = \lambda(n - n_i)/n \tag{3.5}$$

$$\Rightarrow \log(\frac{n}{n - n_i}) = \lambda\frac{n_d}{n} \tag{3.6}$$

The first equation is obtained by dividing Equations (3.3,3.4), and the second is obtained by integration. Consider the point where $\lambda_{\text{eff}} = 1/k$ which implies that $\frac{n-n_i}{n} = \frac{1}{k\lambda}$. At this stage, $f_i = (1 - 1/k\lambda)$ fraction of networks are infected, and a fraction $f_d = \log(k\lambda)/\lambda$ of networks are in the detected stage (obtained by substituting for n_i in Equation (3.6)). The containment C is then given by:

$$C \geq 1 - (f_i + \frac{(f_i - f_d)}{k-1}) \tag{3.7}$$

The term f_i counts the number of infected firewalls when $\lambda_{\text{eff}} = 1/k$, while the second t erm counts the number of firewalls that will be infected from this point onwards (by using the previous lemma). This bound holds for all k, therefore one can minimize over all k, which proves the lemma. □

Our analytical results based on the above lemma indicate that detection and filtering provide about 40% containment at $\lambda = 1.5$ and about 20% at $\lambda = 2.0$.

3.3.3 Analyzing Signaling

First consider the case when implicit signaling is used along with detection and filtering. Denote the total number of deployed firewalls as n. We model the infection process as follows. At time t, denote by $n_i(t)$ the number of infected firewalls, by $n_d(t)$ the number of detected firewalls, and by $n_a(t)$ the number of alerted firewalls (the remaining firewalls

are in the normal state). Note that $n_i(t)$ represents all infected firewalls, so $n_i(t) - n_d(t)$ is the number of infected firewalls that have not yet detected infection. A simple differential equation model for implicit signaling is as follows:

$$\frac{d}{dt}(n_i) = \frac{s(n_i - n_d)(n - n_i - n_a)}{N} = s_n(n_i - n_d)(n - n_i - n_a) \tag{3.8}$$

$$\frac{d}{dt}(n_d) = \frac{n_i - n_d}{t_d} \tag{3.9}$$

$$\frac{d}{dt}(n_a) = \frac{s(n_d)(n - n_i - n_a)}{N} = \frac{s_n n_d(n - n_i - n_a)}{p} \tag{3.10}$$

Here, s_n is the normalized scan rate s/N defined for convenience. The first equation counts the number of successful outgoing scans from infected firewalls that have not yet detected infection, while the second relies on the exponential distribution of the time to detection. The third equation calculates the rate of spread of alerts as the number of alerts per detected firewall multiplied by the probability of the alert being sent to an unalerted (and uninfected) firewall. We could not solve these equation exactly, so we present two additional results: a closed-form upper bound on the containment metric and numerical integration of these equations. Our upper-bound is given as:

Lemma 5. *For $\lambda > 1$, the containment metric C by using implicit signaling is at least $1 - \frac{(log(N)+c)t_d\sigma^2}{hs}(\frac{1}{t_d\sigma} + 1)$ with probability of at least $1 - e^{-e^{-c}}$, where $\sigma = (\lambda - 1)/t_d$.*

Proof. We derive a lower bound on the containment metric by dividing the worm infection into two phases: the first phase extends until the time all firewalls have been alerted, at which point the second phase begins. In the second phase, the worm cannot spread any further since all firewalls have been alerted. We approximate the first phase as follows: we assume that no firewall has been alerted, and thus the worm propagates just as in the case of detection and filtering alone. Clearly, this suffices to derive an upper bound on the number of infected machines at the end of the first phase, and consequently a lower bound

on the containment metric. We first analyze the progress of the worm in the first phase (there are no alerted firewalls, only detection and filtering). This is the same as Equations (3.1,3.2):

$$\begin{aligned} \frac{d}{dt}(n_i) &= \frac{(hsp)(n_i - n_d)(n - n_i)}{N} \\ &\approx (hsp)(n_i - n_d) \qquad (3.11) \\ \frac{d}{dt}(n_d) &= \frac{(n_i - n_d)}{t_d} \qquad (3.12) \end{aligned}$$

The first equation approximates $(n - n_i)/N$ to 1 since an upper bound of n_i suffices for our purposes (we assume that the number of infected firewalls remains low throughout the first phase or equivalently that the worm does not probe infected networks). This implies that:

$$\begin{aligned} \frac{d}{dt}(n_i - n_d) &= (phs - \frac{1}{t_d})(n_i - n_d) = \sigma(n_i - n_d) \\ n_i(t) - n_d(t) &= I_0 e^{\sigma t} \qquad (3.13) \end{aligned}$$

The first equation uses $\sigma = (\lambda - 1)/t_d$, where λ is the birth rate. We assume $\lambda > 1$ since otherwise there is no necessity for implicit signaling. The second equation follows by integrating the first, and imposing the constraints that $n_d(0) = 0$ and $n_i(0) = I_0$, the number of infected firewalls at $t = 0$. This can be substituted in Equations (3.11), (3.12) to derive n_i, n_d as a function of time:

$$\begin{aligned} \frac{d}{dt}(n_d) = \frac{I_0 e^{\sigma t}}{t_d} \Rightarrow n_d &= \frac{I_0(e^{\sigma t} - 1)}{t_d \sigma} \qquad (3.14) \\ n_i &= I_0(\frac{e^{\sigma t}}{t_d \sigma} + 1 - \frac{1}{t_d \sigma}) \qquad (3.15) \end{aligned}$$

The first equation is obtained by integration, and the second by substitution in Equation (3.12). These two equations track the growth of the number of infected and detected firewalls in the first phase. Based on this, we calculate the *total* number of implicit signals

$i(t)$ sent by time t as:

$$\begin{aligned} i(t) &= \int_0^t (hs) n_d \, dt = \int_0^t \frac{I_0 hs}{t_d \sigma} (e^{\sigma t} - 1) \, dt \\ &= \frac{I_0 hs}{t_d \sigma} (\frac{e^{\sigma t}}{\sigma} - t) \end{aligned} \tag{3.16}$$

The allocation of these $i(t)$ implicit signals sent by time t to the various firewalls is analogous to the allocation of coupons in the classical coupon collector's problem [56]. Note that the first phase ends when all firewalls have received at least one alert each, analogous to the condition that all coupons must be collected. Notice here that since the worm scans external networks uniformly at random, implicit signals follow the uniform distribution as well. We use the standard result that if m is the number of coupon collectors and the number of coupons exceeds $m \; log(m) + c \; m$, then the probability that all m collectors have one coupon each is at least $1 - e^{-e^{-c}}$ [56]. In our case, since each firewall must receive an alert to get alerted, we use $m = N$. Thus, the time t_1 at which the first phase ends with probability $\geq 1 - e^{-e^{-c}}$ can be calculated using $i(t_1) = N \; \log(N) + c \; N$ (we approximate $\left(\frac{e^{\sigma t}}{\sigma} - t\right)$ to $\frac{e^{\sigma t}}{\sigma}$). This time t_1 can be substituted in Equation (3.15) to obtain $n_i(t_1)$, the number of infected firewalls at the end of the first phase as:

$$n_i(t_1) = \frac{N t_d \sigma^2 (log(N) + c))}{hs} (\frac{1}{t_d \sigma} + 1) \tag{3.17}$$

and thus $C = 1 - \frac{n_i(t_1)}{N}$ with probability $\geq 1 - e^{-e^{-c}}$. □

By substituting for σ and setting $\lambda > 1$, the above lemma suggests that the containment metric decreases linearly with t_d and s and drops quadratically with p. In the case of explicit signaling at rate E along with implicit signaling, the containment metric can be obtained by simply substituting $s = s + E$ in the denominator of Lemma 5 to obtain $C \geq 1 - \frac{log(N) t_d \sigma^2}{s+E} (\frac{1}{t_d \sigma} + 1)$. Notice that the containment metric varies with the parameter E as $1/(1 + E/s)$.

In conclusion, these three lemmas illustrate the following. If the reproduction rate $\lambda < 1$, then no signaling is required for good containment. If $\lambda > 1$, then without signaling, only moderate/poor containment can be achieved. Implicit signaling is required for good containment if $\lambda > 1$, and explicit signaling can help further improve the containment. Further, the rate of explicit signaling E, can be tuned to achieve a desired level of containment.

3.3.4 Numerical Solutions

We now present numerical results exploring two aspects of cooperation: the containment that it offers against worms of varying virulence and how this containment depends on the various mechanisms used in cooperation. Our results are obtained by two means: (a) numerical integration of the differential equation model (b) discrete-time event simulation to validate the analytical method (marked "Sim" in our plots).

All parameters are as mentioned in Table 3.1. The initial number of infected hosts is set to 10, and the integration/simulation is run until all firewalls are either infected/alerted. The simulation maintains a state variable per firewall and updates this variable every time-step depending on the probes and alerts sent by other firewalls. The time-step used in the simulation is set to the reciprocal of the scan rate, so that every infected host sends exactly one scan per time-step. All simulation results are averaged over 25 runs.

Varying Worm Virulence

We present results corresponding to four different axes of worm virulence in order to evaluate the performance of cooperation against known worms and worms of the future.

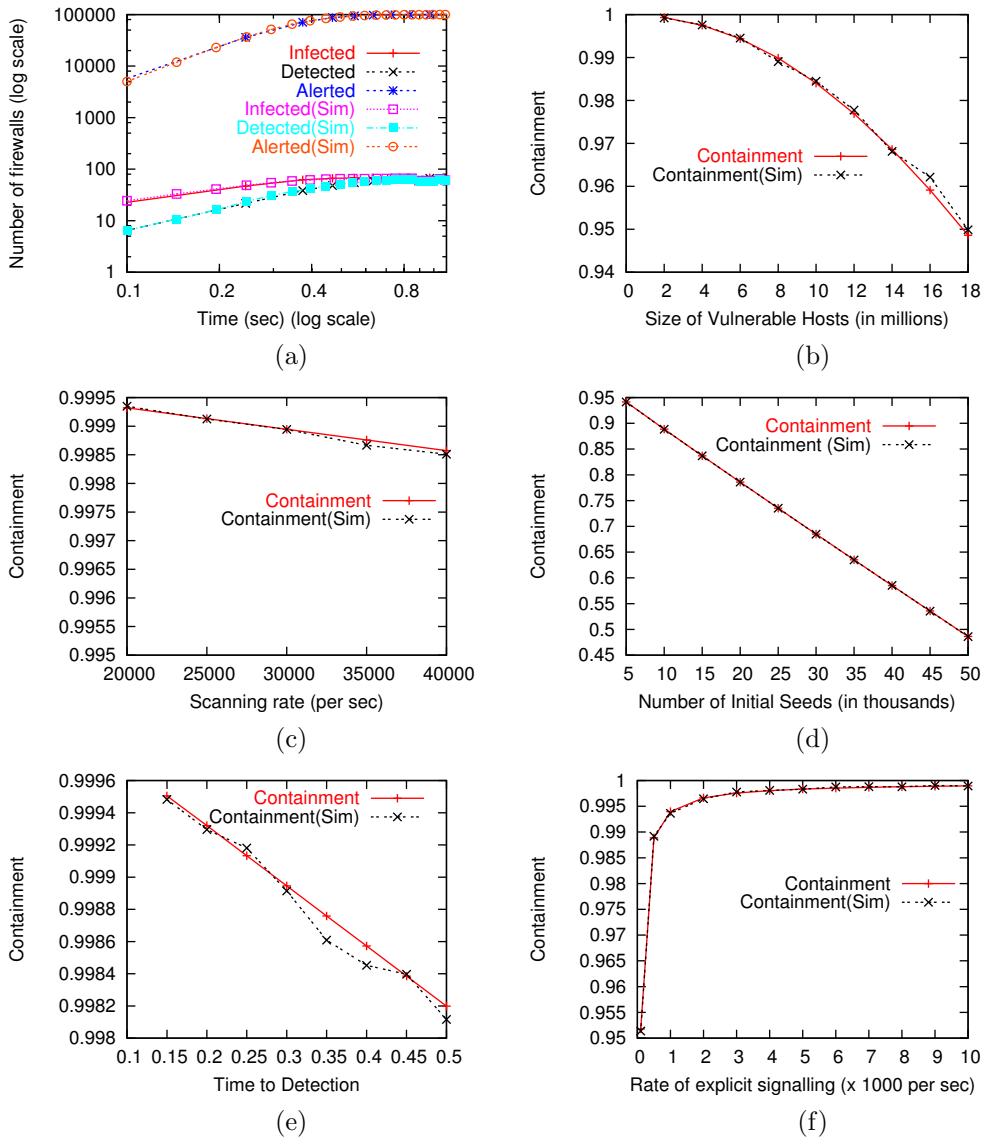


Figure 3.2. (a) Temporal Dynamics: $\lambda = 2.0$ (with implicit signaling) (b) Containment Vs. Size of vulnerable population (c) Containment Vs. Scanning Rate (d) Containment Vs. Number of Initial Seeds (e) Containment Vs. Time to detection (f) Containment Vs. Rate of Explicit Signaling

Temporal Dynamics: We first give results in Figure 3.2(a) of the temporal dynamics of cooperative containment using implicit signaling. Observe that the number of detected firewalls always lags behind the number of infected firewalls as expected. Very early in the propagation, the number of alerted firewalls surpasses the number of infected firewalls, and from then on, grows like the classic sigmoid curve. This is because we are using implicit signaling, in which case the spread of alerts is similar to that of the worm itself.

Number of Vulnerable Hosts: The size of the vulnerable population is a key parameter of worm virulence, and we plotted the containment metric against this parameter in Figure 3.2(b). The number of vulnerable networks was kept constant, and the size of the vulnerable population was varied between 2 and 20 million hosts (corresponding to $p = 0.0005 - 0.005$). As suggested by the analysis, there is roughly a quadratic drop in C with p.

Scanning Rate: The scanning rate of a worm determines its propagation time, and we plotted the containment metric against this parameter in Figure 3.2(c). The containment metric exhibits a slow linear drop with scanning rate of the worm. Since implicit signals are piggybacked on worm scans, faster the worm scanning rate, greater the effective signaling rate as well.

Number of Initial Seeds: Our analytical bounds assume that the set of hosts used to seed the infection is a small subset of the vulnerable population. This assumption may not be true if botnets (networks consisting of already infected zombie hosts) are used to seed a worm attack. We studied this scenario in Figure 3.2(d). The containment metric drops to about 50% if botnets consisting of 1 host each in $50,000$ different networks is used to seed the attack. If on the other hand, the hosts in the botnet can be clustered. Since

every network has 20 vulnerable hosts, a highly clustered botnet would have a total of 5000 infected networks, in which case the containment level is 85%.

Varying Cooperation Parameters

We present results detailing how containment depends on local detection and the explicit signaling rate E.

Local Detection: Figure 3.2(e) illustrates the sensitivity of cooperation to local detection. As suggested by the analysis in the previous section, the dependence is roughly linear. Though the worm propagates in about $0.6s$, even with time to detection t_d as high as $0.5s$, cooperation performs very well. This might appear surprising, but note that $0.5s$ seconds is the *mean* time to detection, and with a finite probability, detection occurs before $0.5s$.

Rate of Explicit Signaling: The rate of explicit signaling corresponds to the overhead of cooperation, and we plotted containment metric for varying rates E in Figure 3.2(f). In obtaining this plot, only explicit signaling is used. The containment metric improves with rate E although the variation is minimal.

3.4 Cost Of Partial Deployment

In this section, we consider the partial deployment scenario when some firewalls in the Internet do not participate in the cooperative. We first propose a simple technique that works in the partial deployment scenario, and then describe a technique called *rerouting* to improve the containment metric.

3.4.1 Baseline Solution

Under partial deployment, the main limitation of our firewall-level model is that it is not possible to prevent undeployed networks from getting infected eventually, since infected hosts in undeployed networks can always infect other undeployed networks. This limitation *cannot* be overcome without support from core routers. We therefore choose our metric for the partial deployment case as the containment among the *deployed* firewalls. As a baseline solution, our existing scheme for signaling works in this case as well. Only deployed firewalls perform detection and signaling: undeployed firewalls do not engage in local detection, signaling, or filtering. The differential equation model in Section 3.3.3 can be extended to study the partial deployment scenario. Unfortunately, we could not obtain any closed-form expressions in this scenario, and hence present only numerical results based on our differential equation model.

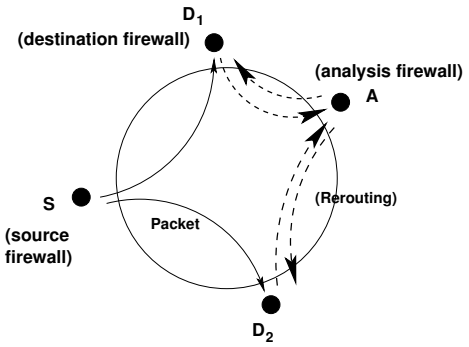
3.4.2 Rerouting

In order to define the limitations of cooperation in the partial deployment scenario, we describe a technique called *rerouting* that emulates the idealized model when all firewalls are deployed. The main idea is that deployed firewalls perform detection and signaling on behalf of the undeployed firewalls. The load of monitoring traffic from undeployed firewalls is shared by all deployed firewalls by using a mapping (say, based on a hash function) from a *source* firewall to an *analysis* firewall. In our rerouting scheme (illustrated in Figure 3.3(a)), all deployed firewalls redirect their incoming traffic from a undeployed firewall X to a analysis firewall $A(X)$. $A(X)$ is a deployed firewall chosen using a hash function, and the traffic seen by it serves as a crude sampling of the traffic sent by X. $A(X)$ monitors

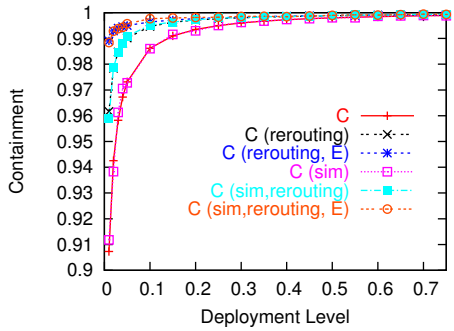
this traffic to perform local detection and signaling on behalf of X. Rerouting improves the containment metric because scans from undeployed firewalls will also be detected and used to initiate signaling. Note however that traffic sent by X to other *undeployed* firewalls will not be monitored: rerouting only approximates the idealized scenario when all firewalls are deployed.

Rerouting suffers from a few limitations that may limit its usefulness in practice. First, observe that rerouting assumes address spoofing is not possible: otherwise a malicious firewall might send scans spoofing the address of an undeployed firewall, wrongly implicating the latter. Thus, rerouting works only for TCP worms or bi-directional UDP worms. Second, there are important practical concerns in rerouting: the overhead and privacy concerns due to traffic redirection. Both of these depend on the local detection scheme in use. Some local detection schemes (*e.g.,* TRW [27], Throttling [65]) require only the redirection of connection setup packets, while in the worst case, all packets may need to be forwarded. In the case of TRW [27], the only information revealed is the source firewall per connection and the success/failure status of the connection. Other optimizations to improve the overhead and privacy may also be possible, such as, sampling connections or sending summaries of several connections.

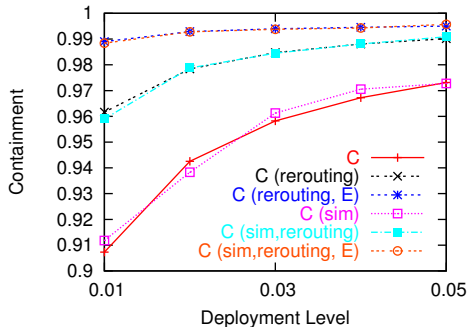
When rerouting is enabled, we argue that the undeployed firewalls effectively behave as deployed firewalls with $t'_d = t_d/\alpha$ where $\alpha = n/N$ (the fraction of deployed firewalls). The argument is that the traffic seen by the analysis firewall $A(X)$ is a fraction α of the traffic sent by X. Under the assumption that detection takes time proportional to the amount of malicious traffic, $A(X)$ detects the infection of X by the fraction $1/\alpha$ slower than X itself would have done if it were deployed. This assumption holds for both TRW [27] and Throttling [65]. One can now modify the previous analysis to model rerouting.



(a)



(b)



(c)

Figure 3.3. (a) Rerouting Mechanism (b) Containment in the Partial Deployment Scenario (b) Containment at very low deployment levels

3.4.3 Numerical Results

We present results that demonstrate the containment in the partial deployment scenario in Figure 3.3(b). We show three results: (a) implicit signaling without rerouting (b) implicit signaling with rerouting (c) explicit signaling ($E = 1000$) with rerouting. The simulation for the partial deployment scenario is as follows. Out of $(M + N)$ firewalls, M randomly chosen firewalls are marked undeployed, and the rest are marked as deployed. For simulation rerouting, at the start of the simulation, we generate a random mapping from each of the M undeployed firewalls to one of the N deployed firewalls. The first observation is that even without rerouting, cooperation offers about 97% containment at deployment levels of 5%. With the inclusion of rerouting, the containment improves to 99%, and with a low rate of explicit signaling to about 99.5%. Rerouting becomes very useful at low levels of deployment (as shown in Figure 3.3(c)), and at deployment levels beyond 50% or so, offers minimal advantages. As seen before, explicit signaling adds little to the containment provided by implicit signaling.

3.5 Cost Of Malice

We now analyze the security of our model against malicious firewalls. In a cooperative framework, there are certain fundamental limitations on the level of robustness against malicious participants.

First, in a cooperative framework, it is important to distinguish between a global Internet-wide attack and an infection localized to a small set of networks. Thus, if a small number of firewalls report that they have been infected, even if they are being truthful, this attack could simply be a localized one. Thus, a fundamental necessity in the cooperative

framework is that a firewall should enter the alerted stage and install filters only if it receives signals from T distinct firewalls, where T is a chosen threshold.

Second, it is also necessary to defend against firewalls attempting to trigger false positives/negatives. A false positive occurs when a firewall sends an incorrect notification that it is infected. It is fundamentally impossible to detect whether such a notification is indeed a false alarm, since a malicious firewall can originate traffic as if its own network had been infected. We handle such false positives using the thresholding mechanism. Since the alerted state is entered only upon receiving T signals, our scheme can resist up to T malicious firewalls that trigger a false alarm. This clearly exposes the tradeoff between containment and robustness to malice. On one hand, filtering should be enabled as soon as possible, but on the other hand, it should not be possible for a few malicious participants to incorrectly invoke filtering. If filtering is invoked incorrectly, packets may be wrongly identified as malicious and dropped. Even if signals are verifiable (as in Vigilante [13]), the cost of using filters to check every incoming packet may be too high. A false negative occurs when a firewall deliberately/otherwise fails to report of its infection. It is possible to design schemes to detect such firewalls, for example, by extending the rerouting mechanism so that traffic from a deployed firewall is audited by a small set of analysis firewalls. We avoid the overhead of such schemes by opting for a simpler approach: we treat such firewalls as an undeployed firewall. Our scheme does not depend on every firewall being deployed, and as we have already shown, our scheme obtains good containment even under about 25% deployment.

Third, in our scheme, one deployed firewall cannot implicate another deployed firewall: our challenge-response mechanism prevents such attack. A deployed firewall can however take advantage of the rerouting mechanism to report infection of all the undeployed fire-

walls that it is responsible for. We handle this case by simply extending the thresholding mechanism so that alerts from T distinct *deployed* firewalls are necessary to trigger filtering.

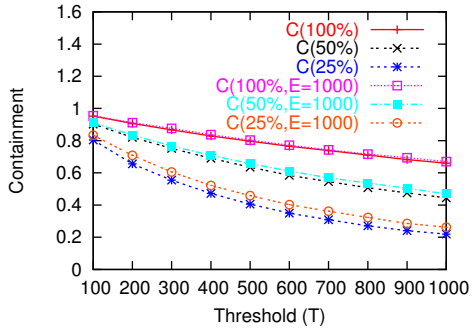
3.5.1 Analyzing Cooperation under Malice

We only have closed-form analytical results for the complete deployment scenario, and rely on simulations in the partial deployment case. Under thresholding, a firewall enters the alerted state after receiving alerts from T distinct firewalls. Our differential equation model in Section 3.3 can be extended to model this case by adding $(T+1)$ differential equations to track the number of firewalls that have received $0, 1, \ldots, (T-1), T$ alerts. This approach is however cumbersome even for low values of T. However, the lower bound on containment metric for the implicit signaling case can be applied:

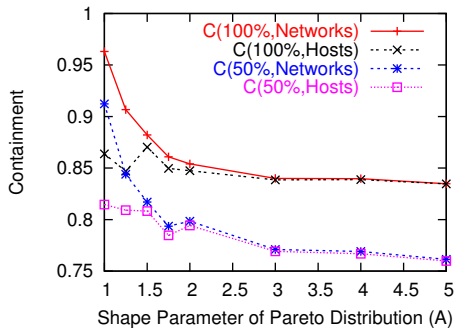
Lemma 6. *For $\lambda > 1$ and $I_0 \ll N$, the containment metric C obtained by implicit signaling and explicit signaling at rate E is at least $1 - \frac{(log(N)+(T-1)log(log(N)))t_d\sigma^2)}{(s+E)}(\frac{1}{t_d\sigma}+1)$ where $\sigma = \frac{\lambda-1}{t_d}$.*

Proof. The only change required from the analysis of implicit signaling is that the number of alerts required before the first phase finishes, has to be at least $(Nlog(N) + (T-1)log(log(N)))$ since each of the N firewalls has to receive D distinct alerts. This follows by the coupon collector problem. □

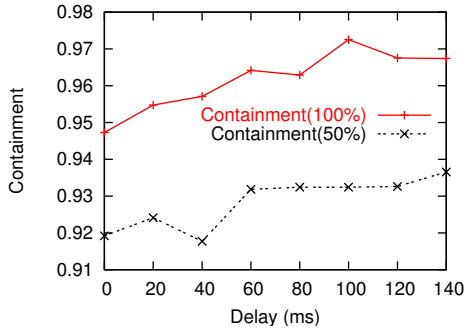
The main implication of this result is that the containment drops linearly with the threshold T. This quantifies the trade-off between the robustness of cooperation and the containment C.



(a)



(b)



(c)

Figure 3.4. (a) Containment Vs. Threshold T (b) Containment Vs. Non-Uniformity (c) Containment Vs. Network Delays

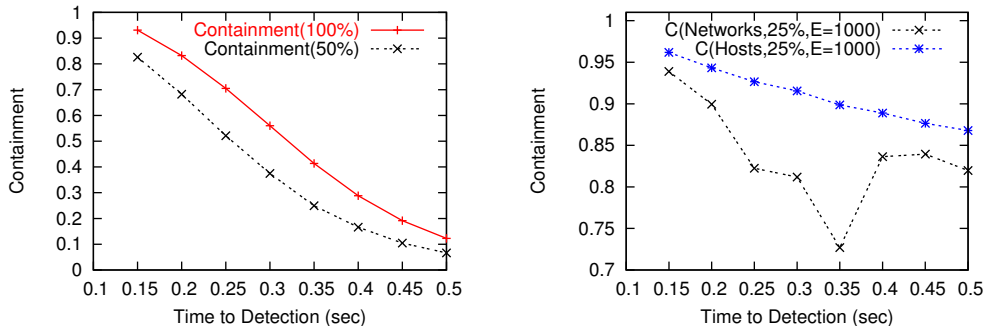
3.5.2 Numerical Results

We present numerical results for the containment in the complete deployment and partial deployment scenario in Figure 3.4(a). The containment metric is plotted against the threshold T at various levels of deployment ranging from 10% to 100%. The simulation under this scenario is as follows. Out of the N firewalls, T randomly chosen firewalls are marked as "bad" firewalls. These firewalls do not mark traffic originating from their infect. Further, if a bad deployed firewall is designated to perform analysis duties on behalf of an unemployed firewall, it does not perform those duties as well. Note that, in this simulation, bad firewalls do not propagate spurious alerts: such alerts would only improve containment since the other firewalls would receive T alerts earlier.

In the complete deployment case, the drop in containment is linear as expected by the analysis, and one can achieve a containment level of over 70% even with 1000 malicious firewalls. Under partial deployment along with rerouting, the containment metric drops super-linearly with the threshold T especially at low values of deployment. The number of malicious firewalls against which the cooperative model can provide good containment decreases to about 400 at 50% deployment, and further can only deal with about 200 malicious firewalls at 25% deployment. Note however that the *fraction* of malicious firewalls under which cooperation performs well is nearly the same.

3.6 Limitations of Modeling

We now discuss the most important simplifying assumptions in our model and their effect on our results.



(a) (b)

Figure 3.5. (a) Containment Vs. Constant time to detection (b) Containment Vs. Time To Detection (realistic environment)

Non-Uniform Host Distributions: First, our analysis assumes that the host distribution is uniform among vulnerable networks. The results of our analysis for a uniform distribution can be suitably interpreted for a non-uniform network. It can be shown that the containment metric under an uniform distribution is an *lower-bound* on the number of infected *networks* under *any* distribution, though the number of infected *hosts* may be higher. Thus, a 99% containment as derived in our model for an uniform distribution, implies that at most the top 1% highly populated vulnerable networks will be infected under a non-uniform distribution (these networks would contain more than 1% of vulnerable hosts however).

To further study this case, we also ran simulations using a non-uniform distribution for the number of vulnerable hosts in each network, and the results are plotted in Figure 3.4(b). Our choice for the non-uniform distribution was guided by Weaver et al [69], who showed that the distribution of the hosts infected by Slammer is heavy-tailed. We used the Pareto distribution, one of the most-common heavy-tailed distributions used in modeling the Internet (we could not find more detailed information about the distribution of vulnerable hosts in literature). In this plot, the shape parameter A of the Pareto distribution is varied

($A = 1$ corresponds to a very uneven distribution while $A = 5$ is nearly uniform), while the scale parameter B is chosen so that the total number of vulnerable hosts is constant. Two sets of results are shown: one under complete deployment and the other under partial deployment level of 50%. For each deployment level, we show two results for the containment metric C: in terms of vulnerable networks that escaped infection, and in terms of vulnerable hosts that escaped infection. The threshold T was set to 100, and all other parameters are set to the default value. First, observe that the network-level containment *decreases* as the distribution becomes uniform. Second, note that the host-level containment is always lower than the network-level containment suggesting that highly populated networks are infected faster. The host-level containment and the network-level containment tend towards each other as the networks becomes more uniform. This trend is observed both under complete and partial deployment.

Network Delays: Second, our analytical model does not include wide-area and local-area network delays. Their effect is illustrated in Figure 3.4(c) which shows the containment metric based on a simulation that includes both kinds of delays. The local-area network delay was set to a constant value of 1 milli-second, while the wide-area delay between the networks was varied between 0 and 140 milli-seconds. These plots illustrates that, under both complete and partial deployment, the containment metric increases slowly as the wide-area delays increases. The reason is that worm probes and cooperation signals are both delayed through the wide-area network, whereas the time for local detection remains the same (since detection is performed at the source firewall which is close to the originating end-host). As expected, this variation with wide-area delay is minimal, since fast scanning worms are bandwidth-limited and are not drastically affected by wide-area delays. Local-

area latencies are much smaller than these wide-area delays, and do not affect the results substantially.

Modeling Local Detection: Third, we have assumed that the time for local detection is a random variable with an exponential distribution. Clearly, this is not necessarily true for all local detection schemes. We also simulated the case where the time to detection is a constant value t_d. The results are plotted in Figure 3.5(a), which shows that containment exhibits similar variations with the detection time t_d as in the exponential distribution. Not surprisingly, the absolute value of the containment is much lower especially at higher values of t_d. For example, given that the worm propagation time is about 0.6 seconds, a detection scheme with a constant time to detection of 0.5 secs cannot be expected to perform well. With such a scheme, no firewall enters the detected stage before the worm has infected over 75% of its victims, and signaling beyond this time cannot provide good containment.

Real-life Results: A study of a bandwidth-limited scanning worm [69] reports that the factors which have a first-order impact on its propagation are the size and distribution of the vulnerable population, the scanning rate of the worm, and network delays. In Figure 3.5(b), we present results on the containment that includes the effect of all these factors. This plot uses a non-uniform Pareto distribution with the shape parameter set to 1, a wide-area delay of 100 milli-seconds, a local-area delay of 1 ms, a deployment level of 25%, and a threshold T of 100 malicious firewalls. The containment metric is plotted in terms of networks and hosts as the average time to detection is varied from 0.1 to 0.5 seconds. The main result is that one can protect over 90% of hosts against a highly virulent random scanning worm even under a highly skewed distribution by using only implicit signaling along with a local detection scheme with a detection period of 0.2 secs. There is still further scope to improve the fidelity of these results by including second-order effects, such as varied access bandwidths

for different networks, wide-area congestion, and hosts belonging to multiple access networks that offer covert channels of infection.

3.7 Related Work

We classify related work into four main categories. The first category includes works, *e.g.,* Vigilante [13], Throttling [65], TaintCheck [42], that use end-host instrumentation in order to detect malicious packets. In general, such host-based mechanisms can detect a wider class of worms. However, in comparison with firewall based solutions (such as ours), they involve a initial deployment cost. The second category relies on inferring and deploying filters to contain worms *e.g.,* Autograph [29], Dynamic Quarantine [73]. As the results in [73] show, since the various entities do not exchange information, filter placement within the Internet core is necessary for most of these schemes in order to acheive effective containment. The feasibility of such placement is not very clear, since routers may be hard to modify and access to in-network processing boxes is usually limited. Existing mechanisms for inferring filters can be used as filtering mechanisms in our model to achieve good containment without modifying core routers. The third category consists of detection schemes that operate at a *single observation point* (not at the end-host) in order to detect a worm attack in progress. These can serve as local detection schemes in our framework. Examples of such detection schemes include Threshold random walks [27], fast scanning worm detection [57], worm fingerprinting [58], honeypot-based mechanisms [30, 55]. We view such work as orthogonal to our own. It is possible to envision a cooperative scheme where some of the participants can be honeyfarms dedicated for the purpose of detection. The final category relates to proposed architectures for worm containment involving multiple

vantage points. Architectures, such as Wormholes [70], Wu et al's proposal [74], are based on the paradigm of multiple monitoring points redirecting traffic to a few dedicated analysis points. While such a paradigm may be useful in certain contexts, we believe that the decentralized option also holds promise since the analysis and detection load is shared equally by all firewalls. This makes it harder to launch directed attacks against the worm detection architecture itself. Decentralized designs include Hard Perimeters [68], Weaver et al [71], Domino [77]. In [68, 71], data collection and analysis are performed by firewalls, while [77] splits these functionalities between satellites and an axis overlay. All participants are implicitly trusted in these works, and in contrast, our model uses simple verification along with a thresholding mechanism to deal with untrusted participants. The implicit trust is suitable for deployment within a single enterprise, but for an Internet-wide deployment scenario, we believe our model may be more suitable. Nojiri et al [43] handles malicious participants, but unlike our model, assumes the existence of apriori trust relationships among firewalls. Anagnostakis et al [6] propose a signaling protocol very similar to our explicit signaling, but their focus is more on multiple propagating worms as against fast worms. Finally, an analytical model for worm containment is proposed by Staniford [61], which however does not study cooperation. Another closely related work is that by Moore *et al.* [39], which studies cooperative worm containment: the main difference is that our work offers analytical solutions as well as simulation-based results, whereas Moore *et al.* focus primarily on simulation-based results.

3.8 Conclusion

In this work, we have discussed a framework for modeling cooperation in order to conduct a preliminary analysis of its efficacy, resilience, and deployability. The advantage of analyzing a general framework is that our results can be applied to cooperation in a variety of scenarios: at the enterprise-level or Internet-level, at the host-level or firewall-level, and also to a limited extent for analyzing sophisticated worms like hit-list based worms. In an Internet-level firewall cooperative, our results indicate that cooperation can indeed be an effective solution in containing fast uniform scanning worms. Cooperation can be made resilient to a small number of malicious participants (100-1000), and even at low levels of deployment (25%), effective containment can be achieved.

In the exchange of information across multiple sites as we have discussed in this chapter, often there are several confidentiality constraints in sharing such information; in the next chapter, we propose an architecture that hides such context from security appliances.

Chapter 4

Hiding Context from Admins

As we have detailed in the earlier two chapters, network security mechanisms gain by analyzing network traffic for context. However, the depth of such analysis poses the risk that they expose sensitive information in network traffic to those in control of security appliances. Such exposure is increasingly of concern given the tendency to out-source administration of security appliances. Such exposure is also of concern when multiple sites cooperate amongst themselves for security purposes; as for instance, in the context of cooperative worm containment that we addressed in the previous chapter. In what follows, we propose an off-path architecture for dealing with the issue of exposure of sensitive information to intrusion prevention appliances.

4.1 Introduction

Two important security goals in today's networks are: (a) to protect internal machines from remote attacks; this goal is served by intrusion prevention appliances that vet incoming network traffic. (b) to prevent the leakage of confidential information in network traffic;

this is achieved by end-to-end encryption of network traffic. However, there is an inherent conflict between these goals: *intrusion prevention appliances cannot analyze encrypted traffic* since such appliances increasingly rely on access to clear-text application payload for high accuracy analysis.

Most existing networks choose *one of these security goals and sacrifice the other.* In networks where *protecting internal machines* is deemed more important, encrypted connections are either disallowed, or it is stipulated that relevant keys be revealed to the appliance; this however exposes sensitive data in network traffic to those in control of those appliances (network admins, and increasingly, third party security companies). Given the increasing trend to out-source security in various enterprises, this represents a serious risk since third-parties outside the company may misuse the data. In networks, such as financial and legal enterprises where *confidentiality* is deemed more important, security appliances simply skip over encrypted connections; this incurs the risk of exposing end-hosts to attacks over encrypted channels.

The problem of resolving this conflict without completely sacrificing either goal has received IETF attention in the context of ensuring that network-layer middleboxes (such as NATs, load balancers, firewalls) can inspect the traffic in clear-text in order to carry out their functionality. This attention has also been directed for other reasons as well (*e.g.,* to avoid the use of two port numbers per protocol). The general solution is to re-write the application and re-design the protocol so that the option to encrypt the traffic can be negotiated. In this fashion, the information necessary for the operation of the middle-box can be sent in the clear before encryption is used.

While the IETF approach concerns a wide variety of network-layer middle-boxes, in

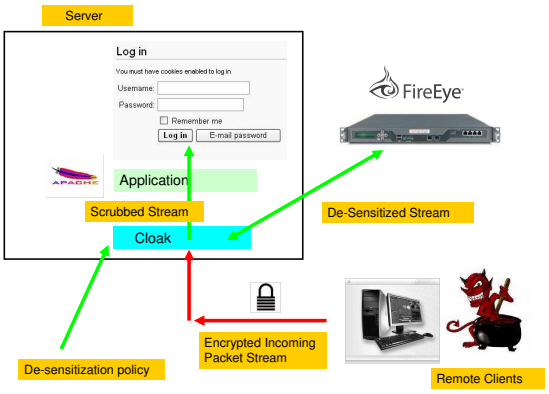


Figure 4.1. Apache HTTPS server.

the context of intrusion prevention devices, it has several limitations. First, every protocol needs to be re-designed and this requires a *standardization process* so that server and client implementations can agree on what portions of a connection should be encrypted. Second, applications need to be *re-written* to adhere to the new standard. This is a serious limitation given that several deployed applications rely on whole-sale encryption using SSL/TLS[1]. The need for *both ends of the communication to deploy* the rewritten software also poses another hurdle. Third, while the goal is to expose clear-text to the security appliance, the IETF approach unnecessarily exposes clear-text to all eavesdroppers on the wide-area network path.

We present the design and implementation of Cloak, a *server-side* solution that lets a server admin[2] hide passwords in her encrypted traffic from security appliances while still vetting the rest of her encrypted traffic using the appliance. The key design philosophy in Cloak that differentiates us from the IETF approach is that *we do not compromise on end-to-end encryption* so that in-network inline intrusion prevention devices can vet the clear-text portion of network traffic. Instead, Cloak moves the intrusion prevention functionality

[1]The TLS protocol is the standardized version of SSL which differs from it in only a few technical details.
[2]The term **server admin** refers to the individual hosting the server; the admin monitoring the network is the **network admin**.

off-path and relies on *host software* to furnish the clear-text version of encrypted traffic to the security appliance. This version is however *de-sensitized*: Cloak relies on configuration to identify and obfuscate any passwords embedded in this traffic. Note that passwords are usually sent from clients to servers, so the password exposure problem due to vetting of *incoming* traffic is at the server side; for this reason, Cloak is deployed at the server.

Figure 4.1 illustrates the use of Cloak by a HTTPS server admin who offers a SSL-based service that authenticates clients using passwords. She would like her incoming traffic to be sanitized by devices provided by Counterpane [1] (a third party security company). To prevent the leakage of passwords to the third party who might mis-use them to access internal services and data, she specifies the authentication method in use to Cloak (HTTP Basic/Digest authentication or POST-based authentication). Cloak then de-sensitizes the clear-text incoming traffic stream before furnishing it to the security appliance for sanitization. Only traffic allowed by the appliance is delivered to the application. We note that though the remote clients are trusted to access the service, since their computers may themselves be compromised, it is still necessary to vet traffic sent by clients.

The realization of our design philosophy presents three significant challenges:

- Retaining high fidelity: Today's advanced intrusion prevention devices use complex techniques such as virtual machine analysis that replicate the application configuration at the end-host for high accuracy attack detection. Cloak needs to ensure that its obfuscation minimally affects the fidelity of attack detection.

- Work with unmodified applications: Due to the large base of existing applications using the TLS/SSL protocol (the industry standard protocol for end-to-end authenti-

cation and encryption in the wide-area), Cloak should not require any modifications to existing applications.

- Minimal trust in host software: The reliance on host software should be carefully examined since it would be the target of attacks by adversaries.

We address the fidelity issue by only minimally altering the traffic stream; only the password field is obfuscated. We rely on deterministic obfuscation schemes that can optionally preserve certain characteristics of the password to achieve fidelity. Our qualitative analysis and empirical analysis of known vulnerabilities show: (a) If the exploit condition does not involve the password, Cloak's obfuscation will not lead to the attack being missed by the appliance. In our study of the CVE repository [2], we found that only 2 out of 357 known attacks in Apache and 1 out of 27 known attacks in ProFTPD involve the password field. Thus Cloak's obfuscation would lead to very few attacks being missed in practice. (b) If the vulnerability can be triggered via the password, then length-preserving obfuscation policies can help catch overflow attacks, at the cost of revealing the length of the password. However, other classes of attacks (such as format string attacks) may be missed since the password characteristics that trigger the vulnerability (say, the presence of "%n") may be exactly characteristics one wishes to hide from the security appliance[3]. This appears to be a fundamental limitation given the secrecy requirement for passwords that we aim to achieve.

Cloak needs to intercept clear-text so that it can be obfuscated before being sent to the security appliance. In order to do so without application modifications, Cloak relies on library interposition techniques. Most popular Linux servers rely on the OpenSSL library

[3]It is possible to use secure multi-party communication techniques along with zero knowledge proofs to avoid such leakage; however, transforming real-life programs into a boolean program so that these techniques can be applied is currently infeasible.

for end-to-end encryption, so Cloak interposes its functionality over the OpenSSL library. In order to minimize the security implications of host software, we seek to simplify the design of Cloak as much as possible. Cloak only aims to protect passwords and other sensitive data in the application payload; since they are not revealed via lower layer fields (such as IP headers, TCP headers, encryption protocol headers), Cloak relies on a traditional in-line security appliance (a firewall or intrusion detection system) to screen vulnerabilities in the layers of software below Cloak. Our implementation is simple (less than 1000 semicolons) thus reducing the chances of vulnerabilities and errors.

Our proof-of-concept implementation of Cloak is based on modifications to the industry standard OpenSSL library, which lets us support Apache and ProFTPD, widely used Linux HTTP and FTP server software. The obfuscation mechanism in Cloak is protocol-specific; our prototype supports obfuscation policies for passwords in HTTP and FTP protocols. Since we do not have access to a third party security appliance, we used a substitute open-source appliance that detects attacks using taint analysis. Such appliances are increasingly useful in today's networks given their ability to catch *unknown* attacks; hence we experimented with them. Further, they expect a high fidelity input stream conforming to that seen by the server itself: thus they represent a hard test case for Cloak's obfuscation. This appliance runs an instrumented version of the application at the server; the instrumentation was done using the TaintPolicy tool [76] by Xu *et al.* . The latency overhead due to Cloak processing is minimal (183 microseconds); the end-to-end latency overhead is dominated by the network latency between the server and the security appliance.

The design and implementation of Cloak are fairly straightforward; our main contribution is an architectural solution to address the burgeoning problem of password exposure without compromising on end-to-end encryption. Given the increase in out-sourcing of se-

curity appliances, it is important to address the conflict between confidentiality and security in a timely fashion without the *necessity for IETF standardization* or *re-writing applications* or *the sacrifice of end-to-end encryption*. Passwords can be misused by untrusted parties to access internal services and data; Cloak prevents such exposure without significant loss in fidelity. Though there may be various kinds of sensitive information embedded in network traffic, passwords are the most generic piece of information across several protocols that is valuable. The risk in cloaking too much information (such as files transferred over FTP) from the security appliance is that various attacks may be missed, and offer adversaries an easy loophole. We believe password obfuscation helps us achieve a *sweet spot* in the trade-off between confidentiality and security; there is little loss in fidelity (very few attacks missed in practice) at the gain of the confidentiality of extremely valuable information. It is for this reason that we chose to focus on passwords. If additional de-sensitization is desired and the resulting loss of fidelity is acceptable, existing application payload rewriting frameworks can be plugged in to our implementation to achieve this.

The rest of the chapter is organized as follows. We describe three motivating scenarios in Section 4.2. We then describe our design goals (Section 4.3), design (Section 4.4), and implementation (Section 4.5). We then present our security evaluation (Section 4.6) and performance evaluation (Section 4.7). We discuss our limitations (Section 4.8), discuss related work (Section 4.9), and then conclude and discuss future work in Section 4.10.

4.2 Three Motivating Scenarios

Scenario #1: Consider enterprises that have out-sourced the security appliance functionality to a third-party. This scenario is becoming increasingly common for various rea-

sons: (a) the difficulty in maintaining security appliances up to date; (b) the convenience factor in deployment, particularly for small and medium-size companies; (c) the visibility advantage that a third party has from collating information across various networks. The reasons for this trend are argued for, for instance, by Scheneir [11]. Appliances provided by companies like FireEye [3], Counterpane [1] etc are now being used by various enterprises. However, passwords are often embedded in network traffic. Internal computers in enterprises host various services, such as a web server, file server, remote desktop, and remote login, for convenient access by their owners. In this case, both the server admin and the network admin have incentive to avoid leaking passwords to the third-party; the server admin wants to constrain access to her service, while the network admin wants to prevent misuse of enterprise resources by third parties. Cloak would be useful in this context to obtain the best security under the constraint that passwords not be divulged.

Scenario #2: There are several institutions where end-to-end encryption is required by legislation. For instance, the Health Insurance Portability and Accountability Act (HIPAA [4]) forbids hospitals from allowing network access to health information without end-to-end encryption. The financial industry has the Sarbanes-Oaxley act [5] which has similar provisions. In these cases, the need to avoid leakage of passwords is crucial since it can be mis-used to access health records and financial data. Cloak can thus help obtain secrecy for passwords along with high fidelity attack detection while at the same time allowing end-to-end encryption for the rest of the data.

Scenario #3: Consider institutions where internal users can act as server admins and a trust scenario where the server admin does not trust the network admin running the security appliance. This includes educational institutions like UC Berkeley. In such cases, encrypted traffic is not inspected by security appliances due to privacy implications; Cloak would be

useful for server admins who can have their traffic sanitized by remote appliances while being assured that their services will not be misused by untrusted parties with access to authorized passwords.

Though there may be various sensitive information embedded in network traffic, passwords, in particular, are extremely valuable. If too much information (say, email contents) is cloaked from the security appliance, then vulnerabilities in such channels can be easily exploited by adversaries. Password obfuscation offers a high value confidentiality gain at the loss of little fidelity. For this reason, we chose to focus on passwords.

We note that the problem of protecting *passwords* from *administrators* of *remote* security appliance is widely prevalent in many enterprises. First, *passwords* continue to be the most widely used security mechanism, particularly for web services. Despite alternative mechanisms such as client certificates (or client keys or challenge-response verification of passwords that is resistant to replay attacks), passwords dominate due to their convenience. Second, though it is tempting to trust security personnel (either inside the enterprise or outside), we note that the incidence of threats from security personnel and admins is increasing. Specifically, the importance of keeping passwords safe from network admins is highlighted in a survey by CyberArk [17], a security management company. Several anecdotal evidences of such breaches is presented in this articles where security personnel have misused access for their own ends. Third, *remote security appliances* continue to dominate the enterprise security landscape for three reasons: (a) Devices like laptops and enterprise desktops may not be powerful enough to run virtual machine analysis techniques (such as Shadow Honeypots [7]); thus, remote analysis helps reduce overhead on the server. (b) Convenience of deployment; an appliance that monitors multiple end-hosts is easy to deploy, and can be out-sourced conveniently. (c) Anomaly detection techniques (such as EarlyBird [58])

represent a promising direction for dealing with zero-day vulnerabilities that do not have any specific behavioral signature (such as, control data being over-written with network data). Such techniques need a large volume of data for their analysis. Given the potential for fast worms, they require online access to information from a number of end-hosts, and this requires remote analysis.

4.3 Design Goals

Our security goal is to scrub encrypted server traffic of sensitive information (particularly, passwords) before vetting such traffic by remote intrusion prevention appliances. We now detail our threat model and design requirements.

Threat Model: The goal of Cloak is to protect passwords from the party who maintains the security appliance. This may either be the third party or the network admin. We assume that the third party or network admin has complete control over the security appliance, but does not have access to the server keys. This threat model applies to all our three scenarios. In the outsourcing scenario (scenario #1), the enterprise (as well as the server admin) wishes to avoid exposing passwords to the third party who controls the security appliance. In the health/financial enterprise scenario (scenario #2), the enterprise wishes to avoid exposing the password to the network admin in control of the appliance. In the university scenario (scenario #3), the server admin wishes to hide passwords from the network admin.

Policy Control: In scenario #1 and #2, where Cloak is deployed to ensure enterprise policy, we assume Cloak is installed and configured by an enterprise-wide authority. This avoids incorrect configuration of obfuscation policies at servers. This is enforceable in most enterprises since employees can only install applications from a select list; thus Cloak can

be installed with a pre-set obfuscation policy. In scenario #3, we assume Cloak is installed and configured by the server admin on her server.

Design Requirements: We have five main requirements:

- *High fidelity:* The security appliance should be able to operate with high fidelity even on the obfuscated stream. By high fidelity, we mean that the set of attacks that the appliance can prevent by operating on the obfuscated stream should be close to the set of attacks it can prevent by operating on the clear-text stream. We only aim to provide high fidelity here under the constraint that the password not be exposed to the appliance.

- *Support for legacy applications:* Due to the large base of existing applications using the TLS/SSL protocol, Cloak should not require any applications to be modified. Such re-writing requires programmer effort, and errors may be introduced in such re-writing; for this reason, Cloak aims to avoid application re-writing.

- *Analysis-neutral:* Cloak should minimally constrain the analysis by the remote appliance. This is important since shadow honeypots (these are security appliances [7] that mirror the application configuration at the server for high fidelity analysis; modern intrusion prevention systems are increasingly incorporating such technology) expect a complete copy of the traffic stream conforming to that expected by the application.

- *Support for unmodified clients:* Cloak should not assume any support from the remote client; it should only require deployment at the server end. Cloak cannot rely on clients being willing to make modifications.

- *Overhead:* The system should minimally impact the end-to-end performance seen by the client.

Assumptions: We make two assumptions:

- We assume that the server is not already compromised. Otherwise, passwords are already at peril and and there is little benefit in vetting traffic to an already compromised machine. Thus, this assumption is reasonable in the context of intrusion prevention.

- We assume a traditional in-line appliance (such as a firewall or intrusion prevention system) is deployed to handle vulnerabilities in the layers of software below the application layer (encryption, transport, network, OS layers). Since Cloak aims to protect sensitive information, such as passwords, that is encoded in the application payload, such appliances that only process lower-layer headers are compatible with our secrecy requirement. This is already the case in most networks, therefore no changes are required to accommodate Cloak.

4.4 Design

The Cloak architecture is shown in Figure 4.2. This figure illustrates the primary difference between Cloak and the typical in-line deployment of security appliances. Cloak moves the security appliance for detecting application-layer vulnerabilities in encrypted connections off the network path; it relies on the traditional on-path appliance (such as a firewall or IDS) for defending against vulnerabilities in the lower layers, as well as, vulnerabilities in un-encrypted connections. If desired, these two appliances can be implemented on the same

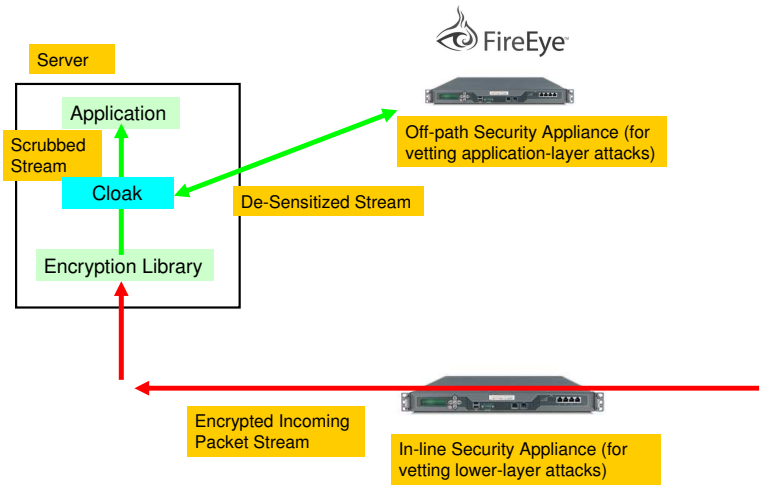


Figure 4.2. Cloak.

device; our architecture only logically distinguishes between them. Cloak uses host software interposed over the encryption library to ensure that the encrypted traffic is sanitized by the off-path appliance before being sent to the application. We now justify these design decisions, and then describe the design of the components of Cloak in detail.

4.4.1 Off-Path Architecture

We now argue that the off-path placement of the appliance that detects application-layer vulnerabilities meets our design requirements. Also, from hereon, we use the term *appliance* to refer to the appliance that detects application-layer vulnerabilities.

Since the client application is not modified, the incoming traffic stream is constrained to be encrypted end-to-end. Further, we would like end-to-end encryption to be preserved so as to avoid exposure of any information to eavesdroppers. This implies that any in-line element (whether implemented as a transparent man-in-the-middle appliance or a explicitly configured proxy), by definition, can either read the entire traffic stream (if provided with

the appropriate key) or not be able to read any of it. Thus, the only way to protect passwords and yet reveal the rest of the content is to move the appliance off-path.

The need for off-path placement of the appliance implies that software running at the host[4] is required to furnish traffic to the appliance for sanitization. This reliance on host software exposes our system to vulnerabilities in the host software itself. To this end, we aim to keep the design of Cloak simple so that the possibility of vulnerabilities can be minimized. Further, since Cloak is designed for intrusion prevention, we assume that the server is not compromised; thus Cloak is not circumventable by incoming traffic.

Since the host software needs to be able to capture the un-encrypted stream without any modification of the application, we chose to layer it above the encryption layer. An alternative is for Cloak to run in, say, the kernel, underneath the encryption layer. This would require the server admin to entrust Cloak with the server key. However, in this option, Cloak needs to implement decryption in order to intercept the clear-text; thus, one needs to rely on the encryption library anyway. Thus, we avoid this issue by implementing Cloak using library interception.

For these reasons, Cloak is implemented in host software that interposes itself over the encryption library. We now explain the components of the host software and then examine the security implications of relying on host software.

4.4.2 Host Software: Architecture

Cloak has three main components: the data capture layer, the password obfuscator, and the security module. Figure 4.3 shows these components running on the server alongside

[4]An alternative deployment mode is to run Cloak at a gateway. We discuss this option in Section 4.8.

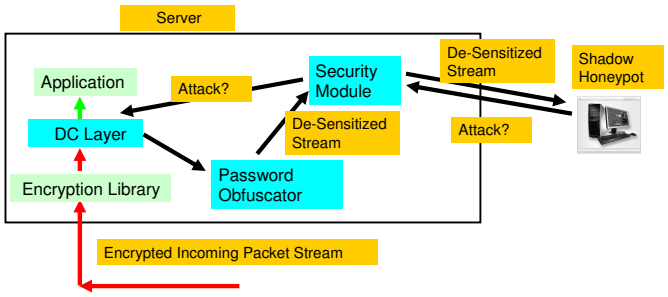


Figure 4.3. Host Software Architecture of Cloak.

an application. The data capture (DC) layer captures the clear-text traffic stream on all incoming connections to the server. The password obfuscator identifies embedded passwords in the incoming stream and obfuscating them; this is controlled by a obfuscation policy. The stream is then handed off to the security module which conveys it to the remote appliance (in this example, a shadow honeypot) for sanitization. If the appliance detects an attack, then the security module conveys the decision to the DC layer which ensures that the malicious traffic is not sent to the application. We now discuss these modules.

The DC layer

The DC layer processes packets per the following pipeline: (a) intercept the incoming clear-text traffic stream on all incoming connections (b) assemble the traffic into application data units (ADUs) (c) convey the ADU to the remote appliance for sanitization (via the obfuscator and the security module) (d) deliver the ADU to the application if no attack is detected by the remote appliance; if an attack is detected, drop the offending ADU. We note that we process the stream in ADUs since our obfuscator operates in terms of ADUs (it may have to update multiple fields in the ADU when it implements obfuscation). We now discuss each of these steps in turn.

Incoming Traffic Interception: The DC layer intercepts the calls to the encryption library used for: (1) accepting a new connection from a client. (2) for reading and writing data on connections. (3) for closing connections. Most end-to-end encryption libraries (including OpenSSL, the most widely used library) support these basic calls in one form or the another. The DC layer intercepts the calls for accepting a new connection and for closing a connection since it maintains state (particularly, buffers) for every new connection. This state is initialized for every new connection, and then re-claimed on connection shutdown.

The calls for reading and writing data are intercepted since the clear-text is sent as arguments to the read and write calls by the application; this gives the DC layer access to the clear-text. We note that we only aim to capture the application layer payload (and not IP headers, TCP headers); this is sufficient since the traditional in-line appliance screens for attacks in those layers.

ADU assembly: ADU assembly is the application protocol-specific module in the DC layer. However, this task is easy to achieve given minimal protocol knowledge. For several commonly used protocols on the Internet (*e.g.,* FTP, IMAP, POP-3, NNTP), this can be done by using a simple delimiter and length fields to assemble ADUs (*e.g.,* "`\n`" for FTP, IMAP, NNTP, POP-3). Binary protocols (such as RPC, CIFS, SMB) typically specify the length field near the start of the ADU; this is used to parse the stream into ADUs. The protocol in use is specified via a configuration file, and if it is a protocol supported by Cloak, the appropriate assembly mechanism is chosen.

Sanitization by the remote appliance: Intrusion prevention devices are concerned with sanitizing incoming traffic, and do not attempt to regulate outgoing traffic. For this reason, outgoing ADUs (intercepted via the write calls made by the application) are passed

without delay to the underlying encryption library. Incoming ADUs (intercepted via the read calls made by the application) need to be sent to the remote appliance for sanitization before they can be sent on to the application.

This sanitization can be performed in two possible ways. The first is to treat the appliance as an in-line element in the data processing path; thus one could deliver all incoming ADUs to the appliance and then read the sanitized version from the appliance. The second is to use the appliance to sanitize ADUs without involving it directly in the data forwarding path. In this option, incoming ADUs are buffered and a copy is sent on to the appliance; the appliance notifies the DC layer as to whether the packet is safe/not.

We chose the second design option since it saves on network overhead. The first option involves the additional overhead of ADU delivery from the appliance back to the host, wherease the second only sends the notification message, which is much smaller. Further, the second option is advantageous from the security perspective as well. Since only a copy of the packet is sent to the appliance, we do not need to implement any additional integrity mechanisms to ensure that the appliance does not tamper with the data.

Delivery to the application: Once an ADU is sent to the appliance, the DC layer awaits the notification from it (sent via the security module). If the ADU is deemed safe, it is sent on to the application. Otherwise, it is dropped so that the application is not compromised. Further, we terminate the connection so that no further packets are processed on the offending connection.

Password Obfuscator

The password obfuscator identifies the password field in the assembled ADU and replaces it with the obfuscated version. This obfuscated version is conveyed to the security module.

We note that our password obfuscator can leverage existing application payload rewriters (*e.g.,* Pang and Paxson's flexible packet-rewriting framework [46]) as plug-ins so that additional elements in the payload can be obfuscated based on a configuration policy. These frameworks are configurable as to which elements of the application payload should be obfuscated (*e.g.,* files tranferred via FTP, the URL portion of HTTP requests). We chose to obfuscate the password field only so as to maintain high fidelity. We now explain the two steps in our password obfuscator.

Identifying the password field: This step requires protocol knowledge along with minimal configuration in order to identify the location of the password in the ADU. For a fairly broad set of protocols (*e.g.,* FTP, IMAP, POP-3, NNTP), the password is embedded in the network traffic only in an ADU specifically meant for conveying the password; thus it is easily identified. For instance, the FTP protocol supports the "PASS" command which is sent as an ADU with the text "PASS" followed by the password itself. For such protocols, the password obfuscator does not require any configuration at all; Cloak incorporates protocol knowledge to achieve this.

The complex case is for HTTP where there is a variety of ways in which the password can be encoded depending on the authentication method: HTTP Basic/Digest Authentication or script-based authentication. In the first method, the password is embedded according to the HTTP standard, and here Cloak relies on protocol knowledge of HTTP to identify the password. In the second, the embedding of the password is dependent

on the authentication script in use. For instance, when the POST method is used to send the password to the server, the password is encoded using a string such as "username=loginname&password=loginpassword" in the body of the POST request. In this case, we rely on minimal configuration (specification of the name of the parameter in the webform encoding the password; in this case, "password") to help Cloak identify the password.

Obfuscating the password field: Once the field has been identified, the value of the field in the ADU is obfuscated, and then the rest of the ADU is updated accordingly (if the length of the password is altered, it might require updating the length field in the ADU, for instance). The challenge in such obfuscation is that (a) we would like the fidelity of detection to be maintained (b) we would like to avoid revealing any characteristics about the password.

The obfuscator can be configured to choose one of two obfuscation functions: (a) *length-preserving obfuscation*: this policy replaces the password with a random value of the same length. (b) *complete obfuscation*: these policies replace the values of sensitive parameters in the traffic with random values of random lengths (within protocol constraints). Length-preserving obfuscation leaks the length of the password, but in return, it can provide better fidelity since overflow attacks in the password can be identified. Both of these obfuscation schemes are *deterministic* schemes based on hashing; this determinism helps improve the fidelity of detection. We will address the subject of fidelity offered by these obfuscation schemes in our security evaluation section.

Security Module

The security module is responsible for conveying de-sensitized ADUs to the remote appliance for sanitization. For this purpose, it initializes a pool of communication channels with the remote appliance (*e.g.,* a shadow honeypot) at startup. Upon receiving an incoming ADU on a new incoming connection, it chooses a communication channel from this pool, and conveys the de-sensitized ADUs to it. Note that we use SSL for this communication to avoid exposing the ADU to eavesdroppers. The honeypot appliance responds with a "yes/no" indicating whether it is safe to convey the ADU to the application. This decision is conveyed to the DC layer which then enforces this decision by either sending the buffered ADU onto the application or terminating the connection.

Security Implications

Having examined the components of the host software, we now mention a pertinent implication of our design choice to layer the DC layer over the encryption library. As stated before, vulnerabilities in the OS are handled by the traditional in-line appliance. Handling vulnerabilities in the encryption layer, in particular, requires greater care. If the vulnerability can be detected without access to the key (say, the vulnerability is in the SSL record protocol and in the initial handshake phase), then an in-line appliance can detect such vulnerabilities. On the other hand, it is possible that the vulnerability relates, for instance, to how decryption of a certain encoded portion of a message is handled by the library. For such vulnerabilities, the in-line appliance may be limited in its fidelity since it does not have access to the server key. Cloak itself would be invoked only after the decryption has been performed by the library, by which time the vulnerability would have

been exploited. This is a fundamental limitation of Cloak given the secrecy guarantee we wish to achieve: the encryption library needs to be trusted concerning vulnerabilities related to the key and fine-grained processing by the encryption library.

4.5 Implementation

Our Cloak prototype works with most recent versions of Apache (above version 2.0) and ProFTPD (above version 1.3), widely used HTTP and FTP server software.

The DC layer is implemented as a patch of 16 semicolons of code to OpenSSL `0.9.7c`. We chose to modify OpenSSL since Apache, ProFTPD, wu-ftpd, PureFTD, Exim, Sendmail, and several other Linux servers use the OpenSSL library. Note that since our modifications are simple interception hooks, our patch can be extended to other OpenSSL versions since the OpenSSL API has not substantially changed. The password obfuscator supports the HTTP and FTP protocol. The security appliance we demonstrate is a shadow honeypot that runs an instrumented version of the application for performing taint analysis. This instrumentation was done using the TaintPolicy tool [76] by Xu *et al.* . This appliance is configured with taint-based policies to detect a variety of exploits, including, memory errors, format string attacks, SQL injection attacks, and cross-site scripting attacks. Such appliances are increasingly useful in today's networks given their ability to catch *unknown* attacks; hence we experimented with them.

4.5.1 DC Layer

Our DC layer captures the clear-text packet stream by intercepting the `SSL_accept()`, `SSL_read()`, `SSL_write()`, and `SSL_close()` calls in the OpenSSL API. The addition to

`SSL_write()` is simple since we do not need to send outgoing traffic to the remote appliance. The `SSL_read()` instrumentation is somewhat tricky since the incoming application data unit (ADU) may be split across multiple SSL records, but we cannot deliver any part of the incoming stream to the application until the entire ADU is sanitized by the honeypot. To handle this issue, we leverage the fact that applications, by definition, only process data in terms of ADUs; thus, we can block on a `SSL_read()` call until the entire ADU is received. This does not affect correctness since the application will not respond to the message until the entire ADU is obtained. This blocking does not affect throughput significantly since Apache (and most Linux servers) forks off a different worker process or spawns a new thread for each incoming connection; thus other ongoing communication is not affected. For HTTP, we currently identify ADUs by looking for consecutive occurrences of "`\r\n`" byte pattern; we currently do not handle chunked encodings and mime encodings. Once the ADU is assembled, it is handed off to the password obfuscator. We note that a similar mechanism works for FTP as well; the new line character can be used to recognize the end of an ADU.

4.5.2 Password Obfuscator

We discuss the issues in identifying the password field for HTTP (for FTP, identifying the password is much easier; it is specified as an argument to the PASS command). The various authentication mechanisms typically used over HTTP are: the HTTP Basic Authentication option (described by Fielding *et al.* in the HTTP RFC 2616 [20]), the HTTP Digest authentication method, and a custom application-layer mechanism that passes the username and password entered in a webform to a authentication script (either via a POST or GET request).

In the HTTP Basic Authentication method, the username and password are embedded

as a Base64-encoded string specified along with the "Authentication:" option in the HTTP header. In the Digest authentication method, the username and password are encoded as a MD5 hashed string in the header (note we would still like to avoid leaking this data, since someone who is privy to the data can use a simple replay attack to gain access to the system). For the authentication script, they are typically specified as parameters to the POST request (we note that the POST request is recommended by the HTTP RFC [20] over the GET request for sending passwords to avoid exposing the password via the URL; besides, POST is recommended for operations that update state at the server). Note that in all these cases, SSL is expected to be used to encrypt the HTTP connection; this is the reason passwords are left in the clear or as simple hashes.

We now describe how we handle the HTTP basic authentication method. In this method, the web server requests the browser to authenticate itself; the browser then displays up a dialog box requesting for a username and the password. This username and password are Base-64 encoded into the HTTP header by the browser and sent to the server. Apache then consults its configuration file which points to a file (".htaccess") listing the usernames and passwords of clients authorized to access that resource. In this case, the configuration file for Cloak specifies the option "HTTP-BASIC-AUTH", upon which the appropriate password obfuscator is used. Once an incoming ADU is received, the obfuscator checks to see if the authentication header option is present. If so, it then extracts the password from the base64 encoding, and replaces this with the base64 encoding of the obfuscated password. A similar mechanism works for Digest authentication method (in this case, the hash of the password in the ".htaccess" file is replaced by an obfuscated version). We note that, in the original RFC specification, a simple MD5 hash of the password was used; this lends itself to replay attacks. This has been revised to include both client nonces

and server nonces, in which case replay attacks are not an issue. However, not all browsers support this option, so the HTTP Basic authentication along with SSL is still widely used.

In the POST method, the password is sent along with a "CONTENT-TYPE" header option set to "application/x-www-form-urlencoded". The password is encoded in the request body (*e.g.,* "username=name&password=pwd"). In this case, the server admin specifies the parameter names under which the password is encoded; this can be obtained from the HTML source for the webform. Cloak uses this configuration to update the corresponding fields, and also updates the "CONTENT-LENGTH" option in the header request (if required). This method is widely used across several web services (*e.g.,* Gmail, Facebook etc).

We note that the interface between the password obfuscator and the rest of the modules is simple: it involves a single call to re-write packets. This ensures our obfuscator can be replaced by existing packet rewriting frameworks for de-sensitizing other traffic elements.

4.5.3 Interaction with Honeypot

Since the shadow honeypot is simply an instrumented version of the application, the honeypot module establishes a TCP connection to this instrumented version pretending to be a client, and sends the ADU. Cloak pre-establishes the TCP connection upon initialization for performance reasons. Note that the shadow honeypot also runs Apache/ProFTPD with the end-to-end encryption option enabled so as to simulate the configuration at the host; so this TCP connection is made using SSL (this also avoids exposing the de-sensitized ADU to on-path eavesdroppers). Once the ADU is sent, the shadow honeypot can respond in two ways: (a) if the ADU is detected as a possible exploit, the instrumentation that

detects the overflow sends a notification to the honeypot module. (b) if the ADU does not cause a buffer overflow, the instrumented Apache responses like regular Apache with the response to the webpage.

To deal with both cases, the honeypot module invokes the `select()` call on a UDP listener socket and the SSL socket used to connect to the honeypot. This is tricky since the SSL socket is not a raw UNIX socket; it is an encapsulation over the raw socket, and thus the UNIX *select()* system call cannot be used. Neither does SSL offer a select-like call we could use. We found Rescorla's tutorial on SSL [19] useful in dealing with this issue. Rescorla's solution is to extract the underlying UNIX socket encapsulated by the SSL socket, and then invoke *select()* on this socket. There are some additional complications here due to the fact that `SSL_read()` can only deliver chunks of data in SSL records, while the raw socket delivers the encrypted stream itself. We refer the reader to Rescorla's tutorial for more details; we simply borrow his solution. Thus, if the honeypot module hears back a UDP message from the honeypot proclaiming an exploit ADU, this ADU is dropped, and not delivered to the application. On the other hand, if it hears a response from the honeypot over the SSL connection, it delivers the ADU to the application.

4.6 Security Analysis of Cloak

We now argue that Cloak meets the goal of protecting passwords from the security appliance. We then present an analysis of the fidelity under Cloak and then finally argue that Cloak does not introduce any new vulnerabilities. The summary of our analysis is as follows.

We show that with some configuration effort to help identify the location of passwords

in the application payload, Cloak meets the goal of protecting passwords for HTTP and FTP. Our analysis argues that this holds for other protocols like POP-3, IMAP as well. Regarding fidelity, we arrive at two conclusions: (a) If the exploit does not involve the password as the vector, Cloak's obfuscation will not lead to the attack being missed by the shadow honeypot, provided the configuration at the server is reproduced carefully at the honeypot. Our study of CVE vulnerabilities [2] shows that out of the 357 reported vulnerabilities in Apache, only 2 involve the password field as the vector. Out of the 27 reported vulnerabilities in ProFTPD, only 1 relies on the password field as the vector. (b) If the exploit does involve the password, then our length-preserving obfuscation policies can help catch overflow attacks. 1 of the 2 password-based vulnerabilities in Apache and the one password-based vulnerability in ProFTPD are based on buffer overflows and fall in this class. For other classes of attacks (such as format string attacks, SQL injection attacks, cross-site scripting, command injection), preserving the length of the password does not suffice. It is feasible to devise obfuscation policies that ensure that such attacks can be caught as well; for instance, some of these attacks can be caught provided specific characters are left un-obfuscated since they rely on control characters like "%" (format-string) or ";" (SQL-injection) being part of the input. Such a policy would help us catch the second password-based vulnerability in Apache as well which exploits a cross-site script injection vulnerability. However, such obfuscation policies also leak considerable information about the password as well. For this reason, we have chosen not to support such obfuscation policies.

4.6.1 Confidentiality Guarantee

The main difficulty in this analysis is that the appliance receives a complete copy of the incoming traffic, and thus, we run the risk that the client leaks the password through fields that we do not de-sensitize. To ensure this does not occur, we rely on configuration in the case of HTTP, and on protocol knowledge in other cases.

HTTP server: The password is typically used only in three contexts for a web service: (a) the remote client specifies it during authenticating when accessing a privileged resource (b) the remote client specifies it when changing the password via a "change password" page (c) the remote client specifies it during the registration process when creating her username and password. We require that the server admin specify the authentication method (Basic Authentication / Digest Authentication / POST authentication) in the Cloak configuration for each of these accesses, along with the obfuscation function; Cloak will then obfuscate the password in all accesses. We note that it is rare for the password to be revealed via other fields in the network traffic. Of course, if the client types in the password, say in an email sent via a web-form, there is little Cloak can do to avoid leakage in that case; this is fortunately rare. Note that simply scanning for the text of the password in the application payload and obfuscating it is not a viable option since the password may occur accidentally in the payload.

FTP server: We rely on our knowledge of the FTP protocol to obfuscate all occurrences of the password in the FTP packet payload; we rely on configuration only to specify the protocol type (FTP). The FTP password is typically sent by the client during the login phase in the form of an argument to the PASS command. This command is sent over the FTP control channel. Our obfuscator duly obfuscates passwords embedded in such

commands. In typical use, the password field is only exposed via this command. Though FTP allows file transfers, passwords are not typically placed in the clear in files; so sending files un-obfuscated to the appliance would not compromise the secrecy of passwords.

Other services: Our preliminary analysis of other network protocols such as mail services (IMAP, POP-3), directory services (LDAP), and news services (NNTP) suggests that simple obfuscation policies suffice to capture the password fields in these protocols. All these protocols have been standardized and the password only occur in well-specified locations in the payload; our password obfuscator can use the protocol specification to identify these locations. Even for binary protocols like SMB, one can write a simple password obfuscator since, by design, the protocol incorporates passwords in only select message types.

4.6.2 Fidelity Evaluation: Known Attacks

We performed a empirical analysis of attacks in Apache and ProFTPD. We found that out of the 357 Apache vulnerabilities and 27 ProFTPD vulnerabilities reported to the CVE [2], only 2 and 1 entries respectively report that the vulnerability is exploitable via the password field. The two vulnerabilities in Apache are: (a) CVE-1999-1237: A long password is the exploit vector that triggers a buffer overflow in a SMB authentication module. (b) CVE-2004-2115: A cross-site script injection attack in the password field. The vulnerability in ProFTPD is CVE-2005-4816: this is triggered using a long password that causes a buffer overflow. Thus, over 99% of the reported attacks do not involve the password parameter; thus their detection would not be affected. Further, CVE-1999-1237 and CVE-2004-2115 would be detected if a length-preserving obfuscation function is used. CVE-2004-2115 is the only vulnerability that would be missed per the default Cloak configuration.

In the afore-mentioned analysis, we only checked the CVE entry to see if the password is one of the reported exploit vector for that vulnerability. However, it is possible that though the password is not reported as an exploit vector, it may still be an exploit vector. To address this concern, we also studied the description of 16 remotely-exploitable overflow vulnerabilities discovered in Apache from 1999 onwards in detail, and ascertained that our perturbation would not affect the fidelity of honeypot-based detection for any of these attacks. This analysis is presented below.

We obtained the buffer overflow vulnerabilities in Apache reported in the CVE by a case-insensitive search for the keywords "apache", "buffer", and "overflow", and then we studied the remotely exploitable vulnerabilities. Note that for each vulnerability, we inspect the source code to see if an attack could use the password field to exploit that vulnerability (even if the listed exploit conditions do not involve those field). In general, the inspection we could do is limited given the size of the Apache source code along with its various modules; we have attempted to use our knowledge of the semantics of HTTP handling to make this judgment. This analysis demonstrates that each of these attacks will be detected by one of the three mechanisms: (a) the complete obfuscation policy (b) the length-preserving obfuscation policy (c) the in-line appliance for filtering lower layer attacks (the mod_ssl vulnerabilities).

Most of these attacks exploit vulnerabilities in the rich set of Apache modules, and are confined to various parameters in the HTTP request particularly the URL and various options such as the "REALM" argument in the authentication option; our obfuscation does not affect them. We also found 4 vulnerabilities reported in *mod_ssl* itself; three of these occur during the handshake protocol (2 are due to long client certificates and 1 is due to a long certificate revocation list), and one is during the data transfer itself. We can currently

detect only the attack during data transfer, since we have only instrumented the part of OpenSSL code that is invoked during the data transfer. However, we note that all these 3 *mod_ssl* vulnerabilities can be detected by the in-line appliance for screening lower layer attacks, since the certificates are sent in the clear. We also did a similar analysis for the 8 overflow attacks in ProFTPD reported to the CVE, and found that the length-preserving obfuscation would not miss any of them (one of them is an attack via the password field; the complete obfuscation policy would miss that attack).

4.6.3 Fidelity Evaluation: Qualitative Analysis For Unknown Attacks

We now analyze the fidelity of Cloak in more qualitative terms in order to gauge how the detection of unknown attacks would be detected by our obfuscation. There are two issues due to the perturbation performed by Cloak. First, it may introduce false positives; the honeypot detects attacks on the obfuscated stream when it does not detect on the original stream. False positives are not a significant issue in practice since our perturbation is minimal; we re-write only the password parameter in the protocol. In our tests, we did not find any such issues. In fact, if it does induce such a false positive, it is helpful since it helps uncover a bug in the program. Second, it may introduce false negatives which are more worrisome; we mostly deal with this case here.

We now present our qualitative analysis for Apache (the analysis for ProFTPD is much simpler and is similar). Our analysis is structured as follows. First, with some configuration effort, we show that the code path will be likely the *same* in the server and the shadow honeypot. Thus, if the vulnerability code is invoked in the server, it will be invoked in the shadow honeypot as well. Second, we examine the exploit condition on the input that leads to the vulnerability being triggered. If this condition does not involve the obfuscated

parameter, then since the code path is the same, the exploit would still be detected. If it does involve the parameter, and the obfuscation does not affect the condition, then the attack would be detected.

Same Code Path: We divide the set of HTTP requests by the remote client into two kinds: (a) those that contain the password (b) those that do not. We will assume that the shadow honeypot replicates the application and OS configuration of the server carefully. This is far from trivial, but this is an orthogonal issue that we do not aim to address in this work; existing work on honeypots aim to address this issue.

Requests involving the password: The server application sees the original HTTP request while the honeypot sees the HTTP request with a obfuscated password. We make two arguments here.

First, it is possible that the shadow honeypot follows a code path different from the server because of differences in application-level configuration. To avoid such deviations, it is necessary to carefully replicate the server configuration file and server version at the honeypot. Particularly, in the case of Apache which supports a wide variety of modules and configuration parameters, this is essential. Second, it is possible that a new code path is invoked in the honeypot because of the modification in the password itself. For instance, it is very likely that the code path is different depending on whether the username and password correspond to a valid or invalid login. We deal with this issue by replicating the ".htaccess" file which records the list of authorized users and the salted version of their obfuscated password at the honeypot. With such replication, the honeypot would treat the user as successfully logged in, if and only if, the username matches with the obfuscated password. Note that this file is updated whenever the list of authorized users or passwords

changes. If an authentication method such as a SQL backend is used, the username and password database is replicated similarly.

We note there are cases where more configuration is required. If the web service checks for certain password characteristics (*e.g.,* presence of three punctuation marks), for instance, when the user creates the password, then the server admin specifies some additional configuration for that URL to ensure that the code path is the same at the server and the honeypot. For such URLs, the configuration specifies: (a) a series of validity test functions that check the password from the remote client to see if which of these validity tests it passes. (b) a file containing a list of passwords of various lengths for every possible combination of validity test results. With this configuration, Cloak invokes the validity functions on the password typed in by the remote client, and then obfuscates it using a password from the configuration that gives the same result as the password typed in by the client. Though this step requires additional configuration, we believe it is feasible in practice and only requires one-time effort. Further, this is only required for the "create new user" and "change password" pages; during authentication requests, only the password validity is checked. One can use static analysis techniques to aid such configuration.

Requests not involving the password: Typically, once the remote client has been authenticated, subsequent requests need not include the username or the password field. The server maintains state recording the status that the remote client has been logged in/not, and this would verify access for subsequent responses. In this case, Cloak would not modify any of the subsequent responses from the client, and thus, the code path would not be modified. Sometimes, the server might even retain state across multiple connections; perhaps by updating some statistics about which client logged in. It is important that the shadow honeypot retain this state as well. When the same client logs in again later, the server

might attempt to retrieve this state or update it; the vulnerability might involve that piece of code. One vulnerability of this kind is a double-free vulnerability that is triggered when the `free()` call is involved on the same memory location. We deal with this issue by only using deterministic obfuscation schemes; thus, multiple accesses by the same client would be recognized as such by the shadow honeypot, so such attacks would be detected as well.

Thus, the code path would be the same at the server and the shadow honeypot, provided care is taken with replicating the configuration file, list of authorized usernames and obfuscated passwords, and careful configuration for specific URLs. This analysis highlights the difficulty of ensuring fidelity even due to minimal modifications of the payload; it is for this reason we avoid obfuscating other payload fields like username or FTP files. Our original implementation obfuscated usernames as well; however, the username name is difficult to hide since it may be revealed indirectly via other means (*e.g.,* , it is easy to infer the username from the name of the home directory typically sent as an argument to the "CD" command). Apart from the fidelity issues, this difficulty is another reason why we focus on passwords.

Exploit Condition: Since we have argued that the code path is likely the same, thus the vulnerable code would be encountered in the shadow honeypot if it is encountered in the server. Thus, the question remains as to whether the vulnerability would still be detected. Clearly, if the exploit condition does not involve the password, the exploit would be detected. If this is not the case, then detection depends on the obfuscation function. We study this question in the context of classes of attacks detected by our tainting based shadow honeypot.

We use Xu *et al.* [76]'s tainting engine that uses taint-based policies for detecting the

following five classes of attacks. Together, these represent over two-thirds of the vulnerabilities reported in the CVE. These classes and the policies that can detect them are as follows (this is reproduced from their paper): (a) Memory Error Exploits: Most memory error exploits (including stack smashing, heap overflows, integer overflows) can be detected by a policy that tainted data (data obtained from the network) may not be de-referenced as a pointer. (b) Directory traversal attacks: In this case, tainting is used to verify the intended use of an access. This policy is coarse-grained; it verifies whether a request for a particular resource (say, a particular filename) originated from the network or within the application itself. (c) Cross-site scripting attacks: These attacks are caught by a taint-based policy that verifies that any code in the web page returned by a server (which is typically enclosed with a `script` tag) is not tainted. (d) Format string attacks: To catch such attacks, the policy checks that the format specifier (*e.g.,* "%n") used in the `printf` family of calls is not tainted. (e) SQL and Command injection attacks: A taint-based policy that disallows tainted data to be interpreted as a command would detect such attacks.

We now consider these attacks and discuss how our obfuscation affects their detection. Regarding memory overflow errors in the password parameters (stack smashing, heap overflow, and other memory attacks), the complete obfuscation scheme would miss them; however, our length preserving obfuscation policy would be able to catch them at the cost of revealing the length of the password. Further, since the password is typically interpreted as a string, integer overflow attacks will also be caught by length-preserving obfuscation. Directory traversal attacks would be unaffected under both kinds of obfuscation policies since the taint-based policy for catching such attacks is coarse grained; they are only used to identify the source of an access.

Cross-site scripting attacks, format string vulnerabilities, SQL injection attacks, and

command injection attacks require finer discussion. The first three of these attacks exploit vulnerabilities in a web application layered over Apache, while the last is triggered within Apache itself. All four will be triggered only only if certain characteristics are present in the exploit parameter. These include presence of "script" tags (for cross-site scripting attacks), presence of format specifiers like "%n" (for format string attacks), and particular characters like ";" (for SQL injection attacks). The tainting engine checks that these special characters when used by the program are not tainted. To detect such attacks, it is necessary that the obfuscation policy does not remove the format specifiers such as "%n" in the exploit parameter. While it is feasible to design such policies, we have elected not to do so since password characteristics would then be leaked to the security appliance. For instance, a legitimate password might contain the "%" character, and this would be revealed to the appliance. Given that we have already chosen to trade-off some security for password secrecy, we currently make this choice. If desired, this option can easily be added to Cloak by adding such suitable obfuscation functions for detecting these attacks; the idea is that the obfuscation should preserve the set of characteristics used by the taint detection policy (such as "%n", "script" tags).

Though our analysis is in the context of our taint-based honeypot, it is applicable to any appliance that detects these class of attacks; our arguments are made only on the basis on perturbation to the input to these appliances. These arguments can also be extended to other classes of vulnerabilities (*e.g.,* input validation errors, remote-file injection attacks, access control errors, cross-site forgery attacks).

4.6.4 Attacks against Cloak

We now present a security analysis of Cloak. The first attack considers an adversary who tries to leverage her knowledge of Cloak to compromise the server while evading detection by the security appliance, while the other attacks consider adversaries who aim to exploit vulnerabilities in Cloak itself.

An adversary familiar with the obfuscation performed by Cloak may try to take advantage of it by hiding her attack in the password parameter, the only parameter perturbed by Cloak. However, she is then limited to using a vulnerability that is exploitable via this field; per our fidelity analysis, this is a small fraction of known attacks (less than 1% for Apache and none for ProFTPD with the length-preserving policy). As we have argued before, this seems to be the necessary price for maintaining the secrecy of the password.

We next discuss attacks that aim to exploit vulnerabilities in the implementation of Cloak itself. Two kinds of attacks known in literature are applicable: an evasion attack and a state exhaustion attack. We describe how our design and implementation defends against these attacks.

A evasion-style attack against Cloak is as follows. Cloak buffers the incoming stream, and sends a copy to the honeypot. The issue here is that the content checked by the honeypot may not be what is sent to the application. For instance, the obfuscator may skip over abnormal traffic it does not understand (because it is difficult to account for all protocol constructs, per the observation of Pang *et al.* [46]). The adversary can evade detection by using vulnerabilities in such skipped portions of the traffic. We defend against this attack by relying on the fact that passwords are present in only very specific locations in specific ADUs. We have a fair degree of confidence in our confidentiality analysis that we

have not missed any locations where the password may be encoded. Our obfuscator does not skip over any traffic; it only aims to recognize the password in the ADUs and obfuscate the password suitably. If a particular ADU, per configuration or protocol knowledge, is deemed to not contain any passwords, it is passed on un-modified.

State starvation attacks aim to exhaust the state in the DC layer used to buffer packets. We resort to the standard fail-safe option here. Once we run out of buffer, we drop further incoming packets and rely on the remote client to re-connect. We believe this option is acceptable since it is only invoked under high load.

Finally, we argue that the Cloak design lends itself to a simple implementation. The obfuscator is the most complex part of Cloak since it performs deep packet inspection. We note that common protocols like HTTP, FTP, XMPP, etc, have already been standardized; these are all ASCII-based and simple to parse. Our HTTP specific code, for instance, is less than 300 lines long. Further, the protocol-specific code does not maintain state regarding ADUs seen before; it operates on only one ADU at a time. In all, Cloak is less than 1000 semicolons of code.

4.7 Performance Evaluation

We characterize the performance impact of Cloak through two experiments using Apache `2.0.49` as an example (we do not present the results for ProFTPD `1.2.9` since they reveal similar trends). We note that this performance evaluation is only meant to be indicative; our current implementation has not been optimized for performance. This evaluation measures the impact of Cloak on legitimate clients (we also sanity checked our code with sample

exploits). We used the `httperf` HTTP server performance benchmarking tool [40] in order to evaluate our system.

The Apache server runs on a dual CPU 3 GHZ machine with 1 GB RAM. Cloak runs at the server. The honeypot we used runs an instrumented version of Apache that performs taint analysis; the instrumentation was done using the TaintPolicy tool [76]. This was run within a VM (VMWare Workstation 5.0.0) on a dual CPU 3 Ghz machine. `httperf` was run from a client on the same 100 Mbps LAN as the server and the honeypot. The latency between the client and the server was 0.95 ms and the latency between the server and the honeypot was 0.28 ms. The first experiment micro-benchmarks the Cloak code and measures the bandwidth and latency impact on a single client request. The second experiment presents the throughput impact of Cloak.

Micro-benchmarks: We had `httperf` issue 100 GET requests to retrieve the password-protected page from the server. This password is sent using the BASIC authentication method and the GET request was 267 bytes long. We measured the end-to-end latency seen by the client (the time taken to retrieve the entire page) as well as the overhead added by Cloak. Without Cloak running on the server, the end-to-end latency as measured from the client was 53.3 ms (on average). With Cloak, the latency increased to 54.2 ms. The increase in latency due to the processing by Cloak was only 183 microseconds; the rest of the increase is due to the network latency and the processing at the honeypot itself. We note that the honeypot we used has only 6% overhead due to instrumentation, so the network latency is the pre-dominant factor. This latency impact may be acceptable in most scenarios, and if necessary, the honeypot can be deployed closer to the server so as to reduce the latency further. We note that the processing latency of Cloak is nearly the same in other authentication options as well (HTTP Digest Authentication and POST request-

based authentication) since they differ only in the exact nature of re-writing performed by the obfuscator.

The bandwidth overhead is the cost of transmitting the GET request to the honeypot (267 bytes in our test). Since the average HTTP request is typically much less than a kilobyte (Mogul *et al.* [37] measured the mean HTTP request size to be 311 bytes in their trace), this bandwidth overhead per request appears to be reasonable. Note that since we only send incoming packets to the appliance, the bandwidth overhead is minimal since most web services offer data for download and the response size is typically much greater than the request size.

Macro-benchmarks: We measured the throughput as follows: we used `httperf` to generate 1000 connections at the rate of 100 per second to saturate the system, and then measured the time taken to process all 1000 requests. This time was measured to be 53 seconds without Cloak and 55 seconds with Cloak. Once again, this is heavily dependent on the latency between the host and the honeypot since Cloak adds only processing latency. This throughput impact, if too high in certain scenarios, can be reduced by situating the honeypot appliance closer.

4.8 Limitations

We now detail some limitations of our work, and discuss how they might be addressed.

First, though Cloak is meant to work with third party appliances, since we do not have access to one, we have not tested it with such an appliance. Currently, we require that the appliance notify Cloak upon detecting an exploit; this may not be supported by appliances designed to operate as in-line elements. In such cases, we envision that the appliance can be

placed on a private network and a fake network layer stream be fed to it from Cloak. Any packets that make it past the appliance are deemed safe and are sent to the application.

Second, in some enterprises, traffic to servers is sent via application-level gateways (*e.g.,* server-side HTTP proxies). In such cases, it may be desirable for the gateway to perform de-sensitization and send the traffic to third-party appliances. If support for this scenario is desired, our implementation of Cloak can be run at the *gateway* instead of the *server*; thus, Cloak would capture the traffic at the gateway and send it to the security appliance. Note however that in this model, unlike the server deployment model, sensitive data is exposed to the gateway (and its admin).

Third, the Cloak implementation currently only applies to Linux servers; it can be extended to Windows as well by interposing Cloak over the DLL libraries for providing end-to-end encryption.

4.9 Related Work

The issue of confidentiality in the context of remote security appliances has seen some active work in the IETF. However, Cloak is the first system that aims to protect *passwords* in the context of *online* intrusion prevention appliances without requiring *any protocol re-design or application modifications.* We classify related work into two main classes; the first do not require any application re-writing, the second require that the application be modified. Before this discussion, we first mention the remote security appliances in use today.

Remote Security Appliances: We refer to any IPS/IDS that analyzes application payload at a computer that is not the source/destination of the payload as a remote IPS/IDS

system. This includes firewalls, NIPS, NIDS like Bro [51], distributed intrusion detection systems (*e.g.,* DIDS [60])), and more recent honeypot-like systems (*e.g.,* Shadow Honeypots [7], SpyProxy [41]). These proposals do not account for sensitive data; Cloak is designed to take such requirements into account. The Shadow honeypots proposal has a tightly-coupled deployment option where the instrumented application runs at the server; this does not lead to any data leakage, but such coupling is not possible for expensive analysis (*e.g.,* binary tainting) and for third-party appliances.

Handling the confidentiality issue without modifying the application: Existing efforts (*e.g.,* Parekh *et al.* [47], Zhang *et al.* [79], Lincoln *et al.* [33]) in this direction primarily focusses on privacy guarantees. Parekh *et al.* aims to protect the identity of the user in the context of anomaly detection schemes that require payload sharing across multiple sites. Zhang *et al.* deal with the same issue in the context of sharing of logs across multiple sites for offline analysis. Lincoln *et al.* address the privacy issues that arise in the context of correlating alerts from multiple sites. Cloak is in the same spirit as this existing work, however Cloak focuses on protecting sensitive information such as passwords in the context of online intrusion prevention. Privacy guarantees are much more difficult to provide since identifying data of various kinds is embedded in network traffic; we instead only aim to provide de-sensitization. This lets us use much simpler obfuscation whose impact on fidelity is easier to analyze.

Handling the confidentiality issue by modifying the application: Another way to help remote security appliances perform analysis on encrypted traffic is to re-write the application to expose data to the appliance. Schaffrath [24] concludes that the most feasible solution given the confidentiality constraints is for protocols to be re-designed so as to

protect only the most sensitive information using encryption and reveal the rest to on-path middle-boxes.

4.10 Conclusion

We have described the design and implementation of Cloak, a system that scrubs passwords from encrypted network traffic before it is vetted by a remote security appliance for attacks. We have tested our system with Apache and ProFTPD, widely used HTTP and FTP servers, along with an appliance that detects unknown attacks using taint analysis. We have shown that we likely miss very few attack in practice (less than 1% for Apache and ProFTPD) and that the overhead is minimal (only about 183 micro-seconds in our prototype Apache and ProFTPD implementation). We have also described how our implementation can leverage existing frameworks to de-sensitize additional data, if required.

Chapter 5

Conclusion and Future Work

Several challenges remain to be met in addressing requirements of network security mechanisms in the future. It is becoming increasingly clear that complex protocol analysis techniques for gaining context will be required for accurate implementation of even simple policy requirements like access control.

In this chapter, we summarize the three main contributions in the thesis, and then discuss some future work.

5.1 Tools for Session Structure Inference:

To reduce the overhead of reverse-engineering of inter-connection context, this thesis discusses semi-automatic mechanisms that can mine a trace of network connection logs to infer *session* structure.

Context at the session level may be necessary in order to infer the intent of a network packet. The knowledge of session descriptors gained by our work can be useful for a de-

veloper in understanding such context. Our work only captures the qualitative nature of such sessions; however, with such session descriptors in hand, one can then set about obtaining distributions for the duration of a session, number of connections in a session, and so forth (using techniques similar to those used in Nuzman *et al.* [44] for HTTP sessions). The main value of our approach is that it requires much less *a priori* information about the applications in the trace (it requires only a mapping from port number to application, whereas previous work explicitly required a session descriptor for the application as well).

5.2 On the Use of Global Context for Cooperative Worm Containment

One of the fundamental issues in obtaining global context is the trade-off between the cost of obtaining overhead against the scale and speed of the security event it is useful for detection. We study this trade-off in the context of worm detection. Exchange of information across multiple entities has been suggested as a solution to deal with fast worm attacks; we use analytical modeling to study the trade-off in this regard.

The implications of this study are two-fold. First, our numerical study in an Internet-like scenario quantify how cooperation would perform against worms of varying virulence. Second, our results also illustrate how the performance of the various building blocks of cooperation, such as local detection, filtering, and signaling, influence containment. This can help benchmark various techniques that implement these building blocks. Although our study deals with Internet-level cooperation among firewalls to contain scanning worms, the advantage of our analytical approach is that it can be applied in a variety of scenarios:

to enterprise-level or Internet-level cooperation, to host-level or firewall-level cooperation, and to a limited extent for analyzing sophisticated hit-list based worms.

5.3 Off-path Architecture for Intrusion Prevention

In order to deal with the confidentiality issues that arise as a result of exposing too much context to the security appliance (and to the administrator of that appliance), we propose a novel off-path architecture in Cloak. Cloak relies on host software to furnish a de-sensitized version of network traffic to a in-network appliance.

Unlike current solutions, Cloak does not require any applications to be re-written and does not require that both ends of the communication deploy the modified application. Further, in Cloak, the traffic continues to be encrypted end-to-end, thus resisting eavesdropping by on-path adversaries.

5.4 Future Work

A particularly fruitful area for near-term future work is extending our methods to extract application sessions that involve multiple remote hosts (per the discussion in Section 2.4.3). Our work in [28] gives some preliminary results in this regard, finding that this extension better captures the behavior of peer-to-peer applications, proxies (mail and Web), and a number of other applications. For example, proxies typically demonstrate the characteristic session structure of an incoming connection from one remote host followed by an outgoing connection on the same port to another remote host.

More broadly, we hope that our session extraction and structure abstraction tools com-

prise a useful addition to the toolbox of administrators and researchers. Our session extraction approach can be thought of as inferring "hidden causality" in network connections; it identifies causally related connections by exploiting the observation that they typically occur closer to each other in practice compared to causally unrelated connections. This observation is the basis of our statistical test, which is by no means a bullet-proof causality test, in that it has non-negligible false negative ratio (and in the adversarial input case, an adversary simply needs to ensure that this connections are far apart enough in time to evade our test). However, its false positive ratio is bounded (as shown in Section 2.4.3), and thus most of the inferred sessions are indeed casually related connections. Furthermore, in cases such as worm propagation, where (if we extend the model to include multiple remote hosts) we can relate outgoing infection attempts to the initial infection connection, the scale of the number of connections originated by the host and the speed of the post-compromise activity may prove readily and quickly detectable by our session extraction mechanism.

We plan to extend Cloak in the future in two directions. First, Cloak can be extended to support client applications. Clients, particularly web browsers, are increasingly subject to attacks from malicious/compromised servers. For this reason, security appliances vet incoming traffic to clients as well. However, the user may like to protect sensitive information in the webpage sent by the server (say, her bank account number) from the appliance. Cloak can rely on existing HTML re-writing tools to perform the necessary obfuscation. Second, Cloak can also be adapted to support other kinds of security appliances that perform, for instance, anomaly detection; anomaly detection schemes usually require such remote analysis since they benefit from aggregating traffic from several end-hosts.

Bibliography

[1] Counterpane Internet Security. `www.counterpane.com`.

[2] CVE: Common Vulnerabilities and Exposures. `http://cve.mitre.org/`.

[3] Fireeye, Inc. `http://www.fireeye.com`.

[4] HIPAA Privacy and Security Compliance Assistance. `http://www.calhipaa.com/`.

[5] Sarbanes-Oaxley Act. `http://www.soxlaw.com/`.

[6] Anagnostakis, K. G., Greenwald, M. B., Ioannidis, S., Keromytis, A. D., and Li, D. A Cooperative Immunization System for an Untrusting Internet. In *Proc. ICON* (Oct 2003).

[7] Anagnostakis, K. G., Sidiroglou, S., Akritidis, P., Xinidis, K., Markatos, E., and Keromytis, A. D. Detecting Targeted Attacks using Shadow Honeypots. In *Proc. Usenix Security* (2005).

[8] Berk, V., Bakos, G., and Morris, R. Designing a Framework for Active Worm Detection on Global Networks. In *IEEE Intl Workshop on Information Assurance* (Mar 2003).

[9] Blum, A., Song, D., and Venkataraman, S. Detection of Interactive Stepping Stones: Algorithms and Confidence Bounds. In *7th International Symposium on Recent Advances in Intrusion Detection (RAID 2004)* (Sep 2004), pp. 20–29.

[10] Bolot, J.-C. End-to-End Packet Delay and Loss Behavior in the Internet. In *Proc. SIGCOMM* (Sept 1993).

[11] Bruce Scheneir. Why Outsource? `http://bt.counterpane.com/why-outsource.pdf`.

[12] Claffy, K., Braun, H.-W., and Polyzos, G. A Parameterizable Methodology for Internet Traffic Flow Profiling. *IEEE JSAC 13*, 8 (1995), 1481–1494.

[13] Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Shannon, C., and Brown, J. Can we contain Internet worms? In *Proc. HOTNETS* (2004).

[14] Crandall, J. R., and Chong, F. T. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proc. International Symposium on Microarchitecture* (Oregon, Dec 2004).

[15] Crovella, M., and Bestavros, A. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *Proc. SIGMETRICS* (May 1996).

[16] Cui, W., Paxson, V., Weaver, N., and Katz, R. H. Protocol-Independent Adaptive Replay of Application Dialog. In *Proc. NDSS* (San Deigo, CA, Feb 2006).

[17] Cyber-Ark Survey. Press Release. `http://www.cyber-ark.com/news-events/pr_20080827.asp`.

[18] Danzig, P., Jamin, S., Caceres, R., Mitzel, D., and Estrin, D. An Empirical Workload Model for Driving Wide-area TCP/IP Network Simulations. *Internetworking: Research and Experience 3*, 1 (1992), 1–26.

[19] Eric Rescorla. OpenSSL examples. `http://www.rtfm.com/openssl-examples/`.

[20] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext Transfer Protocol: HTTP/1.1. Internet RFC 2616, 1999.

[21] Fowler, H., and Leland, W. Local Area Network Traffic Characteristics, with Implications for Broadband Network Congestion Management. *IEEE JSAC 9*, 7 (1991), 1139–1149.

[22] Finite State Automata Utilities (version 6.2). `http://odur.let.rug.nl/~vannoord/Fsa/`.

[23] Graphviz - Graph Visualization Software. `http://www.graphviz.org/`.

[24] Gregor Schaffrath. Network Intrusion Detection Systems and Encryption: Friends or Foes? University of Zurich.

[25] Heimlich, S. Traffic Characterization of the NSFNET National Backbone. In *Proc. Winter USENIX Conference* (1990).

[26] Jain, R., and Routhier, S. Packet Trains — Measurements and a New Model for Computer Network Traffic. *IEEE JSAC 4*, 6 (1986), 986–995.

[27] Jung, J., Paxson, V., Berger, A. W., and Balakrishnan, H. Fast Portscan Detection Using Sequential Hypothesis Testing. In *Proc. IEEE Security and Privacy* (2004).

[28] Kannan, J., Jung, J., Paxson, V., and Koksal, C. E. Detecting Hidden Causality in Network Connections. Tech. rep., University of California, Berkeley, 2005.

[29] Kim, H. A., and Karp, B. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proc. Usenix Security* (2004).

[30] Kreibich, C., and Crowcroft, J. Honeycomb: Creating Intrusion Detection Signatures using Honeypots. *CCR* (2004).

[31] Kumar, A., Paxson, V., and Weaver, N. Exploiting Underlying Structure for Detailed Reconstruction of an Internet-scale Event. In *Proc. ACM IMC* (Oct 2005).

[32] Leland, W., Taqqu, M., Willinger, W., and Wilson", D. On the Self-Similar Nature of Ethernet Traffic. *IEEE/ACM Transactions on Networking 2*, 1 (1994).

[33] Lincoln, P., Porras, P., and Shmatikov, V. Privacy-Preserving Sharing and Correction of Security Alerts. In *Proc. USENIX Security* (2004).

[34] Ma, J., Levchenko, K., Kreibich, C., Savage, S., and Voelker, G. Automatic Protocol Inference: Unexpected Means of Protocol Inference. In *Proc. ACM IMC* (Oct 2006).

[35] Menascé, D. A., Almeida, V. A. F., Fonseca, R., and Mendes, M. A. A Methodology for Workload Characterization of E-commerce Sites. In *Proc. ACM conference on Electronic commerce* (1999), pp. 119–128.

[36] Mogul, J. End-to-End Internet Packet Dynamics. In *Proc. SIGCOMM* (Sept 1997).

[37] Mogul, J., Douglis, F., Feldmann, A., and Krishnamurthy, B. Potential benefits of delta encoding and data compression for HTTP. In *Proc. SIGCOMM* (1997).

[38] Moore, A. W., and Zuev, D. Internet traffic classification using Bayesian analysis techniques. In *SIGMETRICS '05* (2005), pp. 50–60.

[39] Moore, D., Shannon, C., Voelker, G. M., and Savage, S. Internet quarantine: Requirements for containing self-propagating code. In *Proc. INFOCOM* (2003).

[40] Mosberger, D., and Jin, T. httperf: A Tool for Measuring Web Server Performance. *ACM SIGMETRICS Performance Evaluation Review 26*, 3 (1998), 31–37.

[41] Moshchuk, A., Bragin, T., Deville, D., Gribble, S. D., and Levy, H. M. SpyProxy: Execution-based Detection of Malicious Web Content. In *Proc. USENIX Security* (2007).

[42] Newsome, J., and Song, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. NDSS* (2005).

[43] Nojiri, D., Rowe, J., and Levitt, K. Cooperative Response Strategies for Large Scale Attack Mitigation. In *Proc. DISCEX* (2003).

[44] Nuzman, C., Saniee, I., Sweldens, W., and Weiss, A. A compound model for TCP connection arrivals for LAN and WAN applications. *Computer Networks 40*, 3 (2002), 319–337.

[45] Padhye, J., Firoiu, V., Towsley, D., and Krusoe, J. Modeling TCP throughput: A simple model and its empirical validation. In *Proc. ACM SIGCOMM* (1998).

[46] Pang, R., and Paxson, V. A High-level Programming Environment for Packet Trace Anonymization and Transformation. In *Proc. SIGCOMM* (2003).

[47] Parekh, J. J., Wang, K., and Stolfo, S. J. Privacy-Preserving Payload-based Correlation for Accurate Malicious Traffic Detection. In *Proc. LSAD* (2006).

[48] Paxson, V. Empirically-derived analytic models of wide-area tcp connections. *IEEE/ACM Transactions on Networking 2*, 4 (1994).

[49] Paxson, V. Automated packet trace analysis of TCP implementations. In *Proc. SIGCOMM* (New York, NY, USA, 1997), ACM Press, pp. 167–179.

[50] Paxson, V. End-to-End Internet Packet Dynamics. In *Proc. SIGCOMM* (Sept 1997).

[51] Paxson, V. Bro: a system for detecting network intruders in real-time. *Computer Networks (Amsterdam, Netherlands: 1999) 31*, 23–24 (1999).

[52] Paxson, V., and Floyd, S. Wide area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking 3*, 3 (1995), 226–244.

[53] TCP Ports List, UDP Ports List. `http://seifried.org/security/ports/`.

[54] The Protocol Informatics Project. `http://www.baselineresearch.net/PI/`.

[55] Provos, N. A Virtual Honeypot Framework. In *Proc. USENIX Security Symposium* (Aug 2004).

[56] Ross, S. M. *Introduction to Probability Models, 8th Edition*. Academic Press, 2003.

[57] Schechter, S. E., Jung, J., and Berger, A. W. Very Fast Containment of Scanning Worms. In *Proc. RAID* (Sep 2004).

[58] Singh, S., Estan, C., Varghese, G., and Savage, S. Automatic Worm Fingerprinting. In *Proc. OSDI* (2004).

[59] Smith, F. D., Hernandez-Campos, F., Jeffay, K., and Ott, D. What TCP/IP protocol headers can tell us about the web. In *SIGMETRICS/Performance* (2001), pp. 245–256.

[60] Snapp, S. R., Brentano, J., Dias, G. V., Goan, T. L., Heberlein, L. T., lin Ho, C., Levitt, K. N., Mukherjee, B., Smaha, S. E., Grance, T., Teal, D. M., and Mansur, D. DIDS (Distributed Intrusion Detection System) - Motivation, Architecture, and An Early Prototype. In *NIST-NCSC*, pp. 167–176.

[61] Staniford, S. Containment of Scanning Worms in Enterprise Networks. *Journal of Computer Security* (2005 (to appear)).

[62] Staniford, S., Cheung, S., Crawford, R., Dilger, M., Frank, J., Hoagland, J., Levitt, K., Wee, C., Yip, R., and Zerkle, D. GrIDS – A Graph-based Intrusion Detection System for Large Networks. In *Proc. National Information Systems Security Conference* (1996).

[63] Staniford-Chen, S., and Heberlein, L. T. Holding intruders accountable on the internet. In *Proc. IEEE Symposium on Security and Privacy* (Washington, 1995).

[64] Timbuktu Pro Remote Control Software. `http://www.netopia.com/software/products/tb2/`.

[65] Twycross, J., and Williamson, M. M. Implementing and testing a virus throttle. In *Proc. USENIX Security* (2003).

[66] Wang, H. J., Guo, C., Simon, D. R., and Zugenmaier, A. Shield: Vulnerability-driven Network Filters for Preventing Known Vulnerability Exploits. In *Proc. SIGCOMM* (2004).

[67] Wang, X., Reeves, D. S., and Wu, S. F. Inter-Packet Delay Based Correlation for Tracing Encrypted Connections through Stepping Stones. In *Proc. ESORICS* (London, UK, 2002), Springer-Verlag, pp. 244–263.

[68] Weaver, N., Ellis, D., Staniford, S., and Paxson, V. Worms vs Perimeters: The Case for HardLANs. In *Proc. Hot Interconnects* (2004).

[69] Weaver, N., Hamadeh, I., Kesidis, G., and Paxson, V. Preliminary results using scaledown to explore worm dynamics. In *Proc. ACM CCS WORM* (Oct 2004).

[70] Weaver, N., Paxson, V., and Staniford, S. Wormholes and a Honeyfarm: Automatically Detecting Novel Worms. In *DIMACS Large Scale Attacks Workshop* (2003).

[71] Weaver, N., Staniford, S., and Paxson, V. Very Fast Containment of Scanning Worms. In *Proc. USENIX Security* (2004).

[72] Willinger, W., Taqqu, M., Sherman, R., and Wilson, D. Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level. In *Proc. SIGCOMM* (Sept 1995).

[73] Wong, C., Wang, C., Song, D., Bielski, S., and Ganger, G. R. Dynamic Quarantine of Internet Worms. In *Proc. DSN* (2004).

[74] Wu, J., Vangala, S., Gao, L., , and Kwiat, K. An Effective Architecture and Algorithm for Detecting Worms with Various Scan Techniques. In *Proc. NDSS* (Feb 2004).

[75] Xie, Y., Sekar, V., Maltz, D., Reiter, M., and Zhang, H. Worm Origin Identification Using Random Moonwalks. In *Proc. IEEE Security and Privacy* (Oakland, CA, May 2005).

[76] Xu, W., Bhatkar, S., and Sekar, R. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proc. USENIX Security* (2006).

[77] Yegneswaran, V., Barford, P., and Jha, S. Global Intrusion Detection in the Domino Overlay System. In *Proc. NDSS* (2004).

[78] Yoda, K., and Etoh, H. Finding a Connection Chain for Tracing Intruders. In *Proc. ESORICS* (London, UK, 2000), Springer-Verlag, pp. 191–205.

[79] Zhang, J., Borisov, N., and Yurcik, W. Outsourcing Security Analysis with Anonymized Logs. In *Proc. SECOVAL* (2006).

[80] Zhang, Y., Breslau, L., Paxson, V., and Shenker, S. On the characteristics and origins of internet flow rates. In *Proc. ACM SIGCOMM* (2002).

[81] Zhang, Y., Duffield, N., Paxson, V., and Shenker, S. On the constancy of internet path properties. In *Proc. ACM SIGCOMM Internet Measurement Workshop* (2001).

[82] Zhang, Y., and Paxson, V. Detecting Stepping Stones. In *Proc. 9th USENIX Security Symposium* (Denver, CO, Aug 2000).