YAMA: A System For Marking Network Traffic

by

Néstor Felipe Hernández González

B.S. Computer Science, M.I.T., 2005

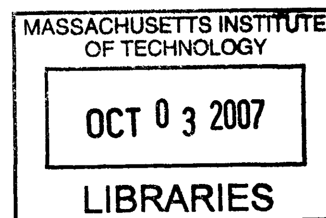
Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

September 2006

© 2006 Massachusetts Institute of Technology
All rights reserved.



BARKER

Author . . . .
Department of Electrical Engineering and Computer Science
August 28, 2006

Certified by . . . .
Robert K. Cunningham
Associate Group Leader, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Students

This work is sponsored by the United States Air Force under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

YAMA: A System For Marking Network Traffic

by

Néstor Felipe Hernández González

Submitted to the
Department of Electrical Engineering and Computer Science
August 28, 2006
In Partial Fulfillment of the
Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Computer security performance analysis requires precise labeling of traffic as either background or attack traffic. When an experiment is performed on-line, it may also be important to identify traffic from the security system. Today this is tedious and difficult, requiring personnel with a deep understanding of multiple protocols.

YAMA (Your Able Marking Aide) is a tool that labels sessions and packets associated with a set of user actions given those actions, the traffic, and a network configuration (host information and web page corpus). An evaluation of a version that processes web traffic is performed using data from Alexas Top 100 Sites. YAMA 1.0 correctly associates the action of visiting a specific site with 90% of all HTTP packets, and 99% of both HTTP GET and DNS packets. Furthermore, YAMA 1.0 produces zero false positives when given a high-level event indicating a user visited one web site and packets from a different site.

Thesis Supervisor: Robert K. Cunningham
Title: Associate Group Leader, MIT Lincoln Laboratory

This work is sponsored by the United States Air Force under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

Acknowledgments

There are a great many thanks to hand out here. First I'd like to thank my advisor, Rob Cunningham, for his time, insight, and support over the past year, and Bill Streilein for sitting in on our meetings and providing some outside perspective. I'd also like to thank Lee Rossey and the rest of the LARIAT team for their continued endorsement over the years, and the great deal of work they have put in to create a fine testbed.

Outside of the Lab I'd like to thank my brothers at the Nu Delta Fraternity for their general support, fun times, and distracting emails that have bettered my spirits throughout. A special thanks goes out to those who lent their editing skills towards the end.

Finally, and most importantly, I want to thank my family for all their boundless love and support over the years. So many thanks to Dad, Mom, my beautiful sister Vivian, and my very cool brother David. Los quiero mucho.

Contents

1 Introduction **15**

1.1 The Problem 15

1.2 Related Work 16

1.3 Goals and Assumptions 18

1.3.1 Goals 18

1.3.2 Scalability and Real-time Execution 19

1.3.3 Assumptions 19

1.4 Thesis Outline 21

2 A Model For Event-Driven Network Traffic **23**

2.1 Agents 25

2.2 Events 26

2.3 Data 27

2.4 Relationships Between Agents, Events, and Data 29

2.5 Examples 29

2.5.1 A Simple Web Transaction 30

2.5.2 A Complex Web Transaction 31

3 Adversary Model **37**
3.1 Intentions in our Event Traffic Model 38
3.2 The Active Adversary 40
3.2.1 Generation of "User" Events 40
3.2.2 Session Initiation 41
3.2.3 Packet Injection 42
3.3 The Passive Adversary 43
3.3.1 Session Redirection 44
3.3.2 File Tainting 45
3.3.3 Packet Alteration 45
3.4 Consequences for Traffic Marking 46
3.4.1 Dealing With The Active Adversary 46
3.4.2 Dealing With The Passive Adversary 47

4 Correlating User Events and Network Traffic **49**
4.1 Bridging the Causal Gap Between User Events and Observed Traffic . 49
4.1.1 Generating Expectations for Events 50
4.1.2 Inferring Events from Network Traffic 52
4.1.3 Matching Expected and Inferred Events 52
4.2 Correlation in the Presence of Adversaries 53

5 Design **55**
5.1 Overview 55
5.2 The Simulation Approach 58
5.3 Data Structures and Algorithms 58
5.3.1 Modules 58

5.3.2 Events . . . 59
5.3.3 Virtual Buffers . . . 60
5.3.4 Generating Expectations . . . 60
5.3.5 Acceptors . . . 61
5.3.6 Acceptor Lists . . . 62

6 Implementation **63**

6.1 Language and Tools . . . 63
6.2 Architecture . . . 64
6.3 Basic Interfaces . . . 64
6.4 Modules . . . 65
6.5 Data Representation . . . 66
6.5.1 Virtual Buffers: The Basic Building Block . . . 66
6.5.2 Representing Higher Level Data . . . 68
6.6 The Browser Module . . . 69
6.6.1 A Two-Tiered Structure for Expected Traffic . . . 69
6.6.2 Data Processing in the Browser Module . . . 71
6.6.3 The Need for a Web Corpus . . . 73
6.7 The DNS Module . . . 73
6.7.1 Performing a Hostname Lookup Through DNS . . . 73
6.7.2 Generating Expectations for DNS Messages . . . 74
6.7.3 Data Processing and Expectation Matching . . . 74
6.8 Matching Expected to Observed Traffic . . . 74
6.9 Simulating Agents as Modules . . . 75
6.9.1 Service and Network Modules . . . 75
6.9.2 User Application Modules . . . 76

6.9.3 Driving the Simulation 76

7 Evaluation **79**

7.1 Metrics 79

7.2 The Data Sets 81

7.2.1 Sampled Site Characteristics 81

7.2.2 Data Collection 85

7.2.3 Traffic Capture Setup 87

7.2.4 Event Generation and Logging 87

7.3 Building a Web Corpus 88

7.4 General Test Procedures 88

7.5 Test 1: Detection Rate 89

7.5.1 Procedure 89

7.5.2 Results 90

7.6 Test 2: False Positive Rate 95

7.6.1 Procedure 95

7.6.2 Results 96

8 Conclusions and Future Work **97**

8.1 Conclusions 97

8.2 Future Work 99

A Sites in the Alexa "Clean 98" **101**

List of Figures

2-1 A Very Simplified Model for Event-Driven Network Traffic 24
2-2 Example Relationships Between Model Components 29
2-3 Top Level Event Graph for A Simple Web Transaction 30
2-4 Client Side Step in a Simple Web Transaction 32
2-5 Top Level Event Graph for A Complex Web Transaction 33
2-6 Example Component Relationships when Fetching HTML Source . . 34
2-7 Relationships for Fetching HTML Sub-Entities 35

3-1 The Active and Passive Adversaries 38
3-2 Spawning a Malicious Agent 39
3-3 Generation of "User" Events by an Active Adversary 41
3-4 Session Initiation by an Active Adversary 42
3-5 Packet Injection by an Active Adversary 43
3-6 Session Redirection via a Compromised DNS Server 44
3-7 The Packet-Altering Passive Adversry 46

4-1 Given Events and Traffic, How do we bridge the gap? 50
4-2 Bridging the gap by matching expectations and inferences? 51

5-1 A Structure for Correlation: Expectation and Inference 57

5-2 A Module 59
5-3 An Event 60
5-4 A Virtual Buffer 60
5-5 Traffic is Tested Against Acceptors 62

6-1 Modules and Traffic Flow in YAMA 1.0 64
6-2 The Browser Module 70

7-1 Composition of the Alexa "Clean 98" 83
7-2 Data Set Composition by Traffic and Site Type 84
7-3 Histograms Showing Number of Sites with Given Traffic Element Counts 86
7-4 Percent Coverage By Traffic Type and Data Set 90
7-5 Percent Coverage For HTTP Packets by Packet Source Type 92
7-6 Percent of HTTP Packets of Each Type Covered 93
7-7 Percent Coverage For HTTP Request by Packet Source Type 93
7-8 Percent of HTTP Requests of Each Type Covered 94
7-9 Percent Coverage For DNSQueries Broken by Packet Source Type . . 94
7-10 Percent of DNS Queries of Each Type Covered 95

List of Tables

7.1 Packet Classifications 80
7.2 The Data Sets 85
7.3 Web Corpora Used For Each Data Set 88
7.4 False Positives Per Traffic Type 96

Chapter 1

Introduction

1.1 The Problem

Computer network intrusion detection performance analysis requires precise labeling of all traffic as a member of one of two and perhaps three categories, so that the false alarm and detection rates can be correctly measured. In the first category is background traffic and is related to the mission of those using the network. In the second category is attack traffic. In the third category is traffic that cannot be labeled, given available information. When an experiment is performed on-line with an intrusion detection system, it may also be important to identify traffic that originates from that system. Today doing this is tedious and difficult, requiring analysis by personnel with a deep understanding of multiple protocols.

In a now-famous April Fools day RFC, The Security Flag in the IPv4 Header, Bellovin jokingly proposed solving this by requiring attackers set an evil bit [7]. Even if such a proposal were to be implemented in a testbed environment, however, controlled evaluations cannot use such an in-packet flag, as it introduces an unwanted

artifact into the testing procedure. Much better is to use a tool that can mark packets after the fact, but doing this is extremely difficult. As Bellovin noted: Firewalls, packet filters, intrusion detection systems, and the like often have difficulty distinguishing between packets that have malicious intent and those that are merely unusual.

All of those devices have the difficult problem of marking packets given only the contents of the packet and perhaps the configuration of the network. Controlled experiments have a significant advantage: the intent of each user action is known, so a tool can be built to associate user actions and mark produced packets. Doing this is difficult - a single user action produces multiple sessions using different protocols, and the total number of packets produced can easily range into the thousands.

1.2 Related Work

This work pre-supposes the existence of moderate to large-scale security software test infrastructures. There are several in widespread use today.

The Lincoln Adaptable Real Time Information Assurance Testbed (LARIAT) is a network testbed which uses events at the application level (such as sending an email or browsing to a website) to create realistic network traffic [29]. LARIAT is used to run repeatable experiments on everything from network hardware components to intrusion detection systems and other information assurance (IA) technologies. Because traffic is generated by user models interacting with applications, it is difficult to know which network packets were directly or indirectly generated by LARIAT. Your Able Marking Aide (YAMA) marks and differentiates traffic purposefully generated by LARIAT user events from traffic which is generated by other sources.

In 1998 and 1999 DARPA sponsored evaluations of research intrusion detection

systems (IDSs) [20, 21]. Attack scenarios were run against a background of generated network traffic to create a corpus of material which was used to test the effectiveness of the IDSs. Great effort was spent in order to separate attack traffic from the background traffic, creating a ground truth against which the results produced by the IDSs could be compared. LARIAT was created as an extension of the traffic generating testbed used in these evaluations. It can be used to perform real time evaluations of IDSs and their responses, and can be integrated into existing networks or a few thousand hosts, eliminating the need for a separate group of testbed computers.

LARIAT allows researchers to run many different experiments with great flexibility. Though LARIAT greatly simplifies setup and operation of experiments [3], analysis of data is left mostly up to the experimenters, and great effort must still be exerted in order to separate traffic particular to the experiment from the background traffic generated by LARIAT. Decreasing the traffic analysis time for experimenters is the primary motivation for this project.

DETER The DETER testbed is a multi-site facility supporting information assurance experiments running "risky" code [5]. The testbed allows for remote and scriptable administration of experiments, including loading of experimental nodes with particular disk images, and connecting of the nodes into almost any network configuration. Although there is no direct IP access to the testbed nodes, the user has complete control of the experiment and the state of each node through a web interface and SSH access to a "Users" machine connected to the actual testbed. Background traffic is produced by replaying captured packets, so traffic can be easily labeled but would not be particularly realistic when security technology alters or limits communication.

Once realistic traffic is available, it needs to be marked. Detecting malicious activity is central to the computer intrusion detection problem [13]. Even today network intrusion detection remains an active research field.

1.3 Goals and Assumptions

1.3.1 Goals

Here we state the goals for YAMA. Though not all of these goals are meant to be achieved in the first versions, they nevertheless play a huge role in shaping the design.

Identification of User-Generated Network Traffic

The primary goal of YAMA is the correct and complete identification and marking of user-generated network traffic. After running YAMA it should be clear which network packets are with which event.

Detection of Adversary Activity In Network Traffic

YAMA should support the detection of adversary traffic as discussed in Chapter 3. Identifying the user-generated traffic gets us much of the way there. The active adversary traffic is revealed by subtracting the traffic generated by users and automated network processes from the total traffic. Passive adversary activity is a bit trickier, as its effects can reside within the user network traffic. YAMA is to allow for the detection of this activity.

1.3.2 Scalability and Real-time Execution

YAMA should be capable of handling large volumes of data. Its design should allow for extension of the system to run in a distributed computing environment, and keep memory use to a minimum.

In addition, its design should allow for event-by-event and packet-by-packet input; it should not rely on complete packet or event lists. This requirement allows for YAMA to potentially be run in real time as the relevant events and traffic are being generated.

1.3.3 Assumptions

Here we present the assumptions upon which the design is based.

Available Inputs

Our requirements for the available inputs to YAMA stems from its eventual use in user-event-driven network testbeds. The following items detail the information which we assume could be collected in such a setting.

Events A time-stamped set of high-level events which affect our network. In the case of events which carry data or context, the data or context should also be available. For example, the event of a user visiting a website would carry with it the host and browser that the event occurred on, and the URL of the site visited.

Traffic A single traffic capture spanning the time frame of all events of interest. This is a timestamped set of traffic data.

Network Topology A description of the interconnections between all devices on the network at all points in the time spanning the events of interest.

Host and Network Component Configuration The initial state and settings for devices in the network.

Server Data The data, or a representation for the data, stored by network servers. This includes information in DNS or web servers. Depending on the type of service, the system might require only the data initially stored in the server, or the data at every point in time within the relevant time period. This data might be provided by simply giving YAMA access to the relevant servers at run time.

Quality of Input

We assume that the input data will be accurate and complete. The event list should contain every event of interest and any related events. Traffic captures be as complete as possible. Other data should reflect the state of the network at the time the relevant events and traffic occurred.

Each form of input should not only be accurate as a unit, but also when cross-referenced with the other forms of input. For example, time stamps for both user events and network traffic should use the same reference clock. Inaccuracies or missing data in the input will results in reduced performance.

Testbed System Characteristics

The testbed system characteristics refer to the properties of the components, both hardware and software, that form and interact with the test network. We assume that these components are predictable. That is, we assume that the components are

either deterministic or pseudorandom. If a component behaves in a pseudorandom way, we assume that we have the necessary information to predict future actions, and that such prediction is computationally feasible.

1.4 Thesis Outline

This thesis describes YAMA, a system for marking network traffic with the associated high-level events. Chapter 2 gives an overview of the event-traffic model that serves as a basis for YAMA's design. Chapter 3 discusses the types and abilities of the adversaries who disrupt the observed network traffic, and by whom YAMA should not be confused. Chapter 4 develops our strategy for correlating events with network traffic. Chapter 5 outlines our design for YAMA, and Chapter 6 describes the implementation for YAMA 1.0, a version of YAMA that marks web traffic and the associated DNS traffic. Chapter 7 shows the YAMA 1.0 evaluation procedures and results. Chapter 8 includes some closing thoughts and ideas for future work.

Chapter 2

A Model For Event-Driven Network Traffic

In this section we develop a model for traffic on a packet-switched network as a causal system driven by events. This model explains network traffic as a consequence of high-level events, and provides a means by which network traffic data can be linked to those high-level events through a series of intermediate events. The model consists of three types of components: agents, events, and data. Throughout this chapter, we will illustrate the interactions between these components as "event graphs."

Agents Agents are the processing components of our model. Agents generate events and data in response to received events. In the process of generating new events, agents may also reference stored data. Agents are represented by boxes in our event graphs.

Events Events describe actions undertaken by agents. Events occur at a specific time, and carry with them all data necessary to describe the undertaken action. Events are represented by rectangles with rounded sides in our event graphs.

Data Data refers to static information generated by an agent in response to some event. It can be stored, referenced, and transmitted, but in itself denotes no action until processed by an agent. Data does not change once generated. When stored data is altered, we say that it is replaced by new data. *Network traffic* is simply data transmitted over the network. Data is represented by parallelograms in our event graphs.

Figure 2-1 presents a simplified version of our event-driven traffic model. This simplified model shows a user event driving an application, which then generates a series of API call events to host-provided network services. The network services respond by transmitting packets over the network.

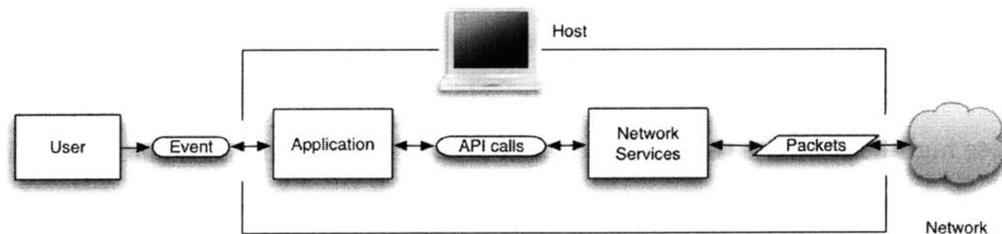


Figure 2-1: A Very Simplified Model for Event-Driven Network Traffic

In the following sections we further develop our model. We begin by describing the main components in detail. Then we describe how the components are causally interconnected, introducing a graph representation that links agents, events, and data. Finally, we show some examples of how our model can be used to describe a few simple web-based transactions.

2.1 Agents

Agents are responsible for generating events and data in response to received events. The nature of the response is described as an agent's *behavior*. Agent behavior varies with the received event and the state of the agent at the time the event is processed.

Agents are abstractions for a set of behaviors and states held within a system. Because they are abstractions, our agents may change with the depth of the model. For example, a high-level model might include users (people) and hosts (machines) as its agents. A very low-level model might include neurons and transistors. High-level agents can often be described using collections of lower-level agents, and vice versa.

Figure 2-1 shows the level of abstraction that we have chosen for the purposes of this thesis. In general, our agents can be described as users, applications, or network services.

Users Users interact with the network to accomplish goals via a variety of information transactions, each of which is initiated by an event. Groups of users work to accomplish related missions by producing and reacting to data and events.

Applications Applications are running programs which users interact with in order to complete certain tasks. Applications include email clients, web browsers, and word processors. Applications must be running to serve as an agent.

Network Services Network services refer to active operating system (OS) implementations of the various network protocols. These services include OS implementations for the Ethernet, IP, TCP, and DNS protocols, among many others.

2.2 Events

In general, events can describe any actions performed by agents in our system. For the purpose of this thesis, we are only concerned with the events which affect network traffic, either by directly or indirectly producing the traffic, or by changing the state of the system and thus affecting the form of future traffic Any event belonging to an event chain that leads to observable bits on the network is said to generate traffic on the network (more on chains later).

Events are generated through agent behaviors, and encapsulate all the data inherent in the agent's action. For example, if the action is a function call, the event includes the parameter data. Because events are so closely tied to agents, the types of events in our model will be determined by our level of abstraction for agents. With this in mind, we describe some general classes of events in our model, and provide examples of the agent behaviors that might lead to such events.

User Events A user event is a command or action which drives a standard user-level application. Since user-level applications include web browsers, mail clients, word processors, messaging clients, and music players, user events include requesting a web site, sending an email, saving a written document, blocking a buddy, and playing a song. Some of these events may not always result in network traffic. For example, saving a document to a local drive might not generate any network traffic, but saving to a document to a network share would.

Application Level Events Application level events include actions initiated by user-facing applications. These usually involve calling functions on an application programming interface (API) provided by a separate application or service. Examples of application-level events include requesting the IP address of a given host name

from the DNS service in response to user web-page request, initialing a connection to a new server in response to images referenced in received HTML, or updating the system time through an NTP server in response to a timer.

Network Service Events Lower level events occur between the user-facing application and the network card, and are usually performed by the operating system on behalf of the application. Network service events include function calls between network services within a host, and the events in the hardware which directly result in network traffic. For example, an operating system's TCP implementation might decide that it should increase the window size for a certain connection. As with application level events, events at this level can occur as responses to other events, incoming data, or internal timers.

Physical Layer Events So far we have focused on events affecting software, however, we can easily extend our event model beyond specific hosts into the physical domain. Physical layer events include unplugging an Ethernet connection, plugging in a connection, or moving a wireless device between access points. These events produce setup or error traffic in most networks, and would affect future traffic. Physical layer events might be generated by users, or represent other happenings generally outside the scope of this model, such as natural disasters.

2.3 Data

Data forms an important component of our causal model. Referenced data can very directly affect events and network traffic. Data also provides a pathway for events to affect agents they do not interact with directly. Data is immutable and separate

from its interpretation. For example, an incremented integer is simply a new integer that was created by referencing the old integer and adding one. An image file may be a simple picture, but when processed by the wrong image library, may end up containing executable code. Data can also contain attached metadata, which is passed on as that data is referenced. We now describe some forms of data in our model.

Application and Network Settings Application and network settings affect the behavior of the agents to which they belong. For example, a browser might be configured to use a proxy server or to never send cookies. Other settings include IP addresses, default DNS servers, and operating system settings that launch select applications at startup.

Files The content of files can greatly affect future events. For example. changes in a source file can change the way it is compiled and eventually run. Even changes in simple HTML can have consequences when that HTML is interpreted by a browser. The HTML specifies among other things, which image files a browser should request from their respective servers. Executable programs are special files which can generate myriad events when processed by the operating system.

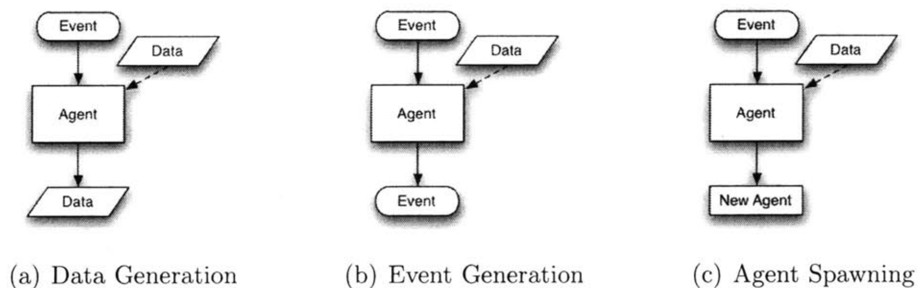
Network Traffic Network traffic is any data transmitted over the network. There are many ways to describe the series of bits that make up network traffic. In packet-switched networks such as those using the Internet Protocol [27], these bits can be aggregated into packets with sources and destinations. Those packets can then be organized into flows, each forming a single channel of communication between two hosts. With further analysis these flows can be joined into sessions, or split into the

application-specific data elements with which applications communicate.

2.4 Relationships Between Agents, Events, and Data

In this section we describe the relationships between agents, events and data. These relationships can be expressed as directed edges between agent, event and data nodes in a graph. We will call such graphs "event graphs."

Figure 2-2 shows some of the primary relationships between components of our model. In general, an agent responds to an event by referencing previous data and taking some action. This action can involve generating data or further events, and even creating new agents.



(a) Data Generation (b) Event Generation (c) Agent Spawning

Figure 2-2: Example Relationships Between Model Components

2.5 Examples

In this section we provide some example applications of our event-traffic model. We begin with a simple web transaction in detail, then look at a more complex web

transaction. In each example we assume an Ethernet network [16].

2.5.1 A Simple Web Transaction

Here we explore in detail the agents, events, and data involved in completing a simple web transaction. The transaction can be broken up into two high-level steps: the client request and the server response. Figure 2-3 illustrates the overall steps involved in the transaction. 1) The user requests a page load from the host's browser; 2) The host sends a request to the server serving the content; 3) the server receives the request and generates a 4) response; 5) the host receives the response and loads the page, which the user then views.

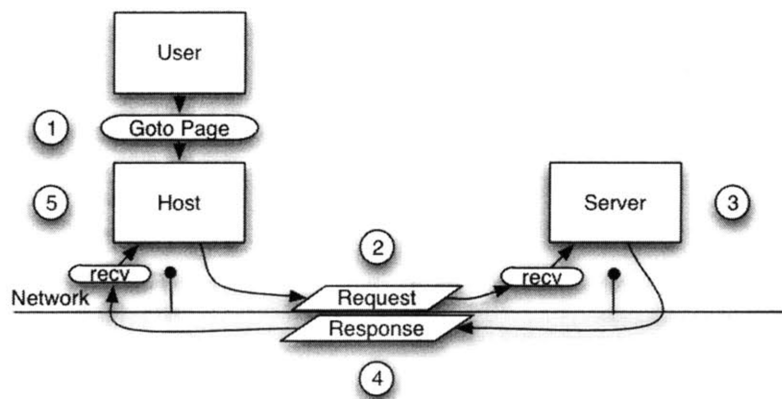


Figure 2-3: Top Level Event Graph for A Simple Web Transaction

The Client Request Figure 2-4 shows a more detailed view of the client request generation. TCP, IP, and Ethernet refer the the protocol implementations provided by the host operating system. 1) The user generates a goto event for a given IP

address-based URL. 2) The web browser connects to the server and sends an HTTP GET request for the '/' resource. 3) TCP initiates and manages the connection, and sends the request. This involves many lower level events in accordance with TCP's defined behavior. 4) IP generates and receives IP packets. 5) Ethernet determines the next hop Ethernet address, and generates the 6) packets transmitted over the network. 7) does not show a single step in the request, but highlights how the original IP address data generated by the user is inherited by the various events and data, including the packets transmitted on the network.

2.5.2 A Complex Web Transaction

Here we expand upon the simple web transaction of the previous page to illustrate the events, agents, and data involved in a web transaction in which the user provides a DNS host name for the server, and the requested html data includes a series of sub-entities which must be fetched as part of the page (on the web these are usually image, CSS, flash, or javascript files).

The basic flow of events and data is shown in Figure 2-5. The transactions for the page source and each subsequent sub-entity follows the basic steps: 1) get the IP address of the server hosting the entity (html, image, etc.) and 2) GET the entity from the specified server.

Event, Agent and Data Relationships We can unroll the high-level event graph in Figure 2-5 to create an event graph which more clearly shows the relationships between the events, agents, and data involved. We continue to use a high-level abstraction for the agents involved, and deal with application-level data rather than the individual packets.

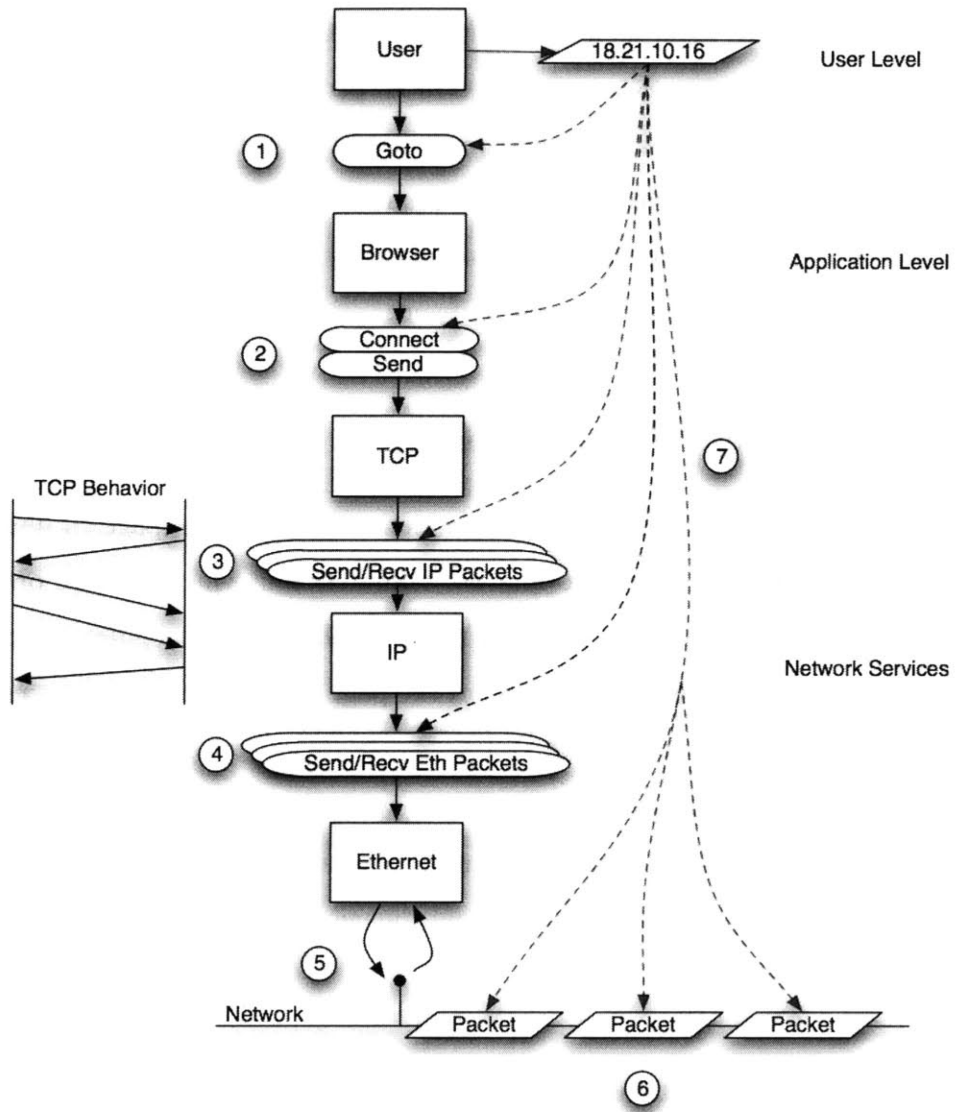


Figure 2-4: Client Side Step in a Simple Web Transaction

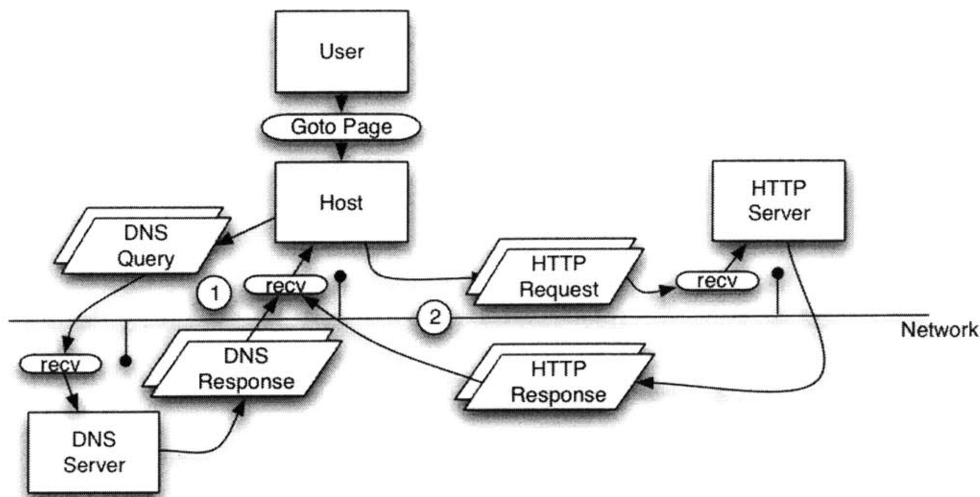


Figure 2-5: Top Level Event Graph for A Complex Web Transaction

Figure 2-6 shows the relationships between model components during the initial HTML source fetch in a complex web transaction. We begin with 1) the user's generation of a goto event for the site http://web.mit.edu. 2) the host resolves web.mit.edu by issuing a DNS query to a default DNS server, and 3) the returned address determines the web server from which 4) the host requests the html source. 5) The web server responds by generating the HTML source and transmitting it to the host.

Figure 2-7 shows relationships after the HTML source is received by the user host. 1) The host processes the received HTML and determines the need to fetch http://images.mit.edu/img.png. This prompts 2) a DNS Query for the host name images.mit.edu. 3) The default DNS server responds with a Name Server (NS) record [24]. This record contains the address of a new server which should be queried for the host name. 4) The host queries this new DNS server and 5) gets an address,

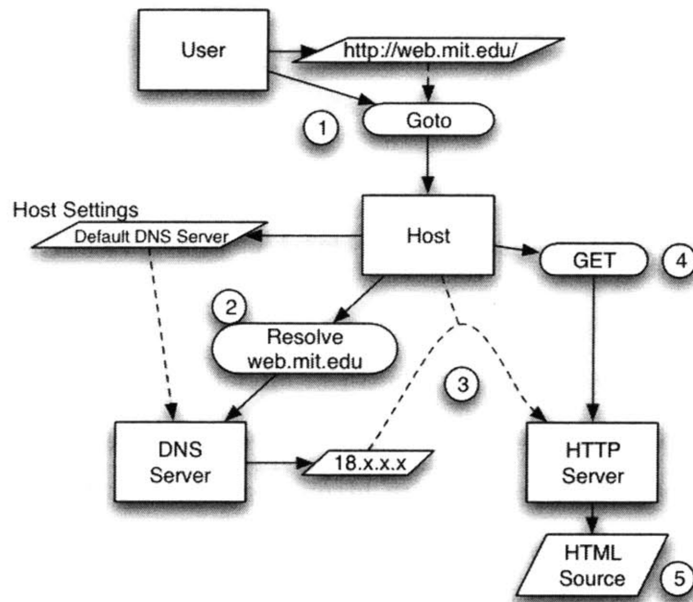


Figure 2-6: Example Component Relationships when Fetching HTML Source

which is used to connect to the HTTP server containing img.png. 6) The host requests the image from the images.mit.edu server, which responds with the image file data.

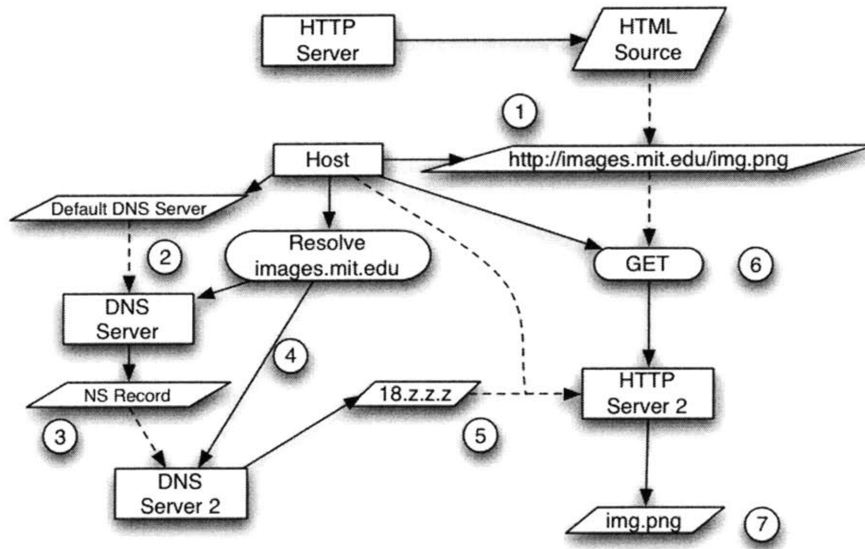


Figure 2-7: Relationships for Fetching HTML Sub-Entities

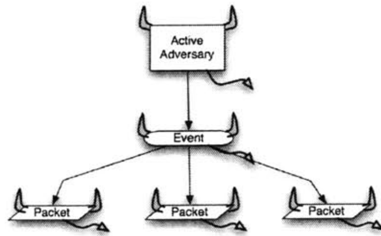
Chapter 3

Adversary Model

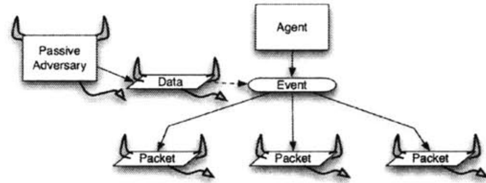
In this section we develop a model for adversaries within the context of our event-traffic model. Adversaries take the form of agents who carry malicious (or non-mission) intentions. Adversaries propagate these intentions by generating events and data. In the course of propagating their intentions across the network, adversaries will cause changes in the observable network traffic. We can describe adversaries as either *active* or *passive* based on their action's effects on the observable network traffic.

Active Adversaries Active Adversaries will insert additional traffic in to the network. In terms of our event traffic model, an active adversary will generate events on a host that directly lead to the generation of network traffic, as shown in Figure 3-1(a).

Passive Adversaries Passive Adversaries will change the form of normally occurring traffic. In terms of our event traffic model, a passive adversary will alter the data referenced by other events or agents. The referenced data leads to changes in the observed network traffic as in Figure 3-1(b).



(a) The active adversary generates events, leading to new traffic on the network.



(b) The passive adversary alters referenced data, effectively altering otherwise normal network traffic.

Figure 3-1: The Active and Passive Adversaries

The presence of either active or passive adversary activity has direct implications for the appropriate marking of user-event traffic. An active adversary will generate traffic which should not be confused with the standard user traffic. The passive adversary will lead to traffic, which, while tainted by the adversary, should still be considered the result of some user event. Detection of a passive adversary thus requires a deeper validation of the observed network traffic data.

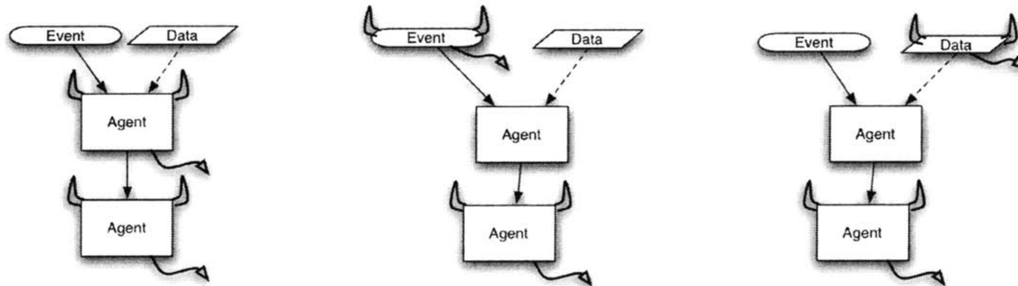
In the next sections we establish the mechanisms for intent propagation in our event traffic model, then explore the active and passive adversaries in more detail providing real-world examples of the powers of each. Finally, we discuss some of the consequences of this model in relation to the marking of user-event traffic.

3.1 Intentions in our Event Traffic Model

In this section, we develop a method for determining the intentions of agents, events, or data on the network. Adversaries, much like regular users, have missions that motivate their generation of events and data. With adversaries, these missions are based

on some malicious intent, whether it be to gain unauthorized control of a resource, steal something, or disrupt the legitimate missions of benign users. Intentions are carried and propagated by agents, events, and data alike.

Intentions and Agents Malicious intentions begin with some malicious agent. This malicious agent produces events and data to accomplish his mission. He may take an active or passive approach (or both) to achieve his goals directly, however, it is often the case that an adversary will spawn other agents that further his intentions. Figure 3-2 shows how this happens, either directly such as by the mis-compilation of a benign piece of code [34] (Figure 3-2(a)), through a malicious event such as the misuse of an otherwise benign application (Figure 3-2(b), or through referenced malicious data such as a replaced executable file (Figure 3-2(c)). Agents have the property that even if they process malicious events and data, it is not the case that they become malicious. Thus an agent could process a malicious event and generate malicious data, then follow by processing a benign event and generating good data.



(a) From A Malicious Agent

(b) Through a Malicious Event

(c) By Processing Malicious Data

Figure 3-2: Spawning a Malicious Agent

Intentions and Events and Data Events and data are the vehicle by which intentions propagate. Each carries the intentions of its source agent, parent events, and referenced data. The main distinction is that while events will propagate their intentions and disappear as soon as they are processed, data can be saved and referenced, propagating its intentions each time. When data is overwritten, it is replaced by new data carrying new intentions. The new data will only derive intentions from the old data if it was produced using some information-preserving process.

3.2 The Active Adversary

The active adversary is an agent who generates his own events, resulting in additional traffic on the network. The nature of this additional traffic can vary depending on the level at which the adversary operates.

3.2.1 Generation of "User" Events

The generation of user events by an adversary is characterized by the adversary's driving of a common user application. The effect is a pattern of traffic indistinguishable from that of a regular user action. However, this is an event that no authorized user intends, and possibly one that would never be performed under normal circumstances. Figure 3-3 illustrates how an adversary generates user events.

Adversaries which introduce user events include adware programs and some major internet worms. For example, adware commonly drives web browsers to produce popup windows showing advertisements. On the network, the activity is indistinguishable from that of a user browsing to the same advertisement site. The infamous Melissa [10] and and LoveLetter [11] Worms also introduce user events, driving the

Outlook application to send email containing themselves to recipients in a user's address book.

User applications can be configured and run through scripting capabilities of the application, or by altering the application itself through plugins or the replacement of modules. In a testbed environment like LARIAT [29], the only difference between user and adversary events may be the record of the user event.

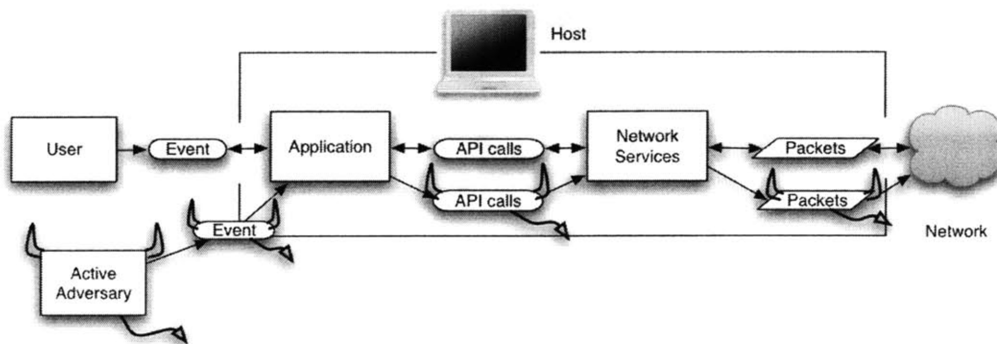


Figure 3-3: Generation of "User" Events by an Active Adversary

3.2.2 Session Initiation

Session initiation is the direct introduction of session traffic into a network without employing a user application. The session is initiated directly by the adversary using network services provided by the host operating system. It may use a standard, esoteric, or custom protocol.

Many worms initiate their own sessions for purposes ranging from downloading of malicious code to run, to checking the time, to infecting other systems. Several variants of the Sobig worm actively check for malicious code to run by initiating

connections to author-provided web sites [19]. Sobig also initiates NTP sessions with network time servers in order to throttle its own spread. CodeRed I initiates TCP connections to the HTTP port of pseudorandom addresses in order to propagate [33].

Initiation of sessions is achieved mainly by custom software utilizing the networking functionality provided by the operating system.

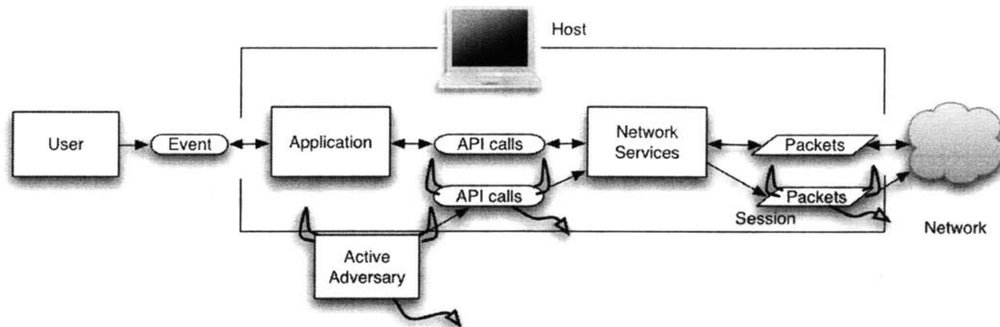


Figure 3-4: Session Initiation by an Active Adversary

3.2.3 Packet Injection

Packet injection is the introduction of "hand-crafted" packets into a network. The packets might be consistent with the host operating system's networking protocol implementations. Injected packets may induce a response from a recipient, but otherwise may be followed by no other related traffic. Injected packets can have spoofed source addresses, and may interfere with existing sessions in the network.

Injected packets can be used to exploit vulnerabilities in remote hosts, hijack or disrupt other sessions, or carry out Denial of Service (DoS) attacks. The Slammer worm used a specially-crafted UDP packet to exploit a vulnerability in Microsoft's

SQL Server [12]. TCP session hijacking is a well-documented practice which can lead to compromise of target systems [14]. The SYN flood DoS attack often uses injection of packets with spoofed source addresses to help hide the source (or sources) of the attack [9].

Packet injection can commonly be achieved using low-level networking APIs provided by a host operating system, or through specialized applications like netcat.

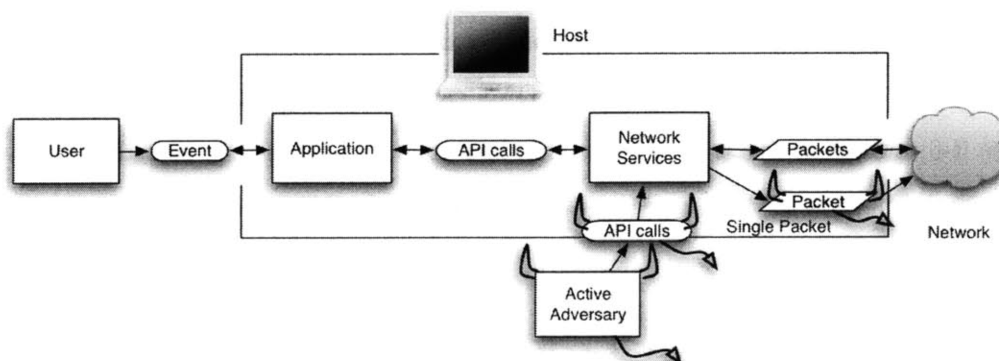


Figure 3-5: Packet Injection by an Active Adversary

3.3 The Passive Adversary

The passive adversary does not directly introduce any traffic into the network. Any changes in the natural flow of traffic are achieved by alteration of referenced data [35]. The passive adversary can change the form of network traffic in ways not easily detected, even by changing specific bits in user-generated outgoing packets. Passive adversary data might stay unreferenced and dormant for long periods of time, only to rise when activated by other activity. The next sections provide highlight some of

the powers that might be exercised by passive adversaries.

3.3.1 Session Redirection

Session redirection involves the alteration of data referenced by a hosts network services so that sessions from a host are aimed at the wrong destination. One way to do this is by confusing a host's DNS implementation. This can be done by editing a user's "hosts" file, manipulating a local DNS cache, or compromising a DNS server. Figure 3-6 shows an event graph for this last example (compare with Figure 2-6). The overall effect is that an intended exchange is had with an unintended party.

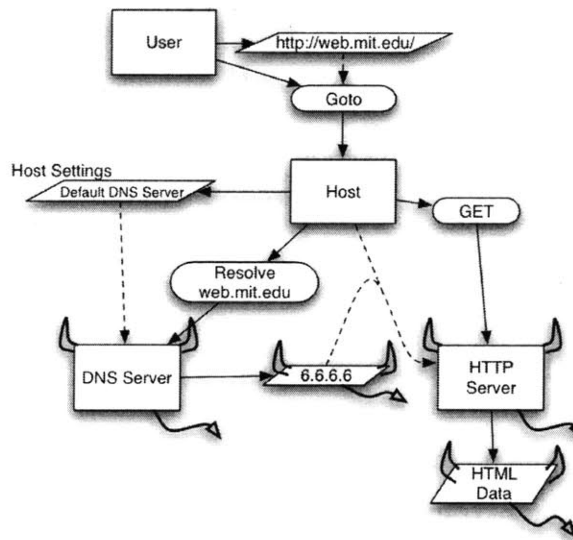


Figure 3-6: Session Redirection via a Compromised DNS Server

3.3.2 File Tainting

File tainting refers to the replacement of all or part of a file by a passive adversary. If the file is on a server, clients accessing the file will be affected when they make use of the file. Tainted files can be activated by execution (where an executable file is processed by the operating system and loaded as a process), or through processing by some other software. Tainted files are at the heart of "contagion" worms, which spread surreptitiously through a user's regular actions [33].

3.3.3 Packet Alteration

While session modification deals with the data at the application level, packet alteration deals with everything else. Packet alteration is the modification of individual bits of a packet. Packet alteration can also be used to perform session alteration by directly changing the destination address of individual packets. In effect, a passive adversary who can alter individual packets owns all the traffic that comes from the controlled host. Figure 3-7 shows the basic operation of the packet-altering passive adversary. The intentions of the adversary are only propagated by the bits actually altered by the adversary, and not necessarily by the entire packet,

Packet alteration can be used for many reasons, but one of the main applications is the hiding of data in standard user channels. There are many documented strategies for doing this [6] [22]. Some uses of packet alteration may be legitimate and useful, however, such as the user event information appended in [28].

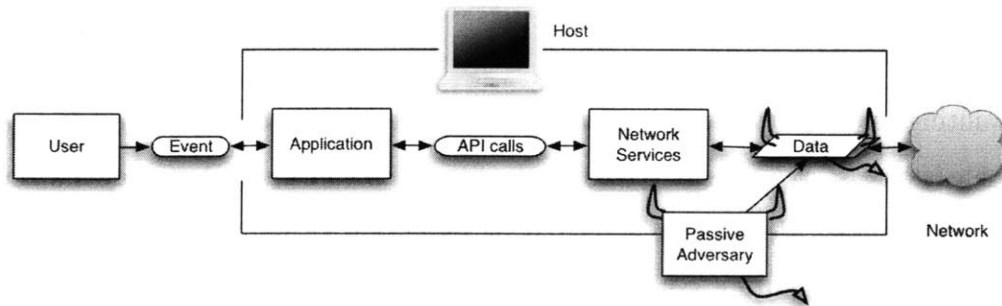


Figure 3-7: The Packet-Altering Passive Adversry

3.4 Consequences for Traffic Marking

3.4.1 Dealing With The Active Adversary

Consequences of an active adversary vary somewhat with his intentions. If the adversary is behaving like a single user, then this means our model for traffic generation must be correct for the standard operating modes. We must be able to distinguish between similar traffic patterns caused by separate events. This allows us to distinguish the adversary traffic from the benign user traffic.

If the adversary is attempting to disturb user activity, either by interfering with user sessions or by rendering important services inactive through an active denial of service attack, we must provide a model for how benign agents react to error, as these error responses are caused both by the original event, and indirectly by the interference from the adversary.

3.4.2 Dealing With The Passive Adversary

Since the passive adversary deals with data, we must be able to deal with possibly unusual data. Any parsing or processing of the observed traffic data must be robust enough to deal with data that might, for example, exploit a buffer overflow in another application. Such data, crafted to confuse an application, must not confuse our traffic marking system. Detection of the passive adversary requires the ability to validate data. Tracking passive adversary activity requires a bit-level resolution for marking so that the passive adversary intent is not incorrectly propagated.

The use of packet alteration for the purpose of covert communication have been studied extensively. A framework for analysis of covert communications in TCP/IP is presented in [22]. Others have addressed the problem in [18] and [30].

Chapter 4

Correlating User Events and Network Traffic

As described in Chapter 2, high-level events are connected to network traffic through a causal chain of events (Figure 2-1). For a given event/packet pair we must re-create an event chain linking the two (Figure 4-1). In this section we describe a process for linking high-level events and observed traffic by matching a set of expectations derived from the high-level events with a set of inferences about possible source events derived from the observed traffic.

4.1 Bridging the Causal Gap Between User Events and Observed Traffic

The procedure for establishing correlations between user events and network traffic relies on YAMA's ability to generate expectations for events generated by the user event, and inferring high-level events and their associated data from the ob-

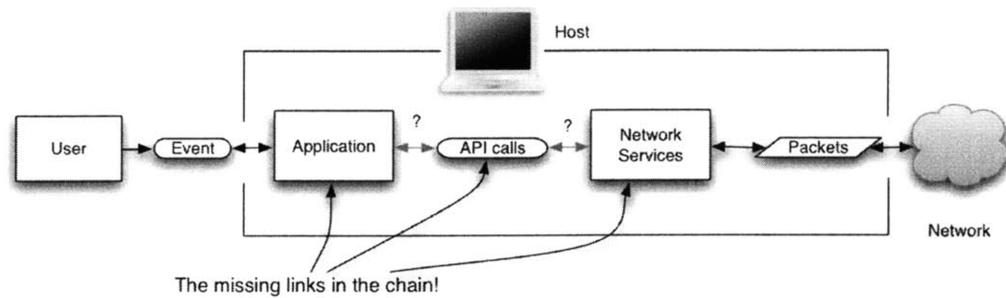


Figure 4-1: Given Events and Traffic, How do we bridge the gap?

served network traffic. We then require a method for matching our expectations and inferences. Figure 4-2 illustrates the three steps in this process.

4.1.1 Generating Expectations for Events

Generating expectations from high-level events requires behavior models for the agents involved. Once these models are in place, we can use them to propagate user level events into a series of application and then network service events. The deterministic nature of most application software makes the modeling of application agents feasible. The behavior of lower level network service agents is largely dictated by specified protocols, and this makes modeling possible. A few testbeds implement network service agent models in order to generate pseudo-realistic network traffic [8] [15]. However, quirks in implementations for each operating system, as well as the many different protocol optimizations employed by each makes completely accurate behavior modeling at this stage extremely difficult [25].

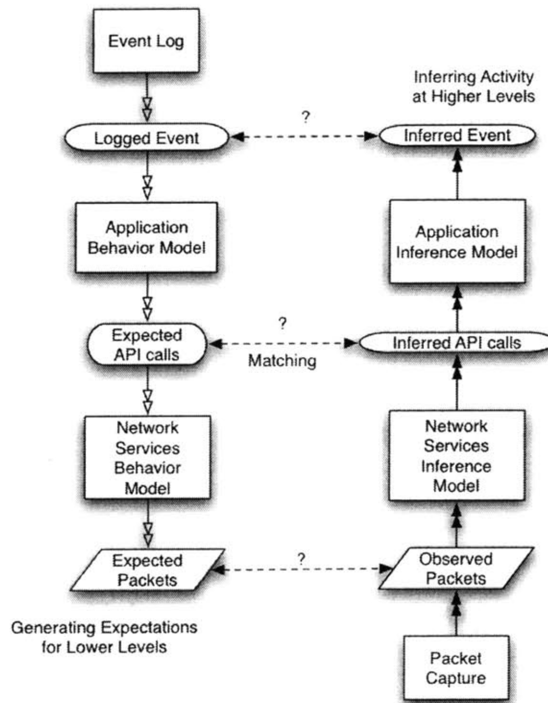


Figure 4-2: Bridging the gap by matching expectations and inferences?

4.1.2 Inferring Events from Network Traffic

The inference of events from their effects is accomplished by the user of agent inference models. Accurate inference models exist for many agents in our system. The deterministic nature of applications again makes modeling plausible. Inference models for network servicesare straightforward. As communication tools, the very purpose of the sub-application network service protocol implementations is to allow a recipient to reconstruct data transmitted by a sender. The encapsulation of high-level protocols within lower level protocols also leaves a clear receipt of the agents involved in generating the packet. We can use the protocols themselves to reconstruct the series of events in the network services leading to each observed packet. This is the domain of network intrusion detection systems (NIDS) such as Bro [26]. The goal of NIDS can be described as the inference, from observed network traffic, of events that can be described as malicious.

4.1.3 Matching Expected and Inferred Events

Matching between expected and inferred events involves their comparison through a set of common features present in the data carried by each. For a function call event, these features include the function name, parameter values, and time of call. Other events will have their own set of matching features.

We could choose to match the corresponding event expectations and inferences at any level. Ease of modeling suggests matching the network API call expectations produced by the application behavior models with the API call inferences made by the network services.

4.2 Correlation in the Presence of Adversaries

Active Adversary Activity The active adversary generates activity which must not be confused with regular user activity. Thus, events inferred from adversary traffic should not be matched with user events. In most cases this will not be a problem because the adversary's malicious intent will differentiate his activity from that of the user. In general however, identical events generated at the same time by users and adversaries are indistinguishable at the packet level. The best we can do is match one and not the other. Then again, in such cases it is not clear that it matters which one we match, as their effects would be the same.

Passive Adversary Activity The passive adversary can be detected by comparison of expected data with inferred data. The inferred data will carry the adversary's intentions, distinguishing it from the data generated from the benign user event.

Chapter 5

Design

5.1 Overview

This section describes the design of Your Able Marking Aide (YAMA), a tool for marking correlations between high-level events and network traffic. YAMA uses the approach discussed in the previous chapter to establish links between large sets of events and observed packets. The system is organized as objects encapsulating the components of our event traffic model. The three types of objects are modules, events, and virtual buffers (vbuffs).

Modules Modules encapsulate our behavior and inference models for specific agents. Modules may reference corpora containing testbed server data.

Events Events are a direct representation of data in our event logs. Events include a time stamp, event type, and associated event parameter data.

VBuffs Virtual buffers (vbuffs) provide a representation for data in our system. They allow us to mark individual bits of data.

These objects are used to create a structure for expectations and inferences which mirrors the structure of agents in our event model. Figure 5-1 shows an example of such a structure for web traffic. We will step through the expectations and inferences generated in this structure as a single web event occurs.

Expectations 1) A user-level event to visit a web page is received by the browser module. The browser module then 2) expects to query DNS to map the site host name to an IP address. 3) DNS consults the DNS corpus to determine the result of such a query, which it 4) provides for the browser immediately. The browser then 5) expects a call to open a TCP connection to the appropriate IP address, and to send an HTTP request for the page URL.

Inferences 6) The Ethernet module receives traffic fed from a traffic capture file, it analyzes the Ethernet header to establish that the 7) packet was requested by IP. The IP module then looks at the packet to determine its type. If 8) the packet came through UDP, the UDP module processes the packet to determine that it contains a 9) DNS message. DNS matches the query expectation it received from the browser with this inferred DNS message to establish a correlation between the packet and event. If the IP module 10) determines that the packet is from TCP, the TCP module will process the packet and others like it to 11) infer incoming and outgoing new connections and transmitted data. The browser module 12) matches its expectations with these inferences. Once the expected request has been made, the browser module 13) queries a web corpus to determine what future requests might be expected.

In the following sections we first show how we perform event-traffic linking by use of a simulation approach. We then describe the design of each object type and

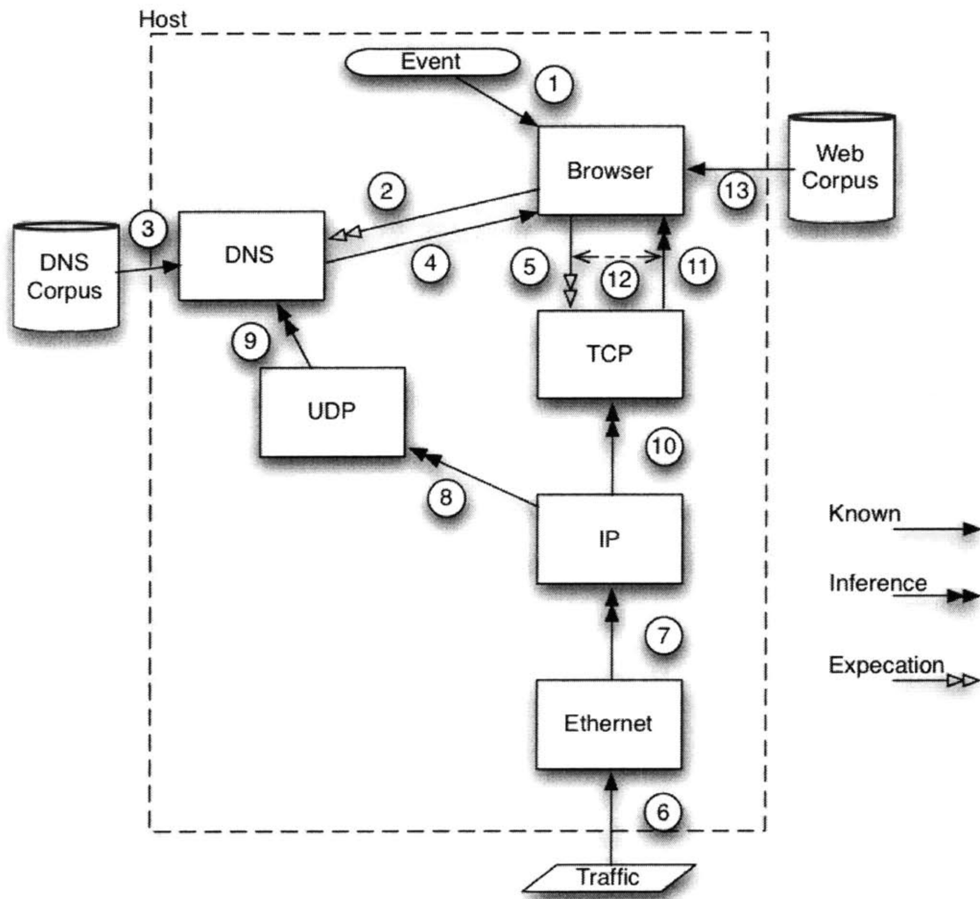


Figure 5-1: A Structure for Correlation: Expectation and Inference

illustrate the basic inner workings of YAMA.

5.2 The Simulation Approach

The temporal nature of events and data can be exploited to ease the task of traffic marking. Instead of attempting to match between the two complete sets of events and observed packets, we can step through the events and packets in time order, generating expectations and matching inferences to those expectations as each is generated. This process can be described as an event-driven simulation [23].

The simulation approach allows us to greatly reduce the computation required, and to automatically enforce temporal constraints in matching. For example, by having event expectations expire after a certain amount of time, we can keep the number of possible matches for traffic inferences to a manageable amount, and mitigate the effects of missing observation data. Because the order of expectation and inference generation is based on time, we also enforce the causality between events and their effects. We cannot match observed traffic to an event that has yet to occur.

5.3 Data Structures and Algorithms

5.3.1 Modules

Modules are the primary object for agent simulation in YAMA. They implement both a behavior and inference model for a given agent. The behavior model is represented in two parts. The first is an event processing interface, through which a module accepts relevant events and sets up expectations for further events and outgoing traffic based on them. The second is through an interface for processing incoming

traffic, which can lead to both expectations and inferences. The inference model is implemented though a traffic-processing interface for outgoing traffic.

If appropriate, outgoing traffic that matches expectations is marked. Generated inferences about higher level activity is propagated in the form of aggregated traffic data. For example, an Ethernet module creates an inference for the IP data sent by the IP module. A TCP module propagates data about entire connections and stream data sent through application modules. The expectations and inferences propagated to and modules are shown in Figure 5-2.

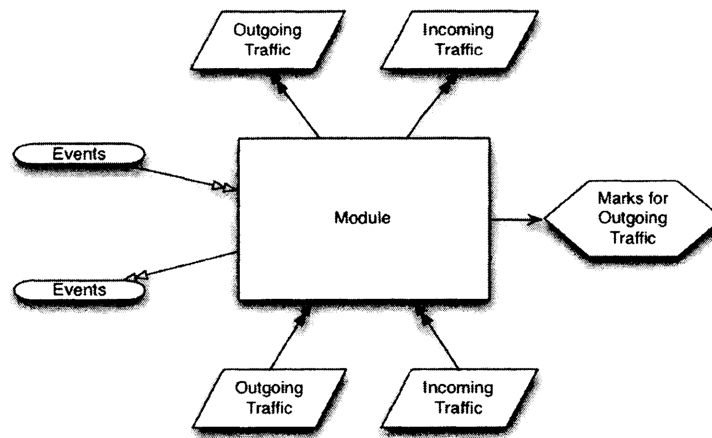


Figure 5-2: A Module

5.3.2 Events

The events objects in YAMA provide a package for event descriptions, be they actual events or expected events. Events can be generated from event log files or by modules, and delivered to destination events for processing. Figure 5-3 illustrates the data held by events.

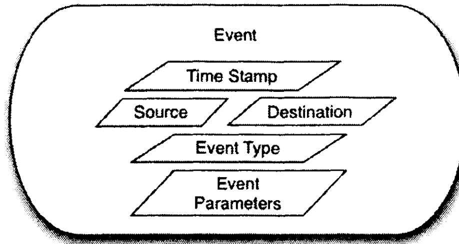


Figure 5-3: An Event

5.3.3 Virtual Buffers

Virtual buffers represent all forms of data in our system. Virtual buffers contain marks which can be propagated to other virtual buffers. The inclusion of markings alongside the data allows for the propagation of event connections and intentions as data moves around the system, even when it is referenced within events and stored as state in modules.

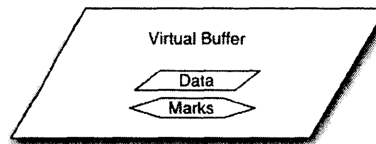


Figure 5-4: A Virtual Buffer

5.3.4 Generating Expectations

As modules are meant to mirror agents in our network model, modules should only process data that is observed by those agents. Likewise data should only be marked by the modules closest to the events that generated it. It is these modules which

hold the state necessary for validating the data. Depending on the depth of our simulation, different modules might hold the responsibility (or rather the privilege) of marking the specific data.

Let us consider the case of a simple client-server protocol, where a client generates requests and a server responds. In a complete simulation, there would be separate modules for the client and server. Each module would be responsible for generating expectations for its output based on its input events and traffic. The client would process an event and generate an expectation for the connected request. The server would receive a request and generate an expectation for the response. In this way, each module only needs access to the state of the agent it represents. Only the client needs to process requests, and only the server needs knowledge of the server contents.

In a more shallow simulation, there might only be a client module. It is then the responsibility of this client to mark the response which it, through its request, is in a sense responsible for generating. The key is that within this simulation, that client module is the one whose events are most directly connected to the server response. In a case such as this, the "client" module requires some knowledge of the server with which it is communicating.

5.3.5 Acceptors

Acceptors provide a mechanism for representing the expectations for specific units of traffic. Acceptors are parametrized to accept and process one specific unit of traffic. Modules can generate acceptors containing the information necessary to accurately describe the expected traffic, and process it further. Often, this processing will involve marking the unit of traffic with the event which generated the expectation. Once an acceptor has met all its expectations, it should cease to accept traffic.

5.3.6 Acceptor Lists

Modules will often generate large numbers of Acceptors at a time. AcceptorLists provide a structure for storing these Acceptors and for supplying them with traffic to potentially accept. AcceptorLists could provide many strategies for presenting traffic to the contained acceptors. One such strategy is to present the traffic to each acceptor in order of creation, and allow only the first one accepting the traffic to process it. Acceptors expire when too much tie has passed for the traffic to be observed.

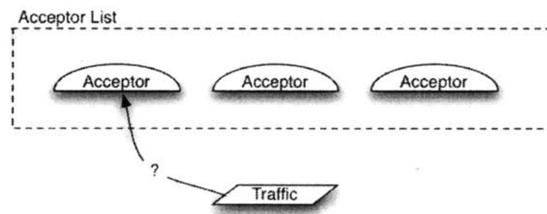


Figure 5-5: Traffic is Tested Against Acceptors

Chapter 6

Implementation

This section describes the implementation of YAMA 1.0 (Your Able Marking Aide, version one). YAMA 1.0 is a proof of concept for our traffic marking system concentrating on web traffic and the associated DNS traffic for a single host connected to the internet. Web traffic was chosen for its ubiquity on corporate networks, and its use as a transport for malware. Web traffic accounts for a large majority of network packets observed (72% of packets on a large enterprise network as measured in December of 2005 [32]).

6.1 Language and Tools

YAMA 1.0 is implemented in C++, and compiled using GCC 4.0.1 [3]. The pcap library provides the tools necessary for reading and writing packet capture files. The open source Boost C++ Libraries [4] are used heavily, especially the smart pointer and regular expression libraries.

6.2 Architecture

YAMA 1.0 includes network processing modules and a browser module. User events (YAMA 1.0 only accepts the browser 'goto' event) are fed directly to the browser module. Traffic data is fed to the Ethernet module. Figure 6-1 shows the implemented modules and their interconnections.

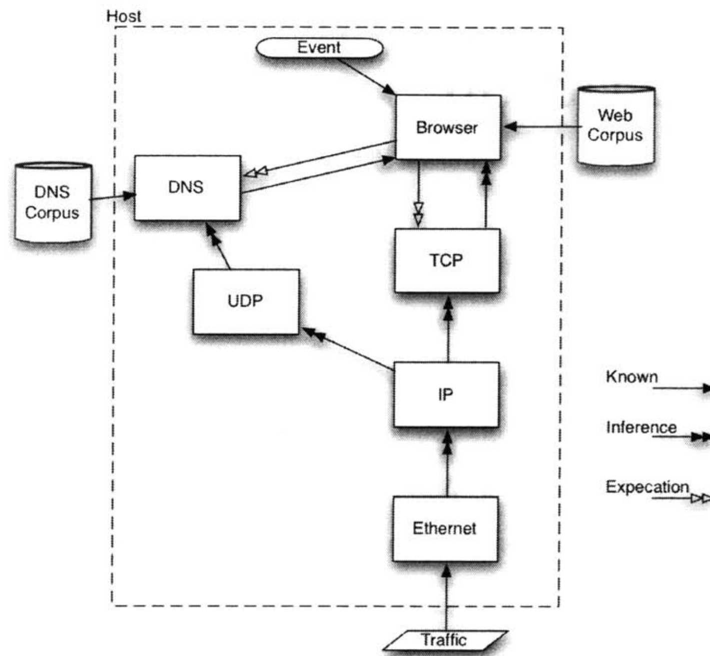


Figure 6-1: Modules and Traffic Flow in YAMA 1.0

6.3 Basic Interfaces

Here we define the basic interfaces for event and data processing in YAMA.

The TrafficProcessor Interface The TrafficProcesor interface is implemented by components which generate and receive inferences from network traffic. It provides functions for accepting and processing both incoming and outgoing traffic. The type of traffic dealt with is variable, allowing TrafficProcessors to deal with their traffic data abstraction of choice.

The EventProcessor Interface The EventProcessor interface is implemented by components which model agent behavior. EventProcessors must provide both a type which determines the types of events that can be handled, and a name to distinguish the EventProcessor from others of the same type. This type/name pair forms the basis for a hierarchical event-addressing scheme. To support event addressing, EventProcessors keep links to both parent and child event processors. Components implementing the EventProcessor interface can register event-processing and remote-procedure functions to be executed when specific events or calls are received.

The Simulatable Interface The Simulatable interface provides functionality for components that change state with the passage of time. It includes functions for updating the simulation time of a given component, and provides implementors with a mechanism for registering closures to be run when the the simulation reaches a certain time. This can be used to implement timeouts and simulate periodic automated actions within components.

6.4 Modules

Modules are the primary object for simulation in YAMA. They implement all three of the above interfaces, allowing them to process both traffic and event data, and

keep track of simulation time. In addition to the functionality provided with each of these interfaces, modules also define a TrafficProcessor target. This target is to be invoked for further processing of traffic data. A traffic target need not deal with the same traffic type as a module. This allows low-level traffic modules to create abstractions of the traffic data for presentation to higher level targets.

Transports Not all components in a system will interact with both events and traffic. Some components might simply forward relevant events and traffic to the appropriate modules. We call these components Transports. Transports can simulate the physical interconnections between components (for example an Ethernet network hub or host data bus), or other high-level event and data propagation mechanisms.

6.5 Data Representation

Here we provide a framework for data representation which allows us to keep track of interesting event connections, and to mark possible areas of passive adversary activity.

6.5.1 Virtual Buffers: The Basic Building Block

Virtual Buffers are the basic data element in our system. They are inspired by Ethereal's Testy Virtual Buffers [2]. We begin by describing the common interface, and then describe the specific virtual buffer types that provide the building blocks for the data in YAMA.

The VBuff Interface Virtual Buffers provide functionality as described by the VBuff interface. This interface is implemented by all virtual buffers and defines the

methods for both data access and marking in YAMA.

Data Access The mechanisms for data access are meant to bridge the gap between the VBuff internal data representation and usable data types. Access to the internal data is provided with bit-level resolution. For example, it is possible to get the boolean bit values from the first and second bits in the VBuff. It is also possible to get the 32-bit integer starting at the 65th bit in the VBuff. Providing such access ensures that our data representation places no limits on the data structures which can be processed. In general, data access is allowed only within the bounds of the VBuff, and errors are generated when there are attempts to access data not contained in the VBuff.

Data Marking The marking interface for VBuffs allows for the marking of individual bit-ranges in a VBuff. Markings are a generalized form of context which can be applied to chunks of data. In YAMA, these markings will generally be used to denote connections from data to the affecting events. Other markings might identify possible passive adversary activity, redundant data (for example retransmitted TCP segment data that was received both times), or other information that might be considered interesting.

VBuffs provide functionality for accessing marks as well, with methods for extracting the mark data for a range of bits in a VBuff. This functionality can be used for propagation of markings between different VBuffs.

RealVBuff RealVBuffs provide the actual data and mark storage in YAMA. They are initialized with real input data and directly store markings applied to them. RealVBuffs in the system would be used as the initial wrapper for external data

such as packets and event parameters.

SubsetVBuff Subsets provide windowed views into other VBuffs. These views take the form of specific bit-ranges in another VBuff. Data access and marking is performed by first translating the indices (given within the windowed context) to the context of the underlying VBuff, delegating the request to the underlying VBuff with the translated indexes.

CompositeVBuff CompositeVBuffs provide a structured way to string individual VBuffs into a single larger VBuff. Data access is performed by looking up the VBuff or VBuffs containing the requested data, and filling in the bits with the data from the relevant VBuff(s). Marking of specific data ranges follows this same basic procedure. However, because CompositeVBuffs can be built in stages, markings applied to "entire" CompositeVBuffs are stored and propagated to VBuffs that are subsequently added.

Other VBuff Types Other VBuff types can be defined for varying reasons, including implementation convenience. One important VBuff type is the MissingDataVBuff. This VBuff can be used to represent gaps in the input data, for example in areas where the traffic data is missing from a packet capture. Such VBuff types can use arbitrary data access and marking strategies, for example returning pseudo-ranodom data or completely ignoring the applied markings.

6.5.2 Representing Higher Level Data

Here we provide some general strategies for representing different types of higher level level data within a virtual buffer framework.

Packets Raw packet data lends itself to storage directly as a VBuff. Input packet data should be structured as a single RealVBuff. Packets reassembled from others can be represented as CompositeVBuffs. Exposing only the VBuff interface for data access allows modules making use of packet data to ignore the difference between the primitive input packet data and packet data reassembled in lower level modules.

Streams Streams lend themselves to representation using CompositeVBuffs. Modules managing the streams can append data one segment at a time by adding VBuffs to the end of the CompositeVBuff structure. Modules that deal with the stream data can treat the stream as one single growing buffer, much as the components they represent might see such data. Streams should provide some form of communication between data managing and consuming modules, for example to announce new data or the closing of the stream by either side.

Application-Level Data Application-specific protocol data can be represented as collections of VBuffs. The final representation completely depends on the specific protocol structure, but separation into the relevant VBuffs allows for easy marking of specific protocol fields or the of the entire structure if necessary.

6.6 The Browser Module

Here we describe the implementation of a web browser module.

6.6.1 A Two-Tiered Structure for Expected Traffic

A web browser processes data at two levels. At a high level, it generates HTTP requests and processes responses. At a lower level it initiates and manages TCP

connections that transport the HTTP traffic. Our browser module generates expectations for both the outgoing TCP connections and the HTTP messages sent through those connections based on the received browser events and incoming traffic. These expectations are represented in two separate TCP and HTTP acceptor lists. In the current implementation our TCP acceptor list simply expects all outgoing connections with destination port 80, and we focus instead on the HTTP level traffic.

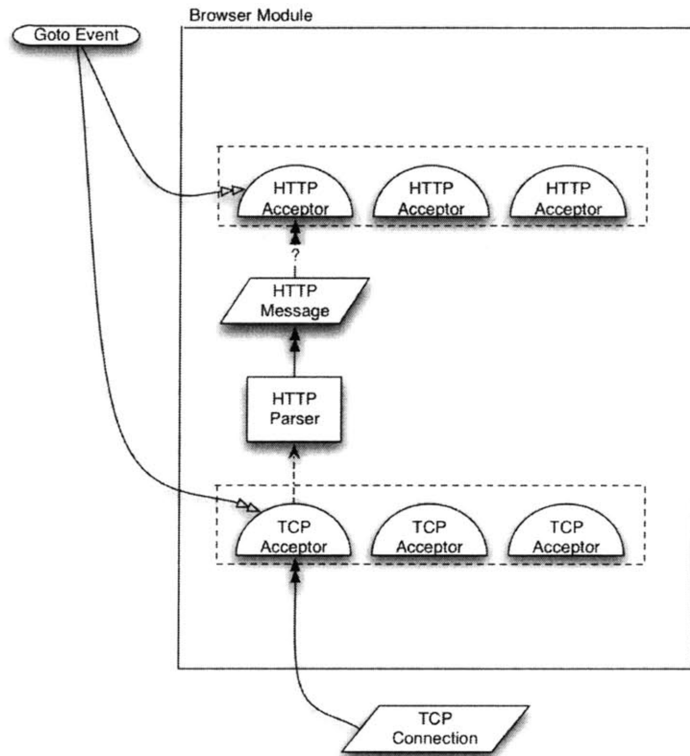


Figure 6-2: The Browser Module

6.6.2 Data Processing in the Browser Module

The Browser Module is a TrafficProcessor for TCP Connections. Upon receiving an acceptable TCP Connection (in our case an outgoing connection with destination port 80), the Browser Module attaches an HTTP parser to the corresponding TCP streams. This parser is driven by incoming data on those streams, and presents complete HTTP messages to the second level of the browser module for acceptance and marking.

Generating Expectations from Events and Observed Traffic At the HTTP level, incoming user events (of which we currently support the 'goto' event for visiting a page) generated expectations for outgoing HTTP requests. Processing of a goto event specifying a target URL will generate an acceptor for a single HTTP GET request/response pair for the target resource. When the request is observed in the outgoing network traffic, we generate additional expectations for HTTP request/response pairs necessitated by sub-entities in the page. These sub-entities include images, CSS, and javascript code which are loaded in response to the page's HTML content. If these sub-entities themselves generate additional requests, those are added to the acceptor list once the original sub-entity request is observed. The sub-entities are determined by lookup in a "Web Corpus," which provides descriptions for entities on the internet, including a list of the sub-entities which will be fetched when a a given entity is loaded in a web browser. A Web Corpus gives us information about browser behavior when loading a given web page.

DNS Interaction For every HTTP request that is expected, the browser module makes a call to a DNS service module to look up the host name for the request URL. In YAMA 1.0, this DNS service is provided by a DNS module set during the

browser module initialization. The DNS lookup serves not only as a means to discover possible connection addresses for the URL, but as a signal to the DNS module that a given hostname lookup should be expected.

HTTP Acceptors An HTTP request/response pair acceptor uses the data within the HTTP request to determine the URL of the requested resource. URLs for HTTP responses are derived from the related request. For HTTP 1.1 requests, we can derive the entire URL from the message using the required "Host" header and the request path. For HTTP 1.0 requests, we ignore the host. Solutions using the data from the DNS lookup above are recommended as future work. When an HTTP message URL matches the URL in a given acceptor, that message is marked as "GOOD." Once an acceptor has processed a matching request, no further requests are matched. When an acceptor has processed a matching response, the acceptor is removed from the acceptor list. Timeouts are not implemented.

An HTTP acceptor is linked to a TCP connection acceptor for its URL's host. When an HTTP acceptor is removed, the TCP acceptor is removed. This behavior is necessary because HTTP 1.1 allows multiple requests per TCP connection, while the general case requires us to expect one TCP connection per request.

Loaded Documents Loaded documents in a web browser might cause periodic web events through the use of javascript timeouts. This information should be encoded in the Web Corpus. YAMA 1.0 ignores this behavior. However, loaded documents do maintain a presence. As long as a document is loaded in the browser, acceptors for its sub-entities remain in the acceptor list. Once a document is closed, all of its active sub-entity acceptors are removed. YAMA 1.0 considers all documents closed when a 'goto' event is received. Thus, YAMA 1.0 is limited to users

who browse one page at a time.

6.6.3 The Need for a Web Corpus

A web browser determines sub-elements that it should fetch by parsing html, CSS, and javascript received from the remote server. The ideal browser module would be able to emulate this behavior simply by processing the captured incoming traffic. Though possible, the handling of missing traffic data and the complexities of the CSS and Javascript engines in modern browsers make such emulation infeasible in YAMA 1.0. Instead, we require such data as an external input in the form of a Web Corpus. The Web Corpus for YAMA 1.0 consists of sets of sub-entities fetched by the browser when a given entity such as a web page is loaded. The Web Corpus eliminates the need for decompressing and interpreting HTTP response data, and makes it possible for YAMA 1.0 to be effective even when data is missing.

6.7 The DNS Module

6.7.1 Performing a Hostname Lookup Through DNS

When resolving a host name for an application, DNS implementations issue DNS requests to different DNS servers according to various strategies. The queries sent might include the exact hostname queried and the hostname with a variety of network-specified suffixes appended. A response including a canonical name (CNAME) record might prompt queries for a new hostname. The initial destinations for the DNS queries are often dictated by a host's network settings, either set by hand or through DHCP. When DNS responses contain records for authorities in the given hostname's domain, subsequent queries might be sent to these new servers. At times

these queries are sent in parallel.

6.7.2 Generating Expectations for DNS Messages

The DNS implementation in YAMA 1.0 generates expectations for specific hostnames in queries, ignoring the hostnames and addresses of the actual servers queried. Expected queries related to the same original hostname are bundled together so that they can all be removed when one of them produces the targeted answer (in our case, at least one address (A or AAAA) record). When a CNAME record is received in a response, an expected query for the new hostname is generated and added to the original bundle.

6.7.3 Data Processing and Expectation Matching

The DNS module is a TrafficProcessor for UDP packets. The implementation for TCP input is left as future work. Received UDP data is is parsed into a DNS message structure giving us access to the different sections including the questions and answers sections. The hostnames in the questions section are used to match both queries and responses with our expected queries. The records in the answers section are used to generate new expected queries (in the case of CNAME records), and to determine that our original hostname has been resolved (an A or AAAA record).

6.8 Matching Expected to Observed Traffic

Here we describe the general guidelines for how expectations should be generated and describe a method for matching traffic to these expectations for marking.

6.9 Simulating Agents as Modules

This section goes into more detail on the general construction and capabilities of modules.

6.9.1 Service and Network Modules

These modules represent the services usually provided by a host operating system, and fit nicely into a tiered structure like the OSI model. Service and network modules should process events relating to settings which affect the way they perform their functions, and to calls or requests from other modules. The calls or events can be implemented through event handlers and remote procedures registered through the EventProcessor interface in the service module, and as service interfaces which can be used directly by other modules with access to the service module.

Service Modules should parse any protocols they use and be able to construct encapsulations of protocol structures for propagation to higher level modules. For example, a TCP module aggregates packets and presents only TCP connections and streams to higher-level modules. When the service protocol indicates the next protocol in the protocol chain, the service module should provide a mechanism for higher level modules to access this information without having to parse the lower-level protocol. For example, an IP module would tag the data section of IP packets with the name of the next protocol (for example TCP or UDP). A UDP or TCP module would then look for the existence of such a tag when determining whether to accept a packet for processing, and as an indicator for the range of packet data that it should parse.

6.9.2 User Application Modules

Application modules are different from lower level modules because they generally propagate data not to other applications, but to the user, who then interprets the data and takes appropriate action. Because our system does not extend all the way up to the user level, our application modules will generally not have a TrafficProcessor target.

User application modules should provide event handlers for all the user events of interest, and be able to parse any relevant application-level protocols. Just as applications call on operating system-provided service APIs to initiate TCP connections and look up host names, application modules should use the APIs provided by lower level modules. These APIs should be used either by directly generating events or remote procedure calls addressed to the lower level modules using the EventProcessor interface, or by finding modules implementing the necessary services thorough the EventProcessor interface and calling the API functions directly. The second approach has the advantage of allowing an application modules to keep a reference to an important service once it has been discovered, eliminating the overhead of delivering each event or remote procedure call to the appropriate EventProcessor.

6.9.3 Driving the Simulation

YAMA can be described as an event-driven simulation [23]. Processing is driven by the input which includes the high events and the incoming and outgoing traffic. This input is pushed to the system by a Feeder in timestamp order. The events and data propagate through the system and are processed by the relevant modules. A Feeder requires both an event target and traffic target. The event target should be the root of our EventProcessor hierarchy tree, and the traffic target should be some common

Transport which can reach all the connected network devices. In the case of two separate networks there might be two Feeders, each pushing the localized network traffic to a Transport on its network. In such a case the two feeders may need to be synchronized in order to ensure in-order presentation of events and traffic to the system as a whole.

Chapter 7

Evaluation

The most powerful assessment of our system would be a measure of its usefulness in the analysis of testbed network traffic. State-of-the-art security testbed systems [5] [29] employ a variety of background traffic usually directly captured from the Internet. Because the primary purpose of YAMA is to correlate user events and network traffic, an assessment of its ability to account for user event traffic will indicate its usefulness to analysts of testbed network traffic. Thus we can use event traffic detection and false positive rates as our primary evaluation metric for the system. By comparing the YAMA 1.0-marked traffic to a ground truth, we can easily identify the correctly (and incorrectly) marked traffic.

7.1 Metrics

Evaluation results are given in terms of counts for packets of varying types, marked and unmarked. Here we describe what it means for a packet to be marked, and how we split up the packets into their respective types.

Interpreting Packet Markings For the purpose of these tests, we ignore the specific region of a packet that was marked, and count a mark in any part of the packet as a mark for the entire packet. As all the markings were "GOOD" markings, there is no need to merge different mark types together. For the remainder of this thesis we will refer to packets with any kind of mark as "marked packets."

Ground-Truth Classification for Packets In order to correctly count marked packets of the types important to YAMA 1.0, we must have an independent method of classifying packets. For this purpose we employ Ethereal [2], a popular open-source network protocol analyzer to create a *packet summary* for a given traffic capture file. This packet summary includes a packet number, destination, source, protocol, and short description for every packet in the traffic capture file. The packet number matches the packet IDs output by YAMA 1.0 with each packet marking, so we can tell if a specific packet has been marked by YAMA 1.0. We can use the protocol and short description output to classify individual packets. The classifications of interest are described in Table 7.1.

Table 7.1: Packet Classifications

HTTP	A packet is considered an HTTP packet if its ethereal summary summary protocol is HTTP.
HTTP Request	A packet is considered an HTTP Request if it is an HTTP packet and its ethereal short description includes the keyword GET.
DNS	A packet is considered a DNS packet if its ethereal summary summary protocol is DNS.
DNS Query	A packet is considered a DNS Query if it is a DNS packet and its ethereal short description includes the substring "Standard query A". This matches queries for both IPv4 and IPv6 addresses.

7.2 The Data Sets

The primary inputs to YAMA 1.0 are a time-stamped list of user events (page visits), a time-stamped capture of the network traffic while those events were generated, and a web-corpus capturing the state of the sites visited. Our data sets include the list of user events and the associated data.

In order to provide some consistency between data sets, while providing complex traffic for YAMA 1.0 to mark, we script the actions of a user browsing popular sites on the Internet. Today's popular Internet sites often include complex user-tracking and advertising elements. Network traffic related to visiting such sites is thus more complex than would be seen using approaches that replay traffic [31] or that use data from just a few years ago [29].

The evaluation uses traffic collected from a single user host connected to the Internet. A data set consists of the events and traffic generated by this user as he surfs, in order, the Alexa "Top 100 English Sites" [1] as measured on July 12, 2006. For the sake of modesty, two of these sites were excluded, leaving us with the Alexa "Clean 98." These sites were fixed and revisited on subsequent dates. A complete list of these sites is included in Appendix A. Sites including an IP address as the host will lead to activity similar to that in the simple web transaction shown in Section 2.5.1. Others will show more complex behavior as shown in Section 2.5.2.

7.2.1 Sampled Site Characteristics

The Alexa "Clean 98" sites can be categorized based on the purpose of the site. Figure 7-1 shows the composition of the sampled sites according to site type.

Commerce sites are those attempting to sell something. Such sites often include

listings for featured products that cycle often.

News sites list the day's news stories, often including pictures for top stories. As with their printed versions, news sites contain revenue-generating cycling advertisements.

Portal sites are those aiming to be an entry point for a user's surfing session. They often aggregate news, weather, user services like email, and advertisements.

Simple search sites are those which focus on search. Often, these sites will contain little more than a search text field and a few identifying pictures. Advertisements are not common until the user inputs a search.

Social sites are stages for user-provided content. They often contain rapidly updated user content and advertisements.

Corporate sites include the official sites of major companies. These sites can remain the same for long periods of time, and often include no advertisements.

Figure 7-2 shows the impact that sites of different types have on the data sets. News sites account for a disproportionately large pergentage of the traffic at all levels, while simple search sites produce a modest number of requests for smaller-than-average page elements.

Figure 7-3 illustrates the complexity of traffic related to sites on the "Clean 98" list as series of histograms generated from a single data set. Figure 7-3(a) shows that the number of HTTP packets generated by a site ranges from the low 10s (simple search sites) to over a thousand (social sites featuring large images). Figure 7-3(b) shows that the number of requests for pictures and other sub-entities in a single site varies from a few to almost 200. These often include small structural page style

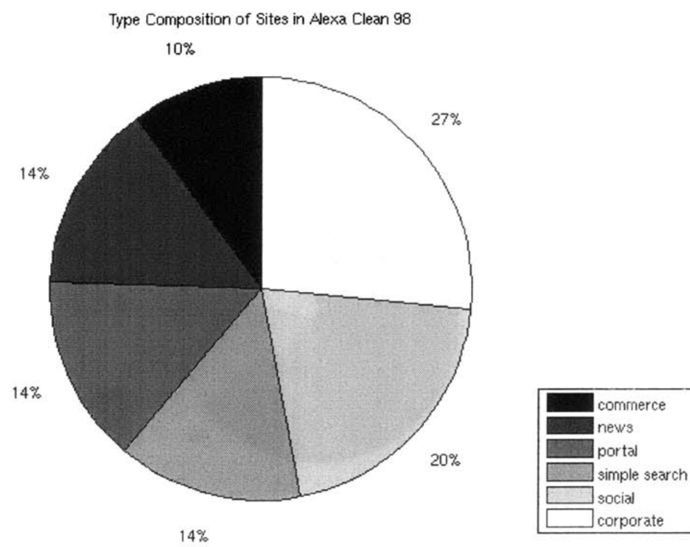
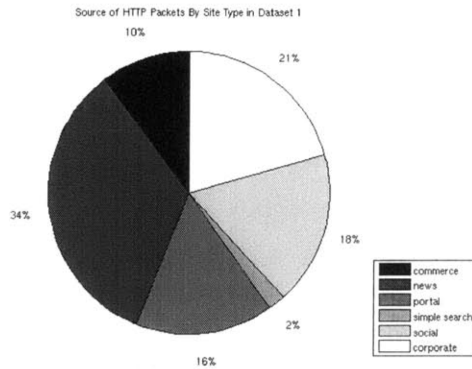
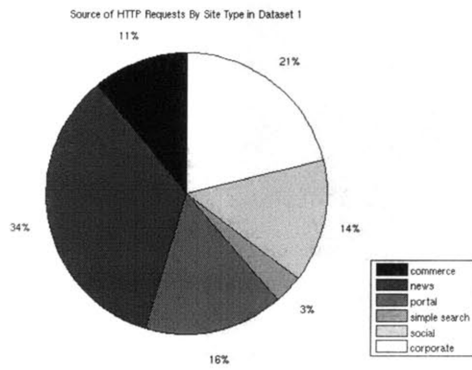


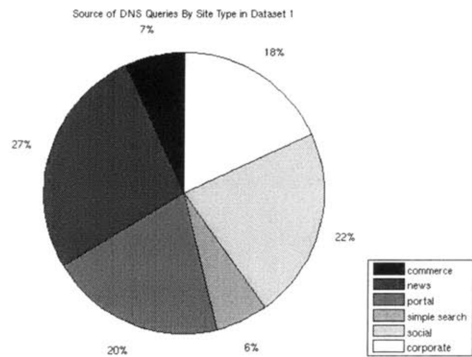
Figure 7-1: Composition of the Alexa “Clean 98”



(a) HTTP Packets



(b) HTTP Requests



(c) DNS Queries

Figure 7-2: Data Set Composition by Traffic and Site Type

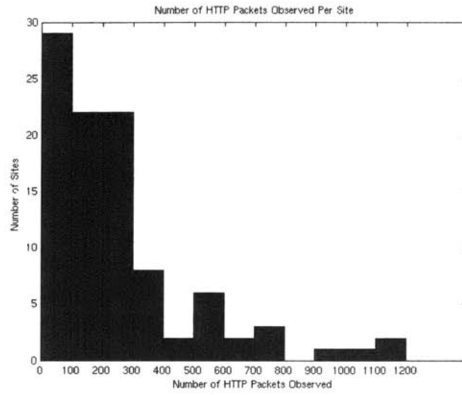
elements and logos. Figure 7-3(c) illustrates the number of hosts that page subentities reside on. A single page load may reference 30 or more servers. Often this is the result of a large number of advertisements and HTTP redirects to different servers used for user tracking. Some sites will also provide load balancing by having referenced content on several mirrored servers.

7.2.2 Data Collection

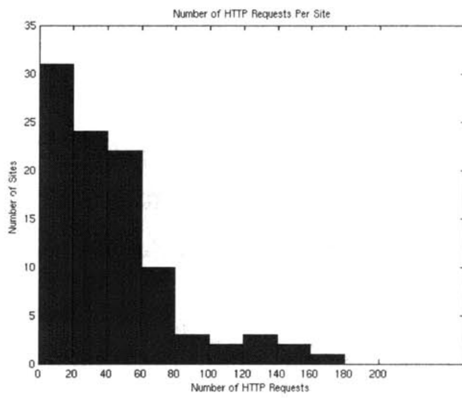
Collection of the data sets was done over the period of several weeks. The first data set, taken on July 12, 2006, served as the base for building a web corpus and analyzing the traffic marking capabilities of YAMA 1.0. The next three data sets, taken a week apart, are used to gauge the changing nature of the Internet. The last four data sets were taken on the same day, an hour apart. Unlike web sites in a test environment, sites on the Internet tend to change constantly, providing a moving target for YAMA 1.0. Table 7.2 shows the details of our collected data sets. Data sets 1-4 were filtered to exclude outside traffic, thus the lower packet counts.

Table 7.2: The Data Sets

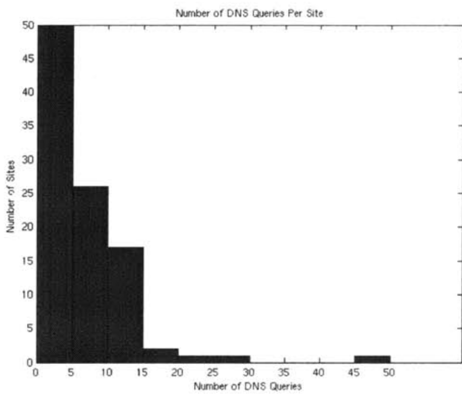
Data Set Number	Collection Date	# Collected Packets
1	2006-07-12	50036
2	2006-07-19	50764
3	2006-07-26	49995
4	2006-08-02	51783
5	2006-08-08	54591
6	2006-08-08	54502
7	2006-08-08	58395
8	2006-08-08	54857



(a) HTTP Packets



(b) HTTP Requests



(c) DNS Queries

Figure 7-3: Histograms Showing Number of Sites with Given Traffic Element Counts

7.2.3 Traffic Capture Setup

Data collection was done using a PowerBook G4 laptop on one of two networks. The first was a switched MIT fraternity network without a firewall (data sets 1-4). The second was a firewalled network connected to the Internet (data sets 5-8). In each case the experiment was isolated from other hosts on the network.

Capture of traffic data is accomplished using the `tcpdump` utility [17], which uses the `libpcap` library to get individual packets from the network card. The "snaplength," which specifies the maximum number of bytes to capture for a given packet, is set above the maximum size for an Ethernet packet, allowing us to capture entire packets.

7.2.4 Event Generation and Logging

User events for each data set were generated by using Applescript to control the Safari 2.0.4 web browser. The Safari browser was chosen for its easy scriptability and convenience on the development machine. It also features a "Reset Safari" option, which allows the user to reset all browser caches and erase all personal data including cookies. This makes it easy to generate multiple data sets with the same base starting conditions.

The Applescript script used to generate events follows the same basic step for every input URL. 1) Close all the existing Safari windows. 2) Open a new window and load an empty URL (about:blank).3) Load the input URL and 4) wait 20 seconds.

For each URL visited, the event generation script appends an event record to the *event file* which serves as input into YAMA 1.0. Each event record consists of a time stamp, an event target application (in our case the browser), an event name (goto), and event parameters (the URL).

7.3 Building a Web Corpus

YAMA 1.0 requires a web corpus that provides, for each URL, a list of all separately referenced components (see Section 6.6.3). Automated analysis of the queries following a specific web page load produces an initial corpus. This corpus is hand-edited to remove parentheses, which YAMA 1.0 uses to denote wild-cards in URLs. We generated two corpora, one from data set 1, the other from data set 5. The corpus used for each data set in the evaluation is given in Table 7.3.

Table 7.3: Web Corpora Used For Each Data Set

Data Set Number	Data Set Date	Corpus Reference Data Set	Corpus Date
1	2006-07-12	1	2006-07-12
2	2006-07-19	1	2006-07-12
3	2006-07-26	1	2006-07-12
4	2006-08-02	1	2006-07-12
5	2006-08-08	5	2006-08-08
6	2006-08-08	5	2006-08-08
7	2006-08-08	5	2006-08-08
8	2006-08-08	5	2006-08-08

7.4 General Test Procedures

This section describes each of the tests we performed on YAMA 1.0 and their results. The tests are meant to assess YAMA 1.0's ability to correctly mark user event traffic while leaving the non-event traffic unmarked. For these tests YAMA 1.0 was configured to mark HTTP and DNS messages related to the given user events. This means that YAMA 1.0 output markings for the data regions in each packet corresponding to TCP data containing those related HTTP messages, the TCP ACK

packets which confirmed receipt of that HTTP data by its destination, and the UDP data segments corresponding to the related DNS messages. All the markings were "GOOD" markings. YAMA 1.0's output markings are saved in a *marks file.*

7.5 Test 1: Detection Rate

This test evaluates YAMA 1.0's ability to mark all user event HTTP and DNS traffic. Our data sets consist of *only* user event traffic, so ideally every HTTP and DNS packet would be marked. This coverage is important because unmarked user event traffic adds to analyst load, and might confuse the analysis of testbed experiment results.

This test also assesses YAMA 1.0's performance, given a static corpus, as each visited site changes over hours, days, and weeks.

7.5.1 Procedure

This test consisted of running YAMA 1.0 on several data sets, using the data set event file as logged by our event traffic generation script. Table 7.2 shows details for each of the data sets. For the remainder of the thesis we will refer to each data set by its number. The web corpora given to YAMA 1.0 as input were derived in part from reference data sets. The corpora thus reflect the state of the web as it was on the day that its reference data set was created. Table 7.3 shows which web corpus was used with which data set, as well as the dates for each. Data sets 1-4 use a web corpus generated from data set 1, while data sets 5-8 use a web corpus generated from data set 5. A test run for a single data set took about ten minutes on a Powerbook G4 1.67GHz Laptop with 2GB of memory.

7.5.2 Results

The data were analyzed to produce a summary of the packet markings for each data set according to type. Figure 7-10 shows the coverage results for each data set by traffic type. Here we clearly note the separation between data sets 1-4 and data sets 5-8 caused by the use of different web corpora. Data sets 1 and 5, used to generate the web corpora which helps mark them, fare extremely well, while subsequent data sets are covered in decreasing amounts. There are also marked differences between coverage for the different packet types. HTTP Packets are more difficult to correctly mark and asymptote lower and later than DNS packets. Furthermore, performance is substantially worse overall than for just HTTP request packets. DNS packet and query coverage is nearly even because there is usually a single response packet for every query packet sent. DNS coverage asymptotes faster - in less than an hour - than HTTP coverage across data sets 1-4 and 5-8.

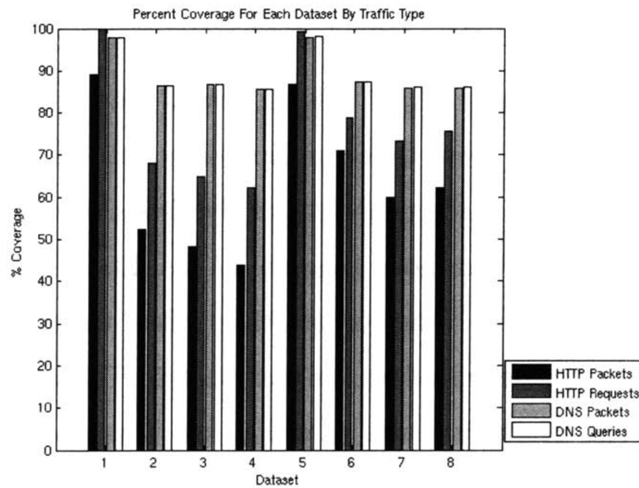


Figure 7-4: Percent Coverage By Traffic Type and Data Set

The coverage within and between data sets can be explained by three factors: the too-specific nature of the web corpora used, changes in the web over time, and missing or mis-parsed network data.

Specificity of Web Corpora Our web corpus is generated directly from the HTTP traffic observed in a specific data set. This accounts for drop in coverage between data sets 1 and 2, and 5 and 6. This large drop is caused by the web corpus containing expectations for web sub-entities that were session or time-specific (such as cycling advertisements, and recent submissions in social sites). We can support this by looking at the overall rates of HTTP Request coverage in data sets 2-4 and 6-8 in Figure 7-8. The sites showing the highest overall coverage and the lowest drop from data set 1 and 5 are the corporate and simple search sites. These are the types of sites generally holding minimal advertisements and dynamic content. The other types of sites, featuring plenty of dynamic content, fall prey to the specificity of the two web corpora.

The Changing Internet Changes in the individual sampled internet sites over time cause the gradual drop in coverage seen through data sets 2-4 and 6-8. These changes take the form of structural updates made to the sampled web sites. The changes can be incremental, as in news stories getting cycled through the days, or drastic, as in a site getting completely redesigned. Figure 7-7 shows the general trend. The HTTP coverage shown in Figure 7-8 shows some of the quirks of this activity. The drop in coverage for portal HTTP requests between data sets 2 and 3 is corresponds with the redesign of www.yahoo.com.

Missing and Mis-parsed Data The missing and misparsed data in the data sets is the source of a of noise, especially in the packet coverage. The disparity between HTTP request coverage and general HTTP packet coverage is the primary result of this mismatched data. Coverage rates for single-packet HTTP requests are unaffected by packet loss as missing packets do not count against YAMA 1.0 (it cannot mark something that is not there). However, HTTP responses are heavily affected by missing data. HTTP response URLs are determined by matching with the corresponding request. A missing request prevents YAMA 1.0 from determining the purpose of the response, and causes it to be left unmarked. A missing response header has the same effect, leaving all the packet carrying the response data unmarked.

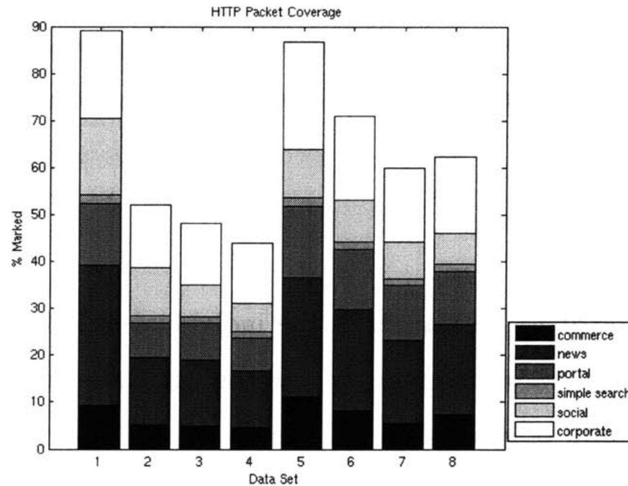


Figure 7-5: Percent Coverage For HTTP Packets by Packet Source Type

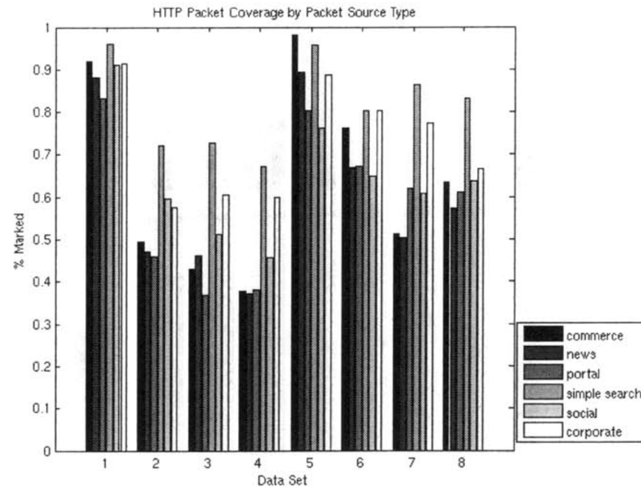


Figure 7-6: Percent of HTTP Packets of Each Type Covered

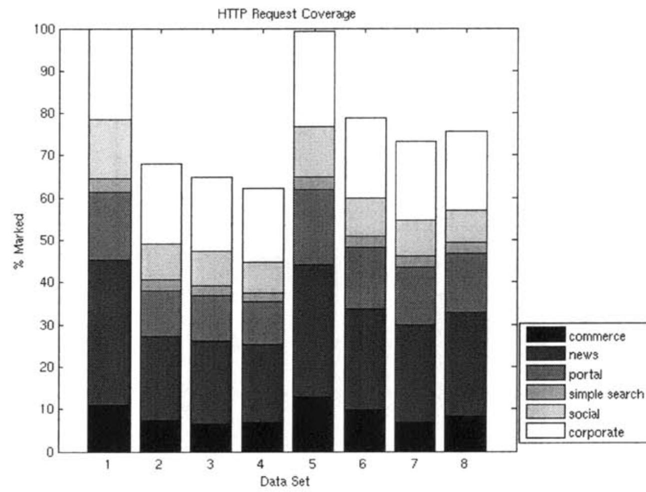


Figure 7-7: Percent Coverage For HTTP Request by Packet Source Type

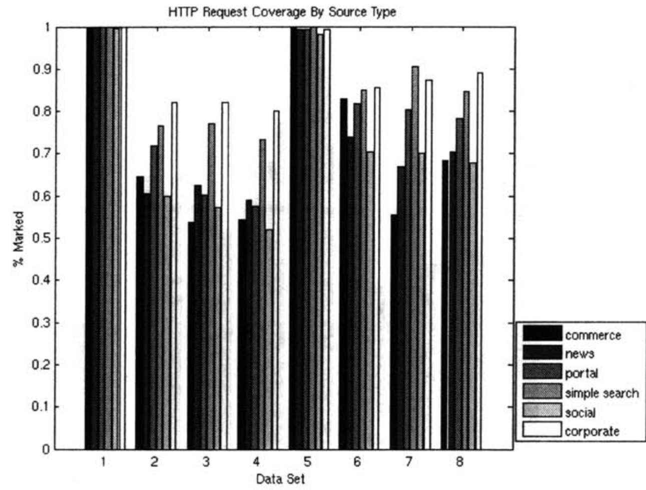


Figure 7-8: Percent of HTTP Requests of Each Type Covered

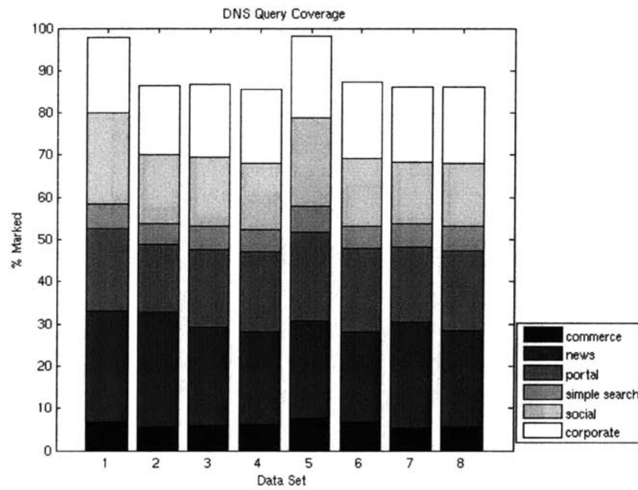


Figure 7-9: Percent Coverage For DNSQueries Broken by Packet Source Type

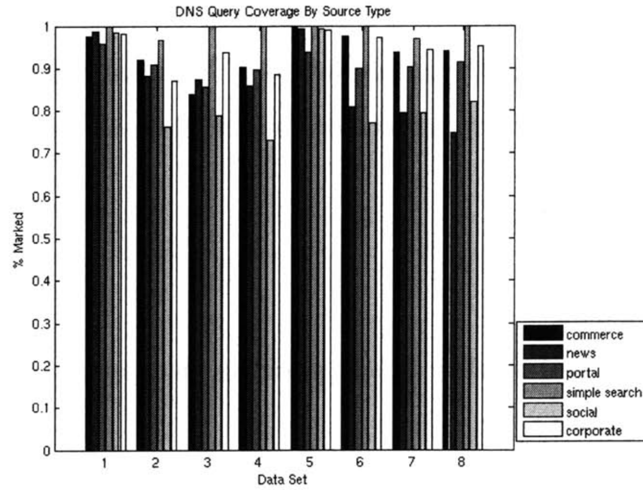


Figure 7-10: Percent of DNS Queries of Each Type Covered

7.6 Test 2: False Positive Rate

Our coverage test lets us know if we are missing anything, but a simple program that marked all packets would perform great on that test while being completely useless. If YAMA 1.0 is to be used to isolate interesting testbed experiment data, it must not mark the non-event traffic as event traffic. Our false positive rate will evaluate YAMA 1.0's ability to do this.

7.6.1 Procedure

In order to measure the rate of false positives, we give YAMA 1.0 an event file indicating a user visits one site while feeding it the traffic generated by visiting a different site, for each of the 98×97 pairs. If YAMA 1.0 marks any of the recorded network traffic as having been generated by a non-matching user event, we score that marked packet as a false positive.

We can also create a confusion matrix showing the number of marks by each of these event files for each of the actual events. The ideal result would show many marks where the advertised event matched the actual event, and no marks anywhere else.

7.6.2 Results

Table 7.4 shows the total number of false positives encountered per traffic type in all of the 98 YAMA 1.0 runs.

Table 7.4: False Positives Per Traffic Type

Type	Total False Positives
HTTP Packets	0
HTTP Requests	0
DNS Packets	0
DNS Queries	0

The lack of false positives indicates that YAMA 1.0 is resistant to confusion by active adversaries generating web-traffic-like sessions and packets. These results are encouraging, especially when noting that many of the sites in the Alexa Clean 98 connect to the same servers for advertising and user tracking purposes. Several of the sites connect to the Google Analytics service for example. The lack of false positives even in the DNS traffic is due to strict matching between the advertised event and the observed traffic. The browser module will not generate expectations for sub-entities such as pictures if it has not observed a request for the main page HTML. This is because a browser's behavior model dictates that sub-entities are fetched in response to received HTML, and not in direct response to the user web request.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis we have discussed the need, theoretical framework, design, and implementation for YAMA, a tool for marking network traffic from associated high-level events. We also performed a series of evaluations on YAMA 1.0.

The need for a tool which correlates high-level events with network traffic is clear. The time and expertise required for analysis of network intrusion detection experiment data would decrease greatly with the use of a tool like YAMA, leading to quicker evaluations for intrusion detection systems, and higher accessibility for analysts.

As part of the theoretical framework for event-traffic correlation, we developed a causal model for event driven network traffic. In this model agents, events, and data components interact as actions are performed on a network. We furthered this model by introducing the notion of intentions which can be propagated across these components, eventually making their way to the network traffic. Adversaries prop-

agate malicious intentions, either actively by generating new traffic on the network, or passively by altering otherwise normal traffic. We feel that this model is powerful enough to describe a wide variety of network interactions, both benign and malicious.

Using our event-traffic model as a tool, we described a method for re-creating the causal chain between high-level events and network traffic. This method makes use of behavior and inference models for agents which allow us to generate expectations for future events based on past events, and infer previous events from observed network data.

In designing YAMA we translated our conceptual event model components into objects that could be implemented in an application. Agent behavior and inference models became modules, and data became a set of bits enclosed in virtual buffers, which keep not only the data but also a set of markings for the data. We then showed a method for matching expectations and inferences.

For the implementation of YAMA 1.0 our proof of concept for YAMA, we decided to focus on web traffic because of its ubiquity in enterprise networks. We implemented some low-level traffic modules, and high-level web browser and DNS modules.

In the evaluation for YAMA 1.0 we saw that we could achieve a very high (90%) coverage rate for general HTTP packets, and even higher coverage rates (99%) for HTTP GET requests and DNS packets, while generating no false positives. The coverage was impacted heavily by the accuracy of our web corpora, which YAMA 1.0 uses to determine the state of the web as it was when an experiment was run. Our success in marking web traffic indicates that marking for other traffic types is possible and should be attempted.

8.2 Future Work

YAMA 1.0 is a good first implementation for YAMA showing some promising results. In order to become more useful as part of a network test environment, it should be expanded in several ways. The following are refinements and suggested future projects.

Multiple Hosts Creating a host module to encapsulate the current set of modules, then connecting several of those together by use of a network Transport object would yield the capability for multiple hosts.

DNS-based TCP Connection Verification Using the DNS reply to validate outgoing requests in the browser module would provide resistance against a passive adversary who redirects user sessions. Such validation would occur at in the TCP acceptors used by the browser module.

Simulatable Functionality Keeping the local time in modules would allow YAMA modules to use timeouts. This would help eliminate stale acceptors and TCP connections left unclosed due to missing data. Time in modules should be updated using the time stamps in processed events and traffic. The time could be synchronized globally or kept by each module. The individual module approach would work better if YAMA were to be turned into a distributed system.

Deeper Network Service Modules The current modules do just enough to handle standard cases. The TCP module should be expanded to handle data retransmission. The IP module should handle packet reassembly.

DNS over TCP In rare cases, a DNS response is too large to fit in a UDP packet. The transaction is then done over TCP. This would require the addition of

a TCPConnection TrafficProcessor interface to the DNS module. This would attach a DNS parser to the streams, which would produce DNS message objects that could be processed as usual.

Additional Protocol Modules Web is a good start, but a lot of network activity also involves email and other services. Implementation of SMTP, POP, IMAP, and FTP modules and parsers to support high-level mail and file transfer application modules would be a good start.

Distributed YAMA YAMA 1.0 is a single-threaded application. When YAMA grows to include multiple hosts or networks, it would be prudent to allow subsets of modules to run in their own threads. Separation of each host or network into its own thread would be good. Flexibility can be achieved by implementing a module wrapper which runs a module in its own threads and has its own queue for events and traffic data. Such a wrapper could be used around modules of any type, from network service modules to entire networks. This would allow optimization for different types of experiments.

Future versions of YAMA will eventually be integrated into LARIAT, where it will hopefully ease the lives of future analysts.

Appendix A

Sites in the Alexa "Clean 98"

This appendix shows the "Alexa Clean 98." This is the subset of the Alexa Top 100 [1] used in the evaluations. The sites are listed in order of popularity as displayed by Alexa on July 12, 2006. Also shown in the table is the site type, and the number of associated traffic elements as observed in evaluation data set 2 on July 19, 2006.

Event URL	Type	HTTP Packets	HTTP Requests	DNS Packets	DNS Queries
www.yahoo.com	portal	131	34	10	5
www.google.com	simple search	14	3	2	1
www.msn.com	portal	252	50	26	13
www.myspace.com	social	78	16	18	9
www.ebay.com	commerce	403	49	16	8
www.passport.net	corporate	4	2	6	3
www.amazon.com	commerce	348	77	8	4
www.microsoft.com	corporate	170	33	22	11
www.google.co.uk	simple search	14	3	0	0
www.blogger.com	social	216	54	4	2
www.live.com	simple search	54	19	10	5
www.youtube.com	social	122	37	28	14
www.bbc.co.uk	news	195	51	6	3
www.go.com	portal	205	28	10	5
www.cnn.com	news	213	48	16	8
www.craigslist.org	social	13	4	4	2
www.alibaba.com	commerce	124	36	6	3
www.imdb.com	corporate	1065	69	16	8
www.aol.com	portal	201	51	17	8
www.orkut.com	social	2	1	4	2
www.ebay.co.uk	commerce	255	31	10	5

Event URL	Type	HTTP Packets	HTTP Requests	DNS Packets	DNS Queries
www.xanga.com	social	114	14	6	3
www.google.ca	simple search	14	3	2	1
www.geocities.com	corporate	97	26	16	8
www.friendster.com	social	2	1	2	1
www.flickr.com	social	23	7	4	2
www.comcast.net	portal	590	44	14	7
www.apple.com	corporate	201	40	10	5
www.facebook.com	social	31	8	4	2
72.14.203.104	simple search	22	6	0	0
www.hi5.com	social	300	51	26	13
www.nytimes.com	news	577	111	40	20
www.megaupload.com	news	209	50	10	5
www.about.com	portal	378	61	26	13
www.cnet.com	news	795	132	20	10
www.soso.com	simple search	45	14	4	2
www.rediff.com	news	208	58	26	13
www.weather.com	news	338	69	12	6
www.google.co.in	simple search	22	6	2	1
www.photobucket.com	social	1147	29	8	4
www.mapquest.com	simple search	169	46	16	8
www.imageshack.us	social	98	19	15	8
www.mediaplex.com	corporate	176	12	4	2
www.google.com.au	simple search	14	3	2	1
www.overture.com	corporate	173	58	12	6
www.statcounter.com	corporate	133	21	6	3
www.sourceforge.net	corporate	273	59	20	10
www.match.com	social	173	28	6	3
www.download.com	portal	273	77	18	9
www.starware.com	corporate	126	39	2	1
www.adobe.com	corporate	625	98	8	4
www.fotolog.net	social	113	31	36	18
www.bankofamerica.com	corporate	183	31	4	2
www.mywebsearch.com	simple search	20	8	6	3
www.earthlink.net	portal	260	59	14	7
www.webshots.com	social	528	59	30	15
www.nba.com	news	781	141	12	6
www.reference.com	corporate	119	23	22	11
www.hp.com	corporate	321	29	6	3
www.tripod.com	corporate	234	58	19	10
www.digg.com	social	311	70	5	2
www.typepad.com	social	186	35	14	7
www.archive.org	portal	772	26	6	3
www.nastydollars.com	commerce	31	8	2	1
www.myway.com	portal	77	15	20	10
www.amazon.co.uk	commerce	232	59	6	3
www.netflix.com	corporate	220	32	6	3
www.monster.com	corporate	137	31	14	7
66.249.93.104	simple search	22	6	0	0
www.mlb.com	news	681	125	20	10

Event URL	Type	HTTP Packets	HTTP Requests	DNS Packets	DNS Queries
www.ign.com	news	943	158	56	28
www.altavista.com	simple search	22	8	8	4
www.macromedia.com	corporate	15	6	4	2
www.aebn.net	corporate	93	14	4	2
www.dmoz.org	simple search	19	6	2	1
www.gmx.net	portal	240	67	12	6
www.digitalpoint.com	corporate	71	13	2	1
www.washingtonpost.com	news	553	97	24	12
www.ups.com	corporate	68	15	4	2
www.usps.com	corporate	137	37	4	2
www.godaddy.com	corporate	203	56	8	4
www.gamespot.com	news	1182	137	24	12
www.netscape.com	portal	113	9	4	2
www.expedia.com	corporate	442	84	6	3
www.target.com	commerce	530	110	6	3
www.ebay.com.au	commerce	249	38	10	5
www.msn.co.uk	portal	165	37	22	11
www.fatwallet.com	commerce	107	29	6	3
www.sitesell.com	commerce	226	44	7	3
www.iask.com	simple search	22	9	6	3
www.slashdot.org	news	245	58	14	7
www.lycos.com	portal	364	77	20	10
www.ibm.com	corporate	175	35	2	1
www.theplanet.com	corporate	268	66	2	1
del.icio.us	social	69	9	4	2
www.metacafe.com	social	265	52	18	9
www.deviantart.com	social	376	72	26	13
www.foxsports.com	news	515	176	92	46

Bibliography

[1] Alexa's Top 100 English Speaking Sites. *http://www.alexa.com/*.

[2] Ethereal: A Network Protocol Analyzer. *http://www.ethereal.com/*.

[3] GCC, the GNU Compiler Collection. *http://gcc.gnu.org/*.

[4] The Boost C++ Libraries. *http://www.boost.org/*.

[5] The DETER Testbed: Overview. http://www.isi.edu/deter/docs/testbed.overview.pdf, August 2004.

[6] Kamran Ahsan and Deepa Kundur. Practical Data Hiding in TCP/IP. October 15 2002.

[7] S. Bellovin. The Security Flag in the IPv4 Header. RFC 3514, 1 April 2003.

[8] Lee Breslau, Deborah Estrin, Kevin R. Fall, Sally Floyd, John S. Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in Network Simulation. *IEEE Computer*, 33(5):59–67, 2000.

[9] CERT. CERT Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks. *http://www.cert.org/advisories/CA-1996-21.html*, 1996.

[10] CERT. CERT Advisory CA-1999-04 Melissa Macro Virus. 1999.

[11] CERT. CERT Advisory CA-2000-04 Love Letter. *http://www.cert.org/advisories/CA-2000-04.html*, 2000.

[12] Stefan Savage Colleen Shannon Stuart Staniford David Moore, Vern Paxson and Nicholas Weaver. Inside the Slammer Worm. *IEEE Security & Privacy*, http://www.cert.org/advisories/CA-1999-04.html:33–39, 2003.

[13] D. E. Denning. An Intrusion-Detection Model. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy (SSP '86)*, pages 118–133, Los Angeles, Ca., USA, April 1990. IEEE Computer Society Press.

[14] David Dittrich. Demonstration: Session hijacking. *http://staff.washington.edu/dittrich/talks/qsm-sec/hijack.html*, 1998.

[15] Mario Gerla, Lokesh Bajaj, Mineo Takai, Rajat Ahuja, and Rajive Bagrodia. GloMoSim: A Scalable Network Simulation Environment. Technical Report 990027, University of California, Los Angeles, Computer Science Department, May 13, 1999.

[16] IEEE. *802.3 Carrier Sense Mulitple Access with Collision Detection (CSMA/CD)*. IEEEP, 1985.

[17] Van Jacobson, Craig Leres, and Steven McCanne. *TCPDUMP(1), BPF...*, 1990.

[18] Butler W. Lampson. A Note on the Confinement Problem. *Commun. ACM*, 16(10):613–615, 1973.

[19] Elias Levy. The Making of a Spam Zombie Army: Dissecting the Sobig Worms. *IEEE Security & Privacy*, 1(4):58–59, 2003.

[20] Richard Lippmann, David Fried, Isaac Graf, Joshua Haines, Kristopher Kendall, David McClung, Dan Weber, Seth Webster, Dan Wyschogrod, Robert Cunningham, and Marc Zissman. Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Los Alamitos, CA, 2000. IEEE Computer Society Press.

[21] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks*, 34(4):579–595, 2000.

[22] David Llamas, Alan Miller, and Colin Allison. An Evaluation Framework for the Analysis of Covert Channels in the TCP/IP Protocol Suite. In *ECIW*, pages 205–214. Academic Conferences Limited, Reading, UK, 2005.

[23] Misra. Distributed Discrete-Event Simulation. *CSURV: Computing Surveys*, 18, 1986.

[24] P. Mockapetris. Domain Names - Implementation and Specification. RFC 1035, November 1987.

[25] Madanlal Musuvathi and Dawson R. Engler. Model Checking Large Network Protocol Implementations. In *NSDI*, pages 155–168. USENIX, 2004.

[26] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Comput. Networks*, 31(23-24):2435–2463, 1999.

[27] Jon Postel. Internet Protocol. RFC 791, ISI, September 1981.

[28] Michael Zink Carsten Griwodz Ralf Ackermann, Utz Roedig and Ralf Steinmetz. Associating Network Flows with User and Application Information. *Proceedings of the 2000 ACM workshops on Multimedia*, 2000.

[29] Lee M. Rossey, Robert K. Cunningham, David J. Fried, Jesse C. Rabek, Joshua W. Haines, and Marc A. Zissman. LARIAT: Lincoln Adaptable Real-time Information Assurance Testbed, May 15 2001.

[30] Craig H. Rowland. Covert Channels in the TCP/IP Protocol Suite. *First Monday*, 2(5), 1997.

[31] Stephen Schwab, Brett Wilson, and Roshan Thomas. Methodologies and Metrics for the Testing and Analysis of Distributed Denial of Service Attacks and Defenses. In *MILCOM 2005*, 2005.

[32] Tim Shimeall (SEI). Personal communication about data collected from a large enterprise. 2005.

[33] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium (SECURITY-02)*, pages 149–170, Berkeley, CA, USA, August 5–9 2002. USENIX Association.

[34] Ken Thompson. Reflections on Trusting Trust. *Commun. ACM*, 27(8):761–763, 1984.

[35] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A Taxonomy of Computer Worms. In *Proceedings of the 2003 ACM Workshop on Rapid Malcode (WORM-03)*, pages 11–18, New York, October 27 2003. ACM Press.

7738 . 98