A Tool for Statistical Detection of Faults in Internet Protocol Networks

Jonathan Roberts, Sandro Fouche, and James Purtilo
E-mail: {jroberts, sandro, purtilo}@cs.umd.edu

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

While the number and variety of hazards to computer security have increased at an alarming rate, the proliferation of tools to combat this threat has not grown proportionally. Similarly, most tools currently rely on human intervention to recognize and diagnose new threats. We propose a general framework for identifying hazardous computer transactions by analyzing key metrics in network transactions. While a thorough determination of the particular traits to track would be a product of the research, we hypothesize that some or all of the following variables would yield high correlations with certain undesirable network transactions:

Source Address
Destination Address/Port
Packet Size (overall, header, payload)
Packet Rate (overall, Source, Destination, Source/Destination)
Transaction Frequency (per Address)

By examining statistical correlations between these variables we hope to be able to distinguish – and normalize for changes over time – a healthy network from one that is being attacked or performing an attack.

Central to this research is that the class information we are analyzing is available without intervention on the participants of the network transactions, and, in reality, can be performed without their knowledge. This characteristic has the potential to allow Internet service providers or corporations the ability to identify threats without large-scale deployment of some kind of intrusion detection mechanism on each system. Furthermore combining the ability to identify existence and source of a network threat with common network hardware automatic configuration abilities allows for rapid reaction to attacks by shutting down connectivity to the originators of the exploit.

This paper will detail the design of a set of tools – dubbed Culebra – capable of remotely diagnosing troubled networks. We will then simulate an attack on a network to gauge the effectiveness Culebra. Ultimately, the type of data gathered by these tools can be used to develop a database of attack patterns, which, in turn, could be used to proactively prevent assaults on networks from remote locations.

Table of Contents

1. Introduction 1
   1.1 Motive 1
   1.2 Methodology 1
   1.3 The Development of Culebra 2
   1.4 Network Model 2

2. The Cclient Module 4
   2.1 General Description 4
   2.2 Technical Overview 5
      2.2.1 Storage of Tool Information 5
      2.2.2 Communication with Tools 5
   2.3 Risks 5
   2.4 Extension 5

3. The Ccomm Module 6
   3.1 Description 6
   3.2 Technical Overview 6
      3.2.1 The Cbuff Component 6
      3.2.2 The Cmsg Component 6
      3.2.3 The Ccomm Top-Level Component 7
      3.2.4 The Culebra Message Passing Protocol 7
      3.2.5 Using the Ccomm Module 8
   3.3 Risks 8
   3.4 Extension 8

4. The Cstat Module 9
   4.1 Description 9
      4.1.1 Options 9
      4.1.2 Single-Shot Mode 10
      4.1.3 Bucket Mode 10
      4.1.4 Reporting for Bucket Mode 11
   4.2 Technical Overview 11
      4.2.1 Phase One: Packet Capture and Data Extraction 12
      4.2.2 Phase Two: Calculation of the Statistics 13
      4.2.3 Phase Three: Analysis of the Statistics and Abnormality Detection 13
      4.2.4 Phase Four: Reporting of Abnormalities 13
      4.2.5 A Closer Look at Single-Shot Mode 13
      4.2.6 A Closer Look at Bucket Mode 14
   4.3 Risks 15
   4.4 Extension 15

5. The Cdummyd Module 16
   5.1 Description 16
      5.1.1 Options 16
      5.1.2 File Types and Formats 17

5.1.3 Example Files 17
5.2 Technical Overview 18
5.2.1 The Main Parent Process 18
5.2.2 The Main Child Process 18
5.2.3 The Dummy Server Thread 20
5.2.4 The Client Handler Thread 20
5.2.5 Portability 21
5.3 Risks 21
5.3.1 Limitations 21
5.3.2 Removal of the Port Scan Detector 22
5.3.3 Known Bugs 22
5.4 Extensions 22
5.4.1 OS-Specific TCP Stack Emulation 22
5.4.2 Configurable Send/Receive Patterns 23
5.4.3 More Complex Behavior 23
5.4.4 Threat Analysis and Countermeasures 23

6. Simulation 24
6.1 Procedure 24
6.2 Testing Results 24
6.2.1 Scenario A 24
6.2.2 Scenario B 25
6.2.3 Scenario C 26
6.3 Statistical Analysis 27

7. Summary 29
7.1 Simulation Conclusions 29
7.2 Open Issues 29

1. Introduction

1.1 Motive

With the constantly increasing number of corporations and homes able to access the Internet, especially at high speeds, the need to detect and mitigate computer attacks increases proportionally. Our proposed research examines the behavior of multiple network-based exploits to automatically identify and prevent new exploits by recognizing network attacks through statistical means. Furthermore, while many current systems that recognize computer exploits run on the target computer, we propose to perform analysis from network-based tools residing on systems not directly affected by the attack.

Today there are an increasing number of hazards to computer security, including denial of service attacks, computer viruses, and malicious network worms. While the number and variety of these attacks have increased at an alarming rate, the proliferation of tools to combat this threat has not grown proportionally. Similarly, most tools currently rely on human intervention to recognize and diagnose new threats. We propose a general framework for identifying hazardous computer transactions by analyzing key metrics in network transactions. While a thorough determination of the particular traits to track would be a product of the research, we hypothesize that some or all of the following variables would yield high correlations with certain undesirable network transactions:

Source Address
Destination Address/Port
Packet Size (overall, header, payload)
Packet Rate (overall, Source, Destination, Source/Destination)
Transaction Frequency (per Address)

By examining statistical correlations between these variables we hope to be able to distinguish – and normalize for changes over time – a healthy network from one that is being attacked or performing an attack.

Central to this research is that the class information we are analyzing is available without intervention on the participants of the network transactions, and, in reality, can be performed without their knowledge. This characteristic has the potential to allow Internet service providers or corporations the ability to identify threats without large-scale deployment of some kind of intrusion detection mechanism on each system. Furthermore combining the ability to identify existence and source of a network threat with common network hardware automatic configuration abilities allows for rapid reaction to attacks by shutting down connectivity to the originators of the exploit.

1.2 Methodology

We propose a simple methodology for mitigating cyber-terrorist assaults. First, we will collect data from headers of packets over the network. Second, we will calculate statistics on this data, and analyze the statistics to establish proper baselines of operation. Finally, we will use statistical measures to detect abnormalities or deviances from the baselines that might signal attacks.

We hypothesize that by calculating statistics on the network packet fields there will be noticeable differences between normal network traffic statistics and statistics calculated during a network attack. These differences serve as the basis for the development of software that detects and responds to network anomalies and attacks.

The programs that we developed have been designed to detect large network anomalies using statistics collected on packet headers. For the purposes of this paper, we collected a subset of the variables listed above in order to detect large network anomalies as a proof-of-concept – that is, to of see if the proposal was accurate and worth continuing in the future. With the data collected, we determined that the proposal was indeed worthwhile and created a solid foundation upon which to base future works.

1.3 The Development of Culebra

The primary purpose of the Culebra suite is to assist in identification of attacks made on a network using statistical analysis of network traffic. Results of this analysis are reported to Culebra Clients.

Ideally, attacks will be targeted at the Cdummyd program. Cdummyd appears as a vast array of vulnerable machines to draw attacks away from functional networks.

The project consists of three components:

- ♣ Cstat – A program that monitors the network and reports on statistical anomalies. The Cstat Module is described in Section 4.
- ♣ Cclient – A program that displays information detected by the Cstat program. The Cclient Module is described in Section 2.
- ♣ Cdummyd – A program that masquerades as a suite of vulnerable machines to draw attacks. The Cclient Module is described in Section 5.

Each component plays a role in identifying network activity patterns that are suspected as threats to network security and alerting the user to the presence of these threats.

The Culebra tools are designed to help those looking to identify acts of cyber-terrorism in a network environment, particularly in an educational environment. Thus, Culebra is intended for students, professors, future developers, and those that study network security.

1.4 Network Model

The suite of Culebra tools operates according to the following network model:

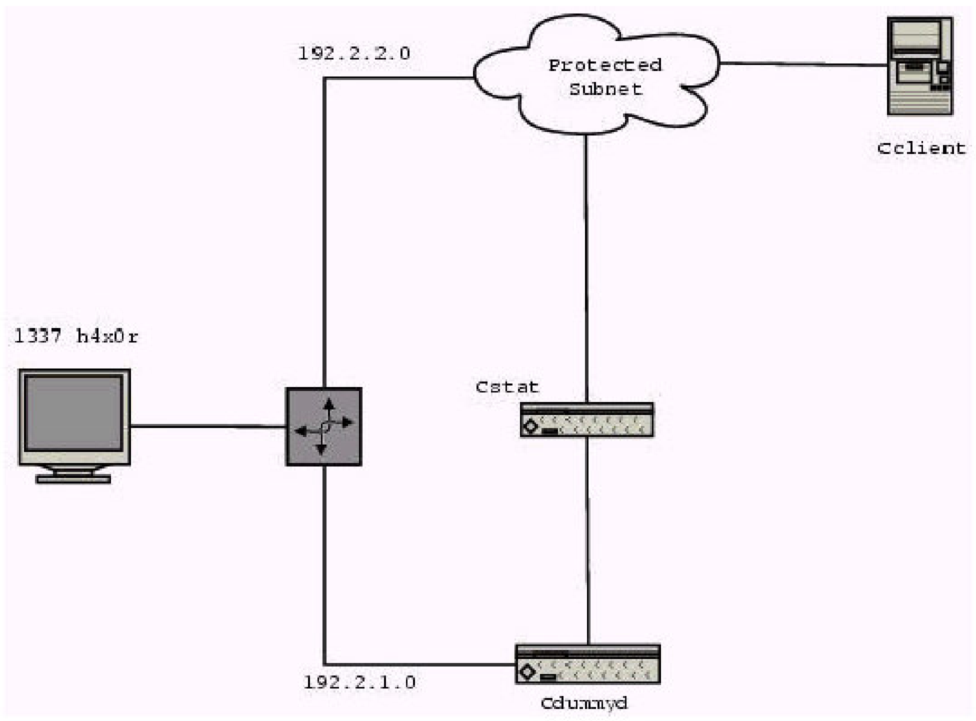

Figure 1.1 – The ideal network model used in designing the Culebra set of tools. Lines indicate direct network connections.

Figure 1.1 illustrates the ideal setup for the Culebra suite of tools. A router will separate the Cdummyd array of false machines from the working subnet. There will be a one-way connection to the Cstat network monitor, which will monitor the Cdummyd array for unusual network activity. The Cstat network monitor will then push updates to the Cclient machine as requested.

2. The Cclient Module

2.1 General Description

The Cclient Module is a graphical user interface (GUI) written in Java that accepts data transmissions from the Cstat module and any other custom user tools that adhere to the Culebra Message Passing Protocol specified in Section 3.2.4. Through the GUI, the user of the suite of tools will be able to analyze the data from each individual tool in order to detect security intrusions. Furthermore, the addition of tools will be handled in a plug-and-play fashion, requiring no coding changes. This will allow for expandability and future survivability of the suite of programs.

The GUI consists of a splash screen, a license agreement screen, a main application screen, a monitor screen, an add tool screen, an about screen and a help screen.

The splash screen consists of our company logo and name along with a progress bar that indicates the loading progress of the application.

The license agreement screen shows the license agreement for our software suite, allowing the user to accept the agreement and continue, or reject the agreement that will close the program.

The main application screen is the heart of the graphical user interface. From the main application screen, the user will be able to access the About Screen, the Help Screen, and the individual tools. The individual tools are brought up in a monitor window by double clicking the row that represents the tool in the table. Note: Tool representation was prepared using JTable.

The Tool Monitor Screen allows users to view the data that specific tools are sending back to the client for analysis.

The About Screen lists the copyright and owner information of the program.

The Help Screen provides users with information about how to use the program, the purpose of the program, and who to contact in case of a problem.

The Add Tool Window allows users to add tools to the list of monitoring agents. You can access this window by selecting Add Tool from the Edit window, or by clicking on the Add Tool Button. See Section 2.2.1 for information on how tool info is stored in a flat file.

To delete an item from the list of tools, the user must select “Delete” from the Edit window. Note: A tool must be deleted in the GUI as well as deleted from the file it is stored in. See section 2.2.1 for more information on this file.

2.2 Technical Overview

The GUI is designed to communicate with the tools through socket connections. The GUI will initiate a connection to a tool and send it a "Get" command. The tool will, in turn, send back its data in quasi-XML format to be displayed by the GUI in a scrolling window. See Section 3.2.4 for more information on the quasi-XML format being used. Tool information is stored in a flat file; see Section 2.2.1 for more information.

2.2.1 Storage of Tool Information

Input from the Add Tool Window is stored in a semicolon-delimited flat file. The format of the file is as follows:

```
<ToolName>;<IPaddress>;<Port>
<ToolName2>;<IPaddress>;<Port>
```

This file is used to load the list of tools at program startup.

2.2.2 Communication with Tools

For information on how the Culebra Client communicates with tools see Section 3.2.4 on the Culebra Message Passing Protocol.

2.3 Risks

Some of the risks involved with using Java and socket connections include the fact that Sun might change its Java Development Kit and deprecate some methods in its inner classes, therefore causing code changes to be required. Socket communications also run the risk of not working if the user is behind a firewall, or behind an improperly configured firewall. Because the file that stores tool information is semicolon-delimited, entering a tool name that contains a semicolon will most likely cause errors with communication.

2.4 Extension

The use of Java, XML, and socket connections allows room for future expansion of the program. Java is a very extensible language, and the way in which the classes were designed and written allows the GUI to be easily modified and expanded to meet new requirements. New items can be added to XML as needed for future updates. Socket connections are supported by almost all programming languages and therefore offer a generic method of communication between the GUI and any existing and future tools.

3. The Ccomm Module

3.1 Description

Ccomm is the module that allows a Culebra tool to make its data available to Culebra Client modules. The Ccomm module accepts data from a tool and relays it to requesting client modules through TCP/IP network connections. Prospective tools using the Ccomm Module will be compatible with Culebra Client modules and therefore conform to the Culebra Communication Framework.

3.2 Technical Overview

The Ccomm Module consists of three components: Cbuff, Cmsg, and the top-level Ccomm. The Cbuff component is a timed buffer for storing data from the tool. The Cmsg component provides a few functions for creating and manipulating messages. The top-level Ccomm component is a pthread that listens for client requests and relays the tool's data using the other two components.

3.2.1 The Cbuff Component

A Cbuff buffer only retains reports added to it for a fixed amount of time. After this fixed amount of time has passed the reports are deleted. This is so that requesting clients only receive recent data rather than copious amounts of potentially stale data.

All reports are given timestamps when they are added to the buffer. The buffer is simply a queue with reports added at the tail. Each buffer has a time limit that is specified when it is created. The timed expiration is implemented by scanning the queue from the head and removing any reports that have been in the buffer longer than the time limit whenever the buffer is accessed by either `cbuff_addreport()` or `cbuff_getreports()`. The amount of time that a report has been in the buffer is easily calculated from the current time minus the timestamp of the report.

The actual reports are strings (character pointers) that are expected to be dynamically allocated by the calling program. They are freed when they expire.

Access to a Cbuff buffer is guarded by `pthread_mutex`'s to provide mutual exclusion in order to protect shared data.

3.2.2 The Cmsg Component

The Cmsg component consists of three functions for creating and manipulating Culebra messages. For information on the format of Culebra messages see section 3.2.4. The `cmsg_out_create()` function builds an outgoing message. The `cmsg_in_validate()` function validates the key of an incoming message by comparing to the key found in the file named by the keyfile argument. If the keys match then the message is valid. The `cmsg_in_getcmd()` function creates a copy of the command field of an incoming message.

3.2.3 The Ccomm Top-Level Component

The Ccomm top-level component consists of the `ccomm_unicate()` thread function. This function listens for requests from Culebra clients and responds with all recent reports. The arguments to the function are the name of the tool using the module, the name of the file with the secret key, the special code, the Cbuff buffer that holds the reports for the tool, and the port on which to listen for client requests.

3.2.4 The Culebra Message Passing Protocol

The Culebra Message Passing Protocol is a very simple request-reply scheme. It resides on top of TCP in the protocol stack. TCP is used to guarantee reliable, in-order transmission of messages. The messages consist entirely of string data in a quasi-XML format.

The requests from the Culebra appear in the following format:

```
<Client>
      <Msg>secret_key</Msg>
      <Message>actual_message</Message>
</Client>
```

The *Msg* field contains a secret key string, which is contained on a file at both the client and the tool. This is meant to provide authentication in a future security extension (see Section 3.4). The *Message* field contains the actual message string.

The replies appear in the following form:

```
<tool_name>
      <Msg>secret_key</Msg>
      <Code>some_code</Code>
      <Timestamp>time</Timestamp>
      <Data>actual_reply</Data>
</tool_name>
```

The container tag specifies the name of the tool that is issuing the reply. The *Msg* field contains a secret key string as above. The *Code* field is unused at this time and therefore has no purpose. The *Timestamp* field contains the time at which the reply is generated. The *Data* field contains the actual reply string.

The current scheme uses only one request and one reply. The request contains the secret key and the string "Get" in the *Message* field. The function of the "Get" request is to ask for all recent reports from a tool. It is sent by the client every XXX seconds to poll a tool for reports. The reply contains the secret key in the *Msg* field, a temporarily meaningless code that is always "1" in the *Code* field, timestamp in the *Timestamp* field, and all recent reports in the *Data* field. Thus the entire interaction between the client and a tool is the client to tool "Get" request followed by the tool to client reply with all recent reports, repeated every XXX seconds.

3.2.5 Using the Ccomm Module

Using the Ccomm Module to create a new tool is simple and intuitive:

- ♣ Include the ccomm.h header file.
- ♣ Create a Cbuff buffer.
- ♣ Fill a `ccomm_args` struct with the appropriate arguments (see Section 3.2.4), including the Cbuff buffer.
- ♣ Start the `ccomm_unicate()` function as a pthread with the `pthread_create()` function (see *man pthread_creat*e), giving it the `ccomm_args` struct as its argument.

This procedure starts the module running as a thread. To make reports available to any requesting clients, simply add reports to the Cbuff buffer using the `cbuff_addreport()` function.

3.3 Risks

The Culebra Message Passing Protocol is completely insecure. Anyone intercepting a message, such as by sniffing the network, can get the key and masquerade as a tool or client. The key is merely meant to allow for security as an extension (see Section 3.4).

3.4 Extension

The Culebra Message Passing Protocol could be extended in two ways:

- ♣ Changing the message format. For example, more fields and information could be added to the current XML-like format.
- ♣ Changing the actual protocol. For example, adding more commands to the current protocol or implementing a more complex protocol than the current simple request-reply.

Both of these enhancements would require changes to the Cmsg and Ccomm components, as well as to the Culebra Client.

The security of Culebra communication could be improved by encrypting all messages. The encryption itself would provide confidentiality. The *Msg* secret key field already in the messages would provide authenticity. The entire purpose of the field is to allow for this particular extension.

4. The Cstat Module

4.1 Description

```
cstat [-s <ss_duration>] [-i <interval>] [-b <buckets>]
      [-c <bcycles>] [-p <port>] [-f <filter>] [-q quiet]
```

The Culebra Statistics Module (Cstat) collects data from packet traffic on a network, calculates statistics from the data, detects abnormalities in the statistics, and reports these abnormalities to any number of Culebra clients. Cstat has two modes of operation: single-shot mode and bucket mode. Cstat defaults to single-shot mode. Cstat is designed strictly for use with Ethernet network interfaces. Use of Cstat with any other interfaces results in undefined behavior.

4.1.1 Options

`-s <ss_duration>`	Specifies the time length of a single-shot run in seconds. An ss_duration of zero causes single-shot to run indefinitely until the user gives ^C (Control-C). The default `ss_duration` is zero.
`-i <interval>`	Specifies the time length of the sampling period for bucket mode in seconds. An interval of zero forces single-shot mode and supercedes the `-b` and `-c` options. The default interval is zero.
`-b <buckets>`	Specifies the number of baseline buckets for bucket mode. The default number of buckets is one.
`-c <bcycles>`	Specifies the number of baseline cycles for bucket mode. The default number is of cycles is one.
`-p <port>`	Specifies the port on which to listen for report requests from Culebra Clients. This argument is only meaningful for bucket mode as single-shot mode does not generate reports and thus does not listen for clients. The default port is 30000.
`-f <filter>`	Specifies the `libpcap` filter string to use for filtering all packets captured. For syntax, see *man tcpdum*p. The default filter string is "`ip proto \tcp or \udp or \icmp`" which captures only packets of the TCP, UDP, and ICMP protocols. The Cstat tool is designed for these three protocols, but it will provide IP layer statistics for any protocols running on top of IP. Changing the filter to capture packets from protocols not running on top of IP will cause undefined behavior.
`-q quiet`	Tells Cstat to be quiet. The program will not display any statistics. It will only display its starting message and reports generated by abnormal statistics in bucket mode.

4.1.2 Single-Shot Mode

In single-shot mode, packet data is collected continuously until the user gives ^C (Control-C) or, if specified by the `-s` option, the single-shot duration expires. Statistics are then calculated on the data and displayed to screen, and the program terminates.

4.1.3 Bucket Mode

In bucket mode, Cstat runs indefinitely. It first builds a statistical baseline and then monitors statistics relative to the baseline. Statistics that are abnormal relative to the baseline, as described in Section 4.1.4, are reported to any requesting Culebra clients. Statistics that are normal relative to the baseline are incorporated into the baseline. Giving ^C (Control-C) at any time causes the program to clean up and terminate.

The basic unit of bucket mode is the sampling period. The time length of the sampling period, in seconds, is specified by the `-i` option. A bucket is a segment of the baseline that is equivalent to one sampling period. The number of buckets in the entire baseline is specified by the `-b` option. After the baseline is established each subsequent sampling period's statistics are compared to the statistics in the corresponding baseline bucket. For example with:

```
cstat -i 3600 -b 24
```

The baseline is established in the first 24 sampling periods (of one hour each). Thus the baseline consists of 24 one-hour buckets. The subsequent sampling periods are compared back to the corresponding buckets, thus:

(◊ means *compared to*)
25th, 49th, … sampling periods ◊ 1st bucket
26th, 50th, … sampling periods ◊ 2nd bucket
…
48th, 72nd, … sampling periods ◊ 24th bucket

The `-c` option specifies how many cycles of the buckets to average when establishing the baseline. Thus the first *bcycles* × *buckets* sampling periods will be used for baseline establishment.

In the above example, adding the option `-c 2` will result in the first 48 sampling periods (hours) being used for the baseline. The statistics from the hours of the first day will be averaged evenly with those of the corresponding hours on the second day to produce one baseline of 24 buckets (hours). Starting on the third day, sampling periods will be compared back to the corresponding buckets, thus:

49th, 73rd, … sampling periods ◊ 1st bucket
50th, 74th, … sampling periods ◊ 2nd bucket

…
72nd, 96th, …sampling periods ◊ 24th bucket

4.1.4 Reporting for Bucket Mode

When abnormal statistics are detected, a report is relayed to any requesting Culebra Clients. Cstat listens for client requests on the port specified by the `-p` option. If the `-p` option is not given, the default port is 30000.

The detection of abnormal statistics is computed by comparing the statistics of each sampling period to those of the corresponding baseline bucket. A statistic is considered abnormal if it is more than two baseline bucket standard deviations above or below the baseline bucket mean. The statistics that are monitored are:

`mean_saddr`	=	mean IP source address
`mean_daddr`	=	mean IP destination address
`mean_iphdr_len`	=	mean IP header length
`mean_total_len`	=	mean IP total packet length
`mean_sport`	=	mean TCP/UDP source port
`mean_dport`	=	mean TCP/UDP destination port
`mean_ttl`	=	mean IP Time-To-Live
`mean_tcpflags`	=	mean TCP flags
`mean_tcpseq`	=	mean TCP sequence number
`mean_tcpwin`	=	mean TCP advertised window
`mean_tcpurg`	=	mean TCP urgent pointer

If and only if all statistics are normal, they are averaged into the corresponding baseline bucket. Otherwise, a report is generated and each abnormal statistic, and its direction of abnormality, is added to the report. Reports consist of the string "warning! ", followed by the statistics and their directions, followed by the string " at ", followed by a timestamp marking the report generation time. The symbols '+' and '-' are appended to the end of statistic names to indicate abnormally high and abnormally low, respectively.

Thus, for example, a report of high mean IP source address and low mean IP Time-To-Live would look like:

```
warning! mean_saddr+ mean_ttl- at <timestamp>
```

4.2 Technical Overview

The Cstat tool is written in C and targeted for a variety of UNIX and UNIX-like systems including Linux, BSD, and Solaris. The operation of the Cstat tool consists of four phases:

1) Packet capture and extraction of data from the packet headers.
2) Calculation of statistics on this data.

3) Analysis of these statistics to detect abnormalities.
4) Reporting of detected abnormalities to any requesting Culebra clients.

The single-shot mode of operation executes the first and second phases once, prints the statistics calculated in the second phase, and exits.

The bucket mode of operation executes all four phases continuously.

4.2.1 Phase One: Packet Capture and Data Extraction

Cstat uses the *libpcap* packet capture library to get its packets. *libpcap* is used under the BSD license. *libpcap* is a portable library that is known to work on a wide variety of UNIX and UNIX-like systems including Linux, BSD, and Solaris. Cstat uses version 0.7.1 of *libpcap* because it is the most recent version as of this time. For further information on *libpcap*, see http://www.tcpdump.org/ or, on any machine with *libpcap* already installed, see *man pca*p.

In the Cstat module, *libpcap* serves to continuously capture packets and pass them to the `process_packet()` function for extraction of the data fields from the headers. By using *libpcap* to set the underlying Ethernet network interface into promiscuous mode and create packet filters, Cstat captures *all* ICMP, TCP, and UDP packets on the network. From these packets the following fields are extracted and used for data:

IP source address	(all packets)
IP destination address	(all packets)
IP header length	(all packets)
IP total packet length	(all packets)
IP time-to-live	(all packets)
Source port	(TCP & UDP only)
Destination port	(TCP & UDP only)
TCP flags	(TCP only)
TCP sequence number	(TCP only)
TCP advertised window	(TCP only)
TCP urgent pointer	(TCP only)

See RFC's 0768, 0791, 0792, and 0793 for more information on these protocol header fields.

Each time a packet is captured the applicable fields, depending on the packet protocol, are extracted into a structure, which is copied into a `cstat_buff` buffer. This buffer is the input for the second phase and is guarded by a `pthread_mutex` because of multi-threading in the bucket mode of operation.

A major shortcoming of using *libpcap* is that it does not reassemble fragmented IP packets. Thus if a packet is fragmented into twenty pieces its data fields could be counted twenty times. Cstat avoids this problem by ignoring all packets with the MF flag (more fragments flag) set. All of the fragments of a fragmented IP packet except the last one must have the MF flag set. Thus, Cstat ignores all but the last fragment and the data fields for a fragmented packet are only counted once. This creates an error in the collection of IP total packet length field. For a description of this error, see the first bulleted item of Section 4.3.

4.2.2 Phase Two: Calculation of the Statistics

The second phase receives its data from the `cstat_buff` buffer that is filled by the first phase. Getting data from the buffer with the `cstat_buff_get()` function returns the entire buffer and initializes a new, empty buffer in its place. This serves to create a precise point for ending sampling periods in the multi-threaded bucket mode of operation.

After the data is extracted from the buffer it is passed to the `calc()` subroutine for calculation of all statistics. All calculations are done with double precision for high accuracy and because the header fields, such as IP addresses, can be very large. The statistics calculated are the means, variances, and standard deviations of all of the extracted header fields. Also the number of packets – total and of each protocol – is counted.

4.2.3 Phase Three: Analysis of the Statistics and Abnormality Detection

Statistics are only analyzed in bucket mode. The analysis consists of comparing the statistics for the current sampling period to a corresponding baseline bucket. For details on how the baseline buckets are established and sampling periods correspond to them see section 4.1.3.

The only statistics that are compared are the means. This is because the program does not have measures such as standard deviation for comparing the other two statistic types, variance and standard deviation. In other words, the program does not have a standard deviation of the standard deviation.

The means from the current sampling period are compared to the means of the corresponding baseline bucket by a measure of two standard deviations from the baseline bucket. Thus, the current sampling period's means are considered abnormal if they are either two standard deviations above of below the baseline bucket means.

4.2.4 Phase Four: Reporting of Abnormalities

Abnormal statistics are reported to any requesting Culebra clients using the Culebra Communication Module (Ccomm). See Section 3 for all information on the Ccomm module. See Section 4.1.4 for a description of the reporting format.

4.2.5 A Closer Look at Single-Shot Mode

Single shot mode is a simple single-thread run of the first two phases. Packets are captured and data is extracted until either the user gives ^C (Control-C) or, if the `-s` option is specified, the duration expires. Statistics are then calculated on all of the data and displayed to screen. The packet capture duration is then displayed, and the program exits.

The duration is implemented by installing the `sigexit()` function as the handler for `SIGALRM` and setting an `alarm()` with the duration as the alarm time argument. The `sigexit()` function gets the data from the `cstat_buff` buffer, calls the `calc()` function, passing it the data, to calculate the statistics, cleans up, and exits. For more information about signals and alarm, see *man signal(2), signal(7), alarm(2).*

4.2.6 A Closer Look at Bucket Mode

Bucket mode consists of three threads running continuously. The first thread runs phase one. It continuously captures packets, extracts data, and fills the `cstat_buff` buffer with that data. The second thread runs phases two and three. It sleeps for the sampling period length to allow the buffer to fill with data. It awakens and extracts all data from the `cstat_buff` buffer, calculates statistics on the data, analyzes the statistics, and adds reports on abnormal statistics to the Cbuff buffer for the Ccomm thread to communicate to requesting clients. For more information on phases two and three, see Sections 4.2.2 and 4.2.3 respectively. For more information on the workings of the Ccomm Module see Section 3.

Thus the operation of bucket mode looks like this:

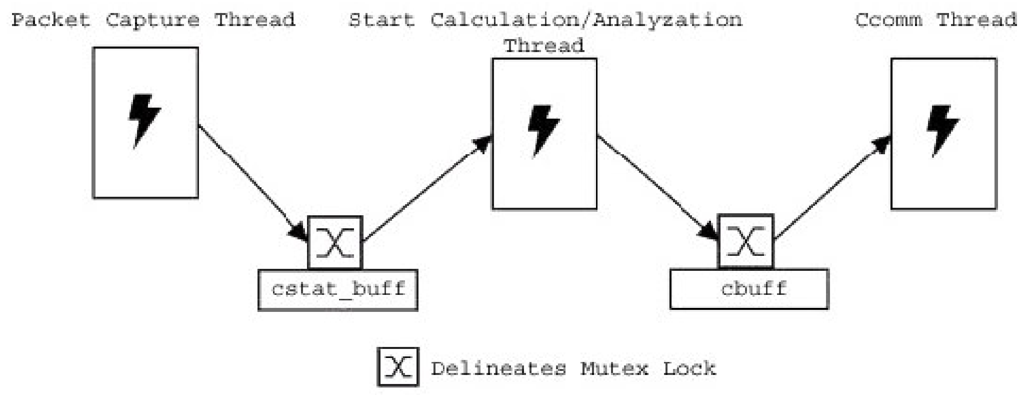

Figure 4.1 – A diagram of bucket mode operation

The three threads communicate through two buffers. Access to the buffers is protected by two `pthread_mutex`'s to ensure mutual exclusion.

It should be noted that the reading of the `cstat_buff` buffer by the second thread. It reads this buffer to get the data for calculation and analysis. While it is reading this buffer, the first thread cannot access this buffer and will block. Because the first thread performs the packet capture, it is very important that the duration of this block is minimized to avoid possible packet loss on busy systems. Thus, the reading of this buffer by the second thread has been optimized to be as efficient possible. For information on exactly what optimizations were made, see the source code and comments in the file `cstat_buff.c` for the `cstat_buff_get()` function (line 79).

4.3 Risks

There is an error in all statistics calculated on the IP total packet length header field when fragmented packets are present. This is because our work-around to the lack of fragment reassembly in *libpcap* is arguably flawed. Our work-around consists of ignoring all but the last fragment. Thus, this method retrieves only the length of the last fragment rather than that of the entire fragmented packet. The ways to correct this are:

- ♣ To implement a fragment reassembly algorithm or at least something to add up the total lengths over the fragments of a fragmented packet.
- ♣ To use an additional packet library, such as *libnid*s, that does fragment reassembly. See http://www.packetfactory.net/Projects/Libnids/.
- ♣ To simply disregard the IP total packet length field.

There is a high risk of this tool not being scalable to large, heavy traffic networks. It is probable that the *libpcap* packet capture functionality would scale; however, it is uncertain that the calculation of statistics would scale. There has been no way to test this, as there have been no large network resources available. If this problem exists, the only ways to remedy it are to change the statistical calculation methods or to improve the speed of the machine on which the program runs.

4.4 Extension

This tool could be extended by extracting more data fields (1) on which to calculate statistics, (2) from the individual packets, and (3) by calculating more statistics, such as covariance of all pairs on the data.

Making this tool configurable through a configuration file also could extend its functionality. There are several parameters in the `cstat_params.h` file that would be better specified in a changeable configuration file than a hard-coded header file. An example of this is the `RPTBUFF_TIME` parameter, which specifies how long to buffer abnormality reports before they expire.

Finally, this tool could be extended by adding another level of statistical calculation. Currently the tool only calculates statistics on fields from individual packets as data points within a sampling period. Adding another level of calculation would mean using statistics from entire sampling periods as data points. For example, the mean number of packets per sampling period could be calculated over five periods. Thus, statistics would not only be calculated for individual sampling periods but also on variable size aggregate groups of sampling periods.

5. The Cdummyd Module

5.1 Description

```
cdummyd [-h] [-V] [-b <-f <Batch File>>]
        [-s <-m <Response File>> <-p <port>> [-i <IPv4 Address>]]
```

The Culebra Target Dummy, or Cdummyd, allows its user to exaggerate the offering of TCP based services on their network. This is accomplished by creating simple server daemons, or "dummies," which are bound to a particular service's well -known port (see http://www.iana.org/assignments/port-numbers for a list of well-known ports). These dummies allow clients to connect and make a request, then respond with a message that is characteristic of the service that they are mimicking, after which they terminate the connection. The client may then be tricked into believing that the full-fledged service is running on the user' s network. Cdummyd's intended use is to mimic those exploitable services that are known to be frequent targets of network attacks. If the dummies are detected and subsequently attacked, then the attack should fail, due to the absence of the exploitable service's specific weakness. The user should also be alerted by another tool in the Culebra suite and thus be able to take the appropriate measures to secure their network.

The number of dummies that Cdummyd can create is limited only by the user's system resources. By using a network interface configuration program like *ifconfig* and carefully designing a batch file, a user may make it appear as if an entire network is running on a single computer.

5.1.1 Options

`-h`	The help flag; a brief description of the possible command line options is displayed. This is equivalent to running Cdummyd with no options.
`-V`	Version.
`-b`	Selects the Cdummyd batch mode. If this flag is set then the `-f` option must also be present.
`-f <Batch File>`	Specify a file containing the data to use in setting up Cdummyd in batch mode. See Section 5.1.2 for more details about a *<Batch File>*.
`-s`	Selects the Cdummyd single-server mode. If this flag is set then the `-m` and `-p` options must also be present; the `-i` option should usually be changed from its default as well.
`-m <Response`	Specify a file that contains the message to use when responding to

`File>`	clients in single-server mode. See Sections 5.1.2 and 5.4 for more information about a `<Response File>`.
`-p <Port>`	Specify a port to bind to in single-server mode. The range of valid port values is between 1 and 65355. The user must be root to bind to ports lower than `IPPORT_RESERVED`, which is technically system specific, but generally has the value 1024.
`-i <IPv4 Address>`	Specify an Internet address to bind to in single-server mode. The address must be in valid IPv4 format (i.e. dotted-quad notation.) The IP address defaults to 127.0.0.1 (i.e. the address of the loopback interface), and is thus not strictly a required argument.

Any incomplete or mismatched options will not be processed.

Cdummyd does not disallow the creation of dummies using both `-b` and `-s` in the same command, as long as all the options required by both modes are present.

5.1.2 File Types and Formats

Batch Files: The `<Batch File>` that is required for the `-f` option can have any name or file extension, but must consist of lines having the following format:

```
<IPv4 Address>:<Profile File>\n
```

(\n means new line character, not the string "\n")

Dummies are created as the batch and profile files are processed, and they are not terminated if an error occurs during later batch processing.

Profile Files: The `<Profile File>` is a file containing port and response file pairs. It can have any name or file extension, but it must consist of pairs of lines having the following format:

```
<Port>\n
<Response File>\n
```

`<Port>` values here are defined the same as those in the description of the -p option under section 8.1.1.

Response Files: The `<Response File>` is any file that the user wishes to have sent to those clients that make a request from a dummy server. This file is assumed to be encoded in ASCII. The user must provide or construct their own `<Response File>`' s.

5.1.3 Example Files

Some example files could be the lines between `#BEGIN ... SAMPLE` and `#END ... SAMPLE` below:

```
#BEGIN <Batch File> SAMPLE
192.168.0.5:sample.profile
192.168.0.17:test.profile
128.8.10.141:sample.profile
#END <Batch File> SAMPLE

#BEGIN sample.profile SAMPLE
80 HTTP.txt
8080 HTTP.txt
#END sample.profile SAMPLE

#BEGIN test.profile SAMPLE
21 FTP.txt
110 POP3.txt
#END test.profile SAMPLE
```

5.2 Technical Overview

The Culebra Target Dummy can be best explained by examining its constituent processes, as its underlying design model is one characterized by the transition of control over time, from older to younger processes. When Cdummyd runs, a minimum of two processes will always be executed during the setup and initialization of the module. Each dummy server that is successfully created exists as a thread and will create an additional thread to manage each connected client. The responsibilities of these processes are explained in detail within the next four subsections.

5.2.1 The Main Parent Process

This process does nothing other than fork off a duplicate of itself, then sleep for 100,000 microseconds before terminating, which places Cdummyd in daemon mode.

5.2.2 The Main Child Process

This process forks off from the original one, and is responsible for setting up the servers and their environment.

First, the child must install signal handlers to ignore the signals `SIGHUP`, `SIGINT`, and `SIGPIPE`. `SIGHUP` and `SIGINT`, the hang-up and interrupt signals, must be blocked to prevent casual termination of the Cdummyd daemon, as they are more likely to be generated by the user or system inappropriately. `SIGPIPE` is raised to signify a broken pipe, and normally terminates the program. Since we do not want Cdummyd to exit when its client misbehaves (e.g. a client aborting the connection while cdummyd is in a `recv()` call), we ignore `SIGPIPE` and check for the equivalent `EPIPE` error.

Next, a global file stream, which will be used by all the dummy servers, is opened in append mode, its output going to the file `cdummyd.log`. The global integer that keeps track of the number of active dummy servers is also initialized. All globals have an associated pthread mutex, which is a locking mechanism used by the pthread library to ensure thread-safe data access.

The child process then parses the user's command line options. The parsing routine that we use was created by the *gengetopt* program. The parsed options are checked to ensure that they constitute a valid request. If they do not, Cdummyd prints out the appropriate message explaining why and exits. If a valid request was made, then either the routine to start a single server or a batch of servers, or both routines will be called.

The routine to start a single dummy server begins by verifying the existence of the specified `<Response File>`. It also checks to ensure that `<Port>` is in the proper range, and that the user has root permissions if `<Port>` is below `IPPORT_RESERVED`. An address structure is then created using the command line input and bound to with a temporary TCP socket, to guarantee that the requested address and port combination is available on the user's network interface. If any of the above checks fail, the appropriate error message is printed and the routine returns `FALSE`. Otherwise, the pthread's argument structure is allocated, initialized, and passed to the dummy server's pthread as it is created. The pthread argument structure contains a string representation of the `<Response File>`'s name, the new pthread's id, and a s ocket address structure, to which the dummy's listening socket will be bound. The new pthread is created in detached mode, so that the child process does not have to be concerned with its resources later. Next, the number of active dummy servers is locked, incremented, and unlocked. The routine then returns `TRUE`.

The routine that starts dummy servers in batch mode begins by verifying the existence of, and then opening the `<Batch File>` specified by the user. The `<Batch File>` is parsed and operated on, one line at a time. After verifying the validity of `<IPv4 Address>` and the existence of `<Profile File>`, `<Profile File>` is opened, then parsed and operated on, two lines at a time. The same checks performed in the single server routine are now performed as the data is extracted from the files. If no errors are encountered, then a new pthread argument structure is allocated, initialized, and passed to the dummy server's pthread as it is created. The new pthread is created in detached mode, and the number of active dummy servers is locked, incremented, and unlocked. The routine then continues processing the `<Profile File>` until an `EOF` is reached. At that time, it begins to process the next line of the `<Batch File>`. This cycle continues until an `EOF` is reached in the `<Batch File>`. If an error occurs, or invalid data is encountered, at any time during the above process, then the routine stops cycling through the files, displays an appropriate error message, and returns `FALSE`. Should this happen, any dummy servers that were successfully created before the error was reached will persist and operate normally. If the entire `<Batch File>` is processed successfully then the routine returns `TRUE`.

If either of the above two routines returns a `TRUE` value, then the main child process is put into suspension until the last dummy server should happen to shut down. This is done using a pthread conditional wait call that is only signaled by the last dummy to exit. If no

TRUE values were returned, or one of the above two routines was not called, or the last dummy has awoken this process upon its exit, then the main child process exits, and Cdummyd is closed.

5.2.3 The Dummy Server Thread

The dummy server thread's first task is to install clean -up handlers to take care of any dynamic resources in the event that it terminates unexpectedly. This is necessary because of daemon nature of Cdummyd the thread has no typical termination sequence. It will only stop when it, or its parent thread, receives the SIGKILL signal. It is also always necessary to install an unlocking cleanup handler prior to locking a pthread mutex, and to pop the handler when the lock is released. This prevents other dummy servers from becoming deadlocked if a dummy were to die between locking and unlocking a mutex.

Next, the server socket is created, using the PF_INET, SOCK_STREAM, and IPPROTO_TCP settings. This socket will utilize a TCP data stream and require addresses from the Internet protocol family. This socket also needs the socket level option SO_REUSEADDR turned on, allowing it to continually accept connections by skirting the TCP TIME_WAIT state. The socket is then bound to the address that was passed into the thread within its thread argument structure, and set to listen for incoming connections. The connection limit is set to SOMAXCONN, which is the system specific maximum number of simultaneous connections.

The dummy server thread now enters an infinite loop. Upon entering the loop, it will block in an accept call until a client connects to its listening socket. The connecting client will be assigned to a client socket, which will also have its SO_REUSEADDR option set explicitly. Next, the thread will output the dummy server' s address and port, and the client's address and port, to the cdummyd.log. The file stream mutex must be locked before output and released afterwards. The client handler thread argument structure is then allocated and initialized; it contains a duplicate of the client socket, and a copy of the <Profile File>'s name. This structure is then passed to the client handler thread, which is now created. The new client handler thread is then detached, the client socket is closed, and the dummy server thread returns to the beginning of the loop.

When the dummy server thread dies for any reason, the following tasks must be performed in the functions that are popped from the stack of clean-up handlers. First, any thread-specific dynamic variables or resources must be released. The thread's argument structure is then de-allocated. Next, the global number of active dummy servers must be locked, decremented, and unlocked. If the current thread is the last active dummy server, then the global connection log file stream must be closed, and its mutex destroyed. Then, the thread will signal the waiting main child process, which will allow the entire Cdummyd program to terminate. If other dummy servers are still active, then the thread will just die.

5.2.4 The Client Handler Thread

First, the client handler thread opens `<Profile File>` with read-only access. It then blocks in `recv`. If the length of the message received from the client is less than zero, then an error has occurred. If the error value is not equal to `ECONNRESET`, `ENETRESET`, or `EPIPE`, then an error message is displayed, the thread's file and socket are closed, its resources are de-allocated, and it exits. If the error value is one of those above, then the same cleaning up is performed, but no error message is displayed.

Next, the thread rewinds `<Profile File>`, and enters a loop until the end of `<Profile File>` is reached. Inside the loop, one line is read from the file and placed into a character buffer. This buffer is then sent to the client. If an `EPIPE` occurs on send then the thread breaks from the loop; any other errors result in an error message being displayed and the thread exiting after performing the requisite cleaning. Otherwise, it continues to loop through `<Profile File>`.

After completing the loop, the thread closes `<Profile File>` and its socket, de-allocates its argument structure, and exits.

5.2.5 Portability

Neither Solaris nor DEC Alpha defines the required constant `MSG_NOSIGNAL`, which must be defined as a macro to be 0.

Solaris does not include the `inet_aton()` function. The FreeBSD code for `inet_aton()` was obtained, and it is compiled into Cdummyd only when the Makefile target 'solaris' is specified.

Solaris must also link the '`socket`', '`nsl`', and '`rt`' libraries during compilation, and have the `_POSIX_PTHREAD_SEMANTICS` macro defined.

5.3 Risks

5.3.1 Limitations

Many of the limitations to Cdummyd result from it being a proof-of-concept type product. The Culebra Target Dummy was developed in a time frame that simply did not allow the exploration of many features and behaviors that would be desirable in a network service mimicry tool. Some of these limitations are discussed below.

The biggest limitation to Cdummyd is its inability to fool a sophisticated OS detecting port scanner. An example of such a port scanner is Nmap, which is freely available at http://www.insecure.org/nmap. Nmap uses the signature quirks inherent in each distinct TCP stack implementation to identify its operating system. Because Cdummyd relies on its platform's TCP transport layer, there is no way to prevent its true identity from being discovered. However, a quick port scan with Nmap will still indicate that the fraudulent services are active. Indeed, operating system detection is not a default option, so perhaps the novice hacker will not select that mode. It is also plausible that an experienced intruder would be intrigued enough by the odd services that are running, and stop to investigate long enough to be detected and barred from further progress into the real network.

Another limitation is Cdummyd's reliance on the user to create an effective configuration. We were concerned about the copyright issues involved in capturing and distributing the actual messages of commercial software packages, which might belong to a proprietary network protocol. Thus, we left it to the user to collect or construct their own `<Response File>`'s.

Development of Cdummyd focused on mimicking Microsoft' s IIS web server. Currently, then, Cdummyd is only be effective at replicating the behavior of servers that expect to have a client send the first message once a connection is established.

5.3.2 Removal of the Port Scan Detector

Cdummyd was developed with the notion that it should exist on the same network as the rest of the tools but be incommunicado. Because Cdummyd is designed to attract attention, this would increase the overall security by preventing its messages to the clients from being traced, and thus giving away the location of the more important nodes. The port scan detector would have provided a direct means of remotely observing the activity of Cdummyd. Its deletion from the product leaves a void in this area and hinders the functional impact of Cdummyd by requiring more effort on the user's part to identify potential threats.

5.3.3 Known Bugs

The `cdummyd.log` file should contain a time and date for each connection, where `(NULL)` currently appears. This was originally just a misuse of the GNU C `ctime()` function, but both calling `ctime()` with its proper parameter and removing the `ctime()` call altogether caused some sockets to become unstable for reasons unknown at the time of this release. The testing of the log file seemed to slip under the radar, as it was unrelated to the networking functionality, which received the bulk of the attention. This was likely due to the failure of non-developers to test the day-to-day usability of the software.

5.4 Extensions

5.4.1 OS-Specific TCP Stack Emulation

The most useful, most exciting, and most complex extension to Cdummyd would be the addition of total TCP stack emulation to replicate an OS's unique behavior. This wa s so far outside the scope of our time restrictions that only a cursory investigation was able to be made into possible avenues of future development. The best method would seem to be the development of a generic, but completely tunable, TCP module that would incorporate all known eccentricities and allow them to be turned on or off at the user's discretion.

5.4.2 Configurable Send/Receive Patterns

Another desirable addition to Cdummyd would be the inclusion of a method to configure the pattern that a dummy server will follow when communicating with a newly connected client. This would allow more complicated protocols to be mimicked easily.

5.4.3 More Complex Behavior

Extending 5.4.2, each send in the pattern could be assigned a different message file, which would create a more realistic façade than just sending the same "Bad Request" message repeatedly. However, if this process continues indefinitely, then Cdummyd would become a sort of configuration-based utility server rather than just a mimic.

5.4.4 Threat Analysis and Countermeasures

The re-integration of the port scan detector would be the first step in identifying threats. Also, an analysis of the sorts of packets being received by the dummy servers should be a good indicator of the intentions of their sender. Hosts that are the persistent source of mangled packets, or packets with a variety of obscure options, would be flagged as at least snoops, and potential intruders. An interface with the network firewall or router could then automatically issue the order to block those hosts' access.

6. Simulation

6.1 Procedure

The first step of our experiment was to set up a network of computers. Our network consisted of three computers: two computers that with Microsoft Windows XP Professional installed, and one computer running Linux Mandrake 8.1. We recorded a baseline by gathering information and calculating statistics on "normal" network traffic. "Normal" network traffic is defined as traffic on a network that is not currently being attacked. The baseline was determined by running the Cstat module on the previously mentioned Linux machine. This module analyzed the communication between the two Windows machines. The communication consisted of constant 'normal' (non-flooding) pings, transferring Java Runtime Environment packages, watching a movie over the network, and/or two users playing a network game called, "Quake III v. 1.31," which uses the User Datagram Protocol (UDP). The Cstat tool reported the mean, variance, and standard deviation of source address IP, destination address IP, header length, total packet size, source port, destination port, IP TTL TCP flags, TCP sequence number, TCP advertise window, and TCP urgent pointer, and we recorded them. We continued to collect multiple baselines of network traffic, which provided us with a variety of baselines to compare to attack-based network statistics. Furthermore, we were able to observe if there were fairly consistent differences between the baseline and flood scenarios.

After recording the baseline information, we conducted the same tests, this time including a broadcast ping flood being sent over the network. We recorded the information relayed to us by the Cstat module and analyzed what network traffic looks like when a ping flood is being conducted against the network. We ran this test multiple times to verify that the information gathered is somewhat consistent, not an anomaly, and to match up the tests with the ones ran to gather baselines. The baseline statistics and the statistics gathered during the attack on the network were compared in attempt to draw any conclusions regarding the above proposal.

6.2 Testing Results

6.2.1 Scenario A

Baseline Statistics (Quake III v. 1.31)			**Attack-based Statistics (Baseline + Ping Flood)**		
packet capture started ~ Sun Apr 28 19:49:42 2002 with interval = INFINITY.			Packet capture started ~ Sun Apr 28 20:05:24 2002 with interval = INFINITY.		
61739 total packets (0 TCP, 61739 UDP, 0 ICMP).			3315133 total packets (0 TCP, 64247 UDP, 3250886 ICMP).		
mean IP source addr	=	3232235522.453	mean IP source addr	=	3232235521.515
mean IP destin addr	=	3232660079.465	mean IP destin addr	=	3232243428.207
mean IP header len	=	20.000	mean IP header len	=	20.000

mean tot packet size	=	76.334	mean tot packet size	=	83.870
mean source port	=	27942.249	mean source port	=	27941.151
mean dest port	=	27940.659	mean dest port	=	27937.077
mean IP ttl	=	127.931	mean IP ttl	=	96.619
mean TCP flags	=	0.000	mean TCP flags	=	0.000
mean TCP seq number	=	0.000	mean TCP seq number	=	0.000
mean TCP advert win	=	0.000	mean TCP advert win	=	0.000
mean TCP urg ptr	=	0.000	mean TCP urg ptr	=	0.000
var IP source addr	=	0.794	var IP source addr	=	0.278
var IP destin addr	=	337049960088663.062	var IP destin addr	=	6280205919250.921
var IP header len	=	0.000	var IP header len	=	0.000
var tot packet size	=	886.430	var tot packet size	=	19.725
var source port	=	476661.267	var source port	=	520051.909
var dest port	=	513319.064	var dest port	=	613969.991
var IP ttl	=	8.413	var IP ttl	=	1023.691
var TCP flags	=	0.000	var TCP flags	=	0.000
var TCP seq number	=	0.000	var TCP seq number	=	0.000
var TCP advert win	=	0.000	var TCP advert win	=	0.000
var TCP urg ptr	=	0.000	var TCP urg ptr	=	0.000
stdv IP source addr	=	0.891	stdv IP source addr	=	0.527
stdv IP destin addr	=	18358920.450	stdv IP destin addr	=	2506033.902
stdv IP header len	=	0.000	stdv IP header len	=	0.000
stdv tot packet size	=	29.773	stdv tot packet size	=	4.441
stdv source port	=	690.407	stdv source port	=	721.146
stdv dest port	=	716.463	stdv dest port	=	783.562
stdv IP ttl	=	2.901	stdv IP ttl	=	31.995
stdv TCP flags	=	0.000	stdv TCP flags	=	0.000
stdv TCP seq number	=	0.000	stdv TCP seq number	=	0.000
stdv TCP advert win	=	0.000	stdv TCP advert win	=	0.000
stdv TCP urg ptr	=	0.000	stdv TCP urg ptr	=	0.000
capture duration was 842 seconds.			capture duration was 880 seconds.		

6.2.2 Scenario B

Baseline Statistics (Quake III v. 1.31 + File Transfer)			**Attack-based Statistics (Baseline + Broadcast Ping Flood)**		
packet capture started ~ Wed May 8 18:50:46 2002 with interval = INFINITY.			packet capture started ~ Wed May 8 18:57:54 2002 with interval = INFINITY.		
100919 total packets (62632 TCP, 37683 UDP, 604 ICMP).			1453832 total packets (3389 TCP, 36454 UDP, 1413989 ICMP).		
mean IP source addr	=	3232235523.525	mean IP source addr	=	3232235524.168
mean IP destin addr	=	3232235524.752	mean IP destin addr	=	4059212162.117
mean IP header len	=	20.000	mean IP header len	=	20.000
mean tot packet size	=	763.132	mean tot packet size	=	86.484
mean source port	=	35257.596	mean source port	=	28887.093
mean dest port	=	16123.084	mean dest port	=	26462.693
mean IP ttl	=	88.125	mean IP ttl	=	102.682
mean TCP flags	=	17.117	mean TCP flags	=	17.226
mean TCP seq number	=	32744.109	mean TCP seq number	=	32351.447
mean TCP advert win	=	13227.994	mean TCP advert win	=	13714.601
mean TCP urg ptr	=	0.000	mean TCP urg ptr	=	0.000

var IP source addr	=	2.751	var IP source addr	=	0.360
var IP destin addr	=	5.725	var IP destin addr	=	194959281552854496.000
var IP header len	=	0.000	var IP header len	=	0.000
var tot packet size	=	486388.217	var tot packet size	=	3388.192
var source port	=	255339040.946	var source port	=	47234850.800
var dest port	=	306729663.657	var dest port	=	59607330.241
var IP ttl	=	962.296	var IP ttl	=	5686.930
var TCP flags	=	7.703	var TCP flags	=	8.476
var TCP seq number	=	356755236.802	var TCP seq number	=	284190317.881
var TCP advert win	=	59860539.363	var TCP advert win	=	65599430.935
var TCP urg ptr	=	0.000	var TCP urg ptr	=	0.000
stdv IP source addr	=	1.659	stdv IP source addr	=	0.600
stdv IP destin addr	=	2.393	stdv IP destin addr	=	441541936.347
stdv IP header len	=	0.000	stdv IP header len	=	0.000
stdv tot packet size	=	697.415	stdv tot packet size	=	58.208
stdv source port	=	15979.332	stdv source port	=	6872.762
stdv dest port	=	17513.699	stdv dest port	=	7720.578
stdv IP ttl	=	31.021	stdv IP ttl	=	75.412
stdv TCP flags	=	2.776	stdv TCP flags	=	2.911
stdv TCP seq number	=	18887.965	stdv TCP seq number	=	16857.945
stdv TCP advert win	=	7736.959	stdv TCP advert win	=	8099.348
stdv TCP urg ptr	=	0.000	stdv TCP urg ptr	=	0.000
capture duration was 302 seconds.			capture duration was 302 seconds.		

6.2.3 Scenario C

Baseline Statistics **(File Transfer, Normal Pings, + Movie)**			**Attack-based Statistics** **(Baseline + Broadcast Ping Flood)**		
Packet capture started ~ Wed May 8 14:40:09 2002 with interval = INFINITY.			packet capture started ~ Wed May 8 15:26:26 2002 with interval = INFINITY.		
74555 total packets (73906 TCP, 43 UDP, 606 ICMP).			883264 total packets (92281 TCP, 12 UDP, 790971 ICMP).		
mean IP source addr	=	3232235524.864	mean IP source addr	=	3232235524.449
mean IP destin addr	=	3232278140.664	mean IP destin addr	=	4034490542.696
mean IP header len	=	20.000	mean IP header len	=	20.000
mean tot packet size	=	1130.095	mean tot packet size	=	184.347
mean source port	=	37342.581	mean source port	=	17253.930
mean dest pobrt	=	8591.018	mean dest port	=	5296.001
mean IP ttl	=	74.506	mean IP ttl	=	95.014
mean TCP flags	=	17.124	mean TCP flags	=	16.707
mean TCP seq number	=	31749.438	mean TCP seq number	=	33966.644
mean TCP advert win	=	15786.392	mean TCP advert win	=	23890.464
mean TCP urg ptr	=	0.000	mean TCP urg ptr	=	10.750
var IP source addr	=	3.199	var IP source addr	=	1.555
var IP destin addr	=	33847829855049.199	var IP destin addr	=	208969018109775552.000
var IP header len	=	0.000	var IP header len	=	0.000
var tot packet size	=	365518.651	var tot packet size	=	131407.676
var source port	=	607210931.208	var source port	=	596144796.112
var dest port	=	372350533.139	var dest port	=	192757724.512
var IP ttl	=	582.537	var IP ttl	=	4426.206
var TCP flags	=	7.765	var TCP flags	=	5.172

var TCP seq number	=	363273113.358	var TCP seq number	=	352603013.114
var TCP advert win	=	101266777.029	var TCP advert win	=	146312714.716
var TCP urg ptr	=	0.000	var TCP urg ptr	=	323062.658
stdv IP source addr	=	1.789	stdv IP source addr	=	1.247
stdv IP destin addr	=	5817888.780	stdv IP destin addr	=	457131291.983
stdv IP header len	=	0.000	stdv IP header len	=	0.000
stdv tot packet size	=	604.581	stdv tot packet size	=	362.502
stdv source port	=	24641.650	stdv source port	=	24416.077
stdv dest port	=	19296.387	stdv dest port	=	13883.722
stdv IP ttl	=	24.136	stdv IP ttl	=	66.530
stdv TCP flags	=	2.787	stdv TCP flags	=	2.274
stdv TCP seq number	=	19059.725	stdv TCP seq number	=	18777.727
stdv TCP advert win	=	10063.140	stdv TCP advert win	=	12095.979
stdv TCP urg ptr	=	0.000	stdv TCP urg ptr	=	568.386
capture duration was 302 seconds.			capture duration was 302 seconds.		

6.3 Statistical Analysis

It is obvious that certain statistics contrast drastically between baseline information and attack-based information. We analyzed the following fields:

- ♣ Time run
- ♣ Total Packets Captured
- ♣ ICMP Packets Collected
- ♣ Mean IP TTL
- ♣ Variance of the IP Source Address
- ♣ Variance of the Total Packet Size

The following tables show the differences in these variables under both normal and flooded circumstances.

Scenario A

	Normal	Flood
Time Elapsed	842	880
Total Packets	61739	3315133
ICMP Packets Collected	0	3250886
Mean IP TTL	127.931	96.619
Variance: IP Source Addr	0.794	0.278
Variance: Total Packet Size	886.430	19.725

Table 6.1

Scenario B

	Normal	Flood
Time Elapsed	302	303
Total Packets	100919	1453832
ICMP Packets Collected	604	1413989
Mean IP TTL	88.125	102.682
Variance: IP Source Addr	2.751	0.360

Variance: Total Packet Size	486388.217	3388.192

Table 6.2

Scenario C

	Normal	**Flood**
Time Elapsed	302	302
Total Packets	74555	883264
ICMP Packets Collected	606	790971
Mean IP TTL	74.504	95.014
Variance: IP Source Addr	3.199	1.555
Variance: Total Packet Size	365518.655	131407.676

Table 6.3

It is evident from the consistent differences in the analyzed variables above that a ping flood can easily be recognized. The total packets are consistently higher during the flood. The ICMP Packets Collected is much higher. The variance of the IP source address is quite interesting because it shows that most of the packets were coming from one address only, clearly designating an attack. Lastly, the variance of the total packet size is quite a bit smaller during the attack because the majority of the packets are of the same size.

While the previous passage describes how we know that a ping flood is taking place, the Mean IP TTL field differentiates between a directed ping flood and a broadcast ping flood. The first scenario is the result of a directed ping flood and the Mean IP TTL is lower than the baseline. However, the Mean IP TTL is higher on scenarios two and three, which were both broadcast ping floods. Therefore, not only were we able to detect that a ping flood has occurred, but we were also able to tell what kind of ping flood was instantiated on the network.

7. Summary

7.1 Simulation Conclusions

As per our hypothesis, we have found that by calculating statistics on the network packet fields, we were able to observe and measure differences between normal network traffic statistics and statistics calculated during a network attack. This was derived from the data collected and through comparing and contrasting each of the baseline statistics sets with their corresponding network attack statistics. We also feel that furtherance of our research will allow a database of attack patterns to be created which will in turn open the gateway for the development of tools that accurately map network anomalies to known attacks as well as being able to possibly learn new patterns dynamically.

7.2 Open Issues

Due to the low amount of resources available to us and the short time frame there are many areas in which our experimentation and research can be furthered and improved. For instance, research must be conducted on larger scale network that more closely resembles industry environments. The same testing methodology can be applied to larger networks and conclusions can be gathered from the information obtained. It must be understood that different times of the day, week, and year will provide us with a diverse range of baseline network traffic. For instance, it is likely that at the University of Maryland there will be more traffic during the week as opposed to the weekend, and in addition there will be more traffic during a week in the month of October then there will be during a week in the month of July.

Another area of improvement is to calculate more statistical pairs of baseline vs. attack. For instance, the calculation of packet rate (overall, Source, Destination, Source/Destination), weighted statistics on source and destination addresses (i.e. heaviest traffic user, computer sending most packets, etc.) and transaction frequency (per address) could be added to the functionality of the Cstat module, and attacks such as denial of service and several types of worms could be used.

All of these factors need to be accounted for when interpreting and making conclusions from the information gathered by our research. One way to extend our research would be to apply the methodology weekly to a network for one year. This would give a broader perspective of network traffic statistics and would allow us to more precisely create a product that will be able to identify, react to, and prevent network attacks.