DETECTING NETWORK INTRUDERS CONNECTED THROUGH LONG STEPPING-STONE CHAINS

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Wei Ding

August 2014

DETECTING NETWORK INTRUDERS CONNECTED THROUGH LONG STEPPING-STONE CHAINS

Wei Ding

Acknowledgements

My utmost gratitude goes to my supervisor, Dr. Stephen Huang for allowing me to join his team, for his knowledge, expertise, and kindness. My thanks and appreciation goes to my dissertation committee members, Dr. Ernst Leiss, Dr. Kam-Hoi Cheng, Dr. Ricardo Vilalta, and Dr. Tsorng-Whay Pan. I thank Ming-Chih Shih, Ying-Wei Kuo, and Grace Wu for their help and comments on my research and during group presentations. I also appreciate all the students in our summer REU program who helped me a lot with experiments. Above all, I thank my parents who give me the most selfless love and the strongest support, and my thanks go to my wife Xiaofan Wu who cares about my life, tolerates my temper, and shares happiness and sadness with me. Finally, I would like to thank all my friends.

DETECTING NETWORK INTRUDERS CONNECTED THROUGH LONG STEPPING-STONE CHAINS

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Wei Ding

August 2014

Abstract

A common technique hackers use to avoid being detected is to route their network connections through a chain of stepping-stone hosts. There is no valid reason to use a long connection chain for remote login such as SSH connection. In this dissertation, we focus on protecting hosts from being attacked via stepping-stone connection chains. Our objective is to detect intruders at a stepping-stone host in the middle of the connection chain and at the target host at the end of the chain.

Along with the developing of correlation-based stepping-stone detection algorithms, hackers also developed new techniques to evade being detected. Hackers can add chaff packets or jitter the original packets to decrease the detection rate of these correlation algorithms. Dealing with chaff packet-added intrusions has already been studied, while the jittering part hasn't been touched. Our jittering detection algorithm utilizes statistical distributions to fit the inter-arrival time gaps of traffic flows, extracting features from fitting, and separates jittered ones from normal ones by using support vector machines. The algorithm does not work well for light jittering. Hence, we further propose a hybrid stepping-stone detection algorithm to employ both correlation and jitter detection algorithms to detect intrusions. Experiment results show that our hybrid stepping-stone detection algorithm can successfully detect more than 90% stepping-stone intrusions in most cases with a 0% false positive rate.

It is always important for a host to protect itself from being a victim. To detect long connection chain intrusions at the target host, we propose two detection algorithms: a nearest neighbor-based algorithm and an anomaly detection-based algorithm. The first algorithm centers around analyzing the delay between the time a user presses "enter" to finish a command and the time that the user types the next character, and uses an approximated upstream round-trip time to separate a long connection chain from short ones. Experiment results show that our method can correctly identify long chains from short ones with good accuracy. Besides, based on the idea of anomaly behavior detection, a novel method to identify long connection chains from short chains using a pre-defined short chain profile has been proposed. Each new connection will be compared to the profile. Any connection that differs significantly from the profile will be considered as a suspicious long connection. In addition, several methods are proposed to increase the detection rate by adapting to a user's different typing speed. This algorithm can get better detection accuracy compared to the first one.

With the algorithms proposed in this dissertation, we can detect stepping-stones in the middle of the chain in a robust way, and we can further and more effectively protect victim hosts from stepping-stone intrusions at the end of the chain.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As far back as 50 B.C., diplomats and military commanders already understood that it was necessary to provide some mechanism to protect the confidentiality of messages and to have some ways of detecting message tampering [1]. Julius Caesar was credited with the invention of the Caesar Cipher. Later, by the time of the First World War, encoding became more sophisticated as machines were employed to scramble and unscramble information. The end of the 20th century and early years of the 21st century saw rapid advancements in telecommunications, computing hardware and software, and data encryption. With the development of smaller, more powerful, and less expensive computing equipment, electronic data processing devices reached small business and home users. These computers quickly became interconnected through the Internet.

Due to the rapid growth and widespread use of computers and electronic business conducted through the Internet, as well as the numerous occurrences of international terrorism, we need better methods to protect the computers and the information they store, process and transmit. The common goals of ensuring the security and reliability of information systems become a very important research field in the context of computer science. Information security and network security are two major branches.

In this chapter, we first introduce the definition of information and network security. After that, we discuss the necessity of intrusion detection and briefly introduce the classification of it. Next, we talk about stepping-stone detection which is the major topic of this dissertation. Last, the organization of this dissertation will be introduced.

1.1 Information and Network Security

Information security is the practice of keeping information's confidentiality, integrity and availability [2][3][4][5][6][7]. Applying security to information is analogous to the application of security to any physical asset. Historically, people referred to information security by a number of different terms, such as data security, IT security or computer security. Confidentiality is the assurance that information is shared only among authorized parties. Breaches of confidentiality may take place during the transmission of the information, such as transferring unencrypted data in a public network. Integrity assures that the information is authentic and complete. Integrity is one of the primary indicators of information security. The integrity of data means the data are correct, not tampered, as well as it can be trusted and relied upon. Availability assures that the systems used to deliver, store, and process information are accessible. To invalidate the availability, denial of service (DOS) or distributed denial of service (DDOS) attacks are widely used to launch the attack the systems [8][9][10][11][12][13]. Besides these three root components of information security, scholars added accountability and auditability as another two core attributes of information security. These two attributes emphasize on keeping someone is personally accountable and the system is auditable.

Nowadays, most attacks against information systems are launched through networks. Network security became an unneglectable part of information security. Network security consists of the provisions and policies adopted by a network administrator to prevent and monitor unauthorized access, misuses, modifications, or denial of a computer network and network-accessible resources [14][15]. Network security concerns protecting a computer network from unauthorized accesses, misuses or modifications. Authentication is the first step of network security, commonly completed by using a username and a password as well as other further authentication methods such as implementing security token or fingerprint. Network security is often used interchangeably with information security, which focuses on protecting data resources from unauthorized access or attacks as well as misuse by legitimate users inside an organization.In practice, software or hardware tools used to achieve network security or information security may overlap. These two tasks normally complement each other in the sense that if you cannot make sure the network is secure, you can never guarantee the information or the information system in this network is secure. The purpose of most network attacks is to jeopardize the information security among the systems in the network. This dissertation will mostly discuss research and methods to solve information and network security problems.

1.2 Intrusion Detection

Network security and intrusion detection have become important topics of active research even in the last century [16][17][18][19]. As the use of the Internet becomes more common and widespread, there are more network attacks and security breaches. Because the growing number of network attacks is more costly, intrusion detection now plays a more crucial role in ensuring the smooth operation of computer networks. Intrusion detection has become one of the most important techniques to secure our networks and systems. It is the process of monitoring the network and the system activities to detect malicious activities or policy violations which are those actions attempting to compromise the confidentiality, integrity or availability of a resource. Intrusion detection techniques are often combined together with intrusion prevention methods which is used to stop possible incidents detected.

Intrusion detections can be implemented on a single host or a whole network [20][21][22][23][24]. Host-based intrusion detection (HIDS) is used to identify unauthorized, illicit, and anomalous behavior on a device. HIDS normally requires an agent to be installed on the system to monitor and alert local OS and application activities. The detection agent may combine attacking signatures, detecting rules and heuristic methods to identify unauthorized or suspicious activities. The host-based intrusion detection is passive only focusing on gathering, identifying, logging, and alerting suspicious detected behavior.

Network based intrusion detection is implemented to monitor the whole network which attempts to identify unauthorized, illicit, and anomalous behavior based on analyzing captured network traffic [25][26][27][28][29]. Many tools can be used to filter and capture network traffics on a single device. A network-based intrusion detection system mostly utilizes either a network tap, span port, or a switch to collect packets that traverse a given network. Using the captured data, the IDS system processes and flags any suspicious traffic. A network-based intrusion detection system also does a passive job by gathering, identifying, logging, and alerting the network administrator after finding a suspicious activity.

We will introduce more detailed techniques regarding intrusion detection and intrusion detection systems in the next chapter. Besides, we will discuss active intrusion detection techniques, such as intrusion prevention systems, as well as widely deployed intrusion detection and prevention systems and products later.

1.3 Stepping-stone Detection

Living in the information era, most computers and mobile devices are connected via the Internet. Keeping our electronic devices and data safe from intruders is essential. Nowadays, hackers are smart enough to try to use various techniques to intrude other's system, steal data, and evade being caught at the same time. In order for intruders to steal information from a computer host, it is necessary for the intruders to remotely login to the host and establish a connection session. When a person logs into one computer and uses this computer for logging into another, or even a number of other computers, we call this sequence of logins a connection chain [30]. Any intermediate host on a connection chain is called a stepping stone. To avoid being detected, most of these intruders use long connection chains of stepping stones to reach their destination host (called the "victim" host). This process is illustrated in Figure 1.1 below.

Attacking through stepping stones is a technique widely used by network attackers to attain anonymity and prevent to be traced back. Attackers don't attack others directly from their own computer but from intermediate hosts which are already compromised and controlled by the attackers. Intruders often collect some accounts on compromised hosts, and before they conduct a new attack, they log in through this series of hosts before finally carrying out an assault or intrusion on the target. These hosts can be heterogeneous, diversely located, and even may be in many different countries. Hence, it's very difficult to trace an attack back through these un-administrable and unaccountable hosts to the origin.

There are many benefits to detecting stepping stones [31]. A lot of research regarding stepping-stone detection had been done during the last decade [32][33][34] [35][36][37][38]. Staniford-Chen and Heberlein are forerunners who first addressed the problem of detecting stepping stones in a ground-breaking paper [30]. With respect to the intermediate hosts, it helps to flag suspicious activity, to maintain logs in case a break-in is subsequently detected as having come from the local site, to detect inside attackers laundering their connections through external hosts, to enforce policies regarding transit traffic, and to detect insecure combinations of legitimate connections.

We will discuss stepping-stone detection related issues, techniques and algorithms in the following chapters. Details of our study will be presented in Chapter Three.

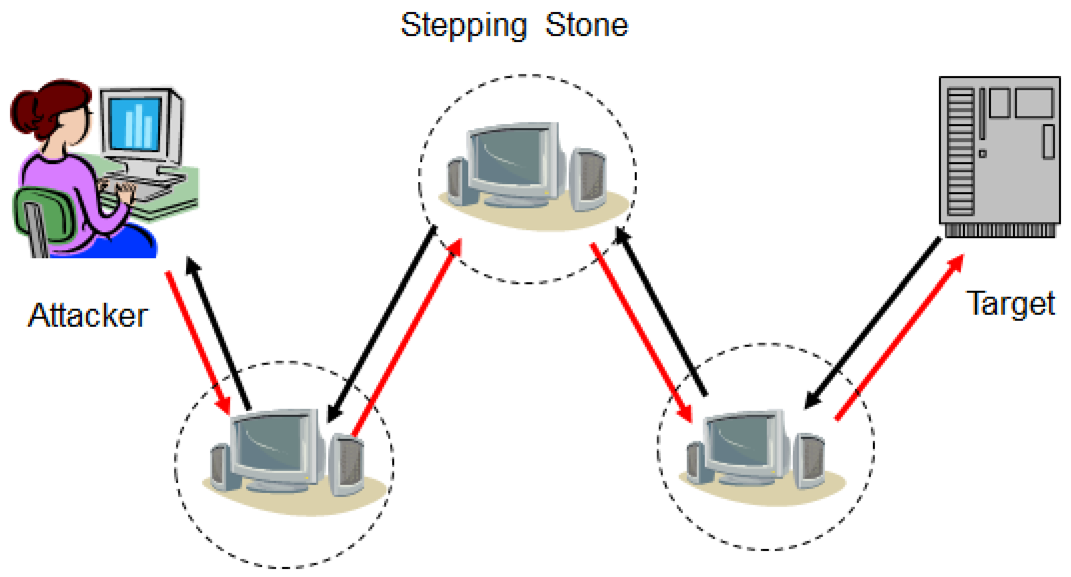


Figure 1.1: Illustration of Using Stepping Stones to Attack a Victim Host

1.4 Organization of the Dissertation

The rest of the dissertation is organized as follow. Chapter 2 will review and discuss some research which is directly related to our study. In addition, we will introduce some commonly used network traffic analysis tools used in our research to collect packet data. Furthermore, we also present several network traffic manipulation tools which are used not only by security researchers, but also hackers. As one can imagine, the only way one can defeat hackers is one can perform as a "hacker" oneself. Based on the order of our research, in the following two chapters, we will discuss detecting stepping-stone intrusions at the end of the chain in the first place. After that, in Chapter 5, we will switch back to discuss detecting stepping stones in the middle of the chain. In Chapter 3, we present a new method to extract two different time gaps from a linux commands constituted network traffic stream. To complement these intermediate hosts based stepping-stone detection algorithm, we propose the idea to detect long stepping-stone connection chains at the end of the chain. In this chapter, we also discuss the impact of a user's typing delay for estimating round-trip time (RTT). Chapter 4 discusses a solution to deal with intrusions from a long connection chain. The study involves a strategy to quantitatively measure the distance between long connection chains and short connection chains. A distance based long connection detection algorithm will be presented. Furthermore, in Chapter 5, we introduce several evasion techniques developed by hackers. These stepping-stone detection evasion techniques can invalidate formerly developed detection methods to a large extent. To complement previous stepping-stone detection algorithms, we propose a new jittered traffic detection algorithm which utilizes the characteristics of intentionally manipulated network traffic to detect smarter intrusions. Last, the concluding section summarizes our work and points out potential further improvement.

Chapter 2

Reviews and Related Work

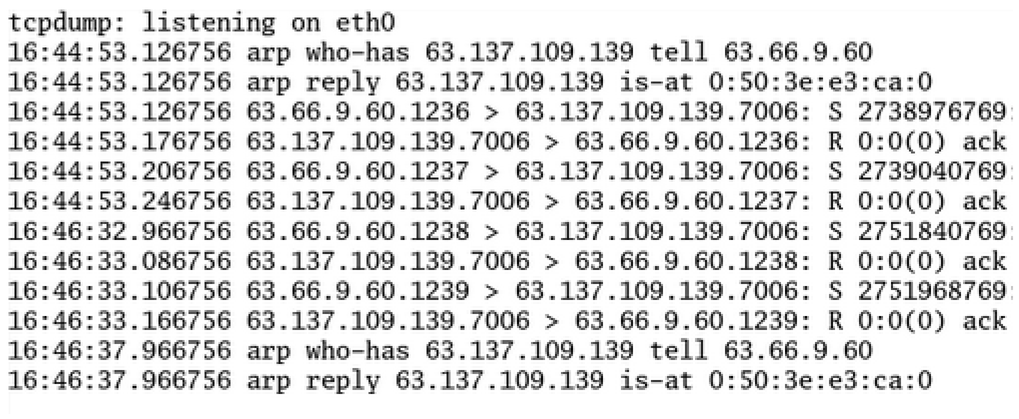
2.1 Network Traffic Analysis Libraries and Tools

To monitor network traffic, we need a tool to capture the network packets. Packet capturing is the process of intercepting and logging all packets going through a point in the network. There are many packet sniffers developed to capture network packets, either in the Ethernet or wireless networks [39][40][41][42][43]. These packet sniffers are either implemented as a computer program or made as a piece of computer hardware. They use a network interface controller (NIC) to monitor the traffic in and out of an interface. Furthermore, a promiscuous mode can be turned on to enable sniffing among the whole local network. If necessary, packet analysis tools can decode the packet's raw data, showing the values of various fields in the packet, and analyze the content according to specifications [44].

2.1.1 Libpcap/Winpcap and Tcpdump

Libpcap [45] is the most widely used C/C++ library for network traffic capture. It was originally developed by the Tcpdump team in the Network Research Group at Lawrence Berkeley Laboratory [46]. Libpcap consists of an application programming interface (API) for capturing network traffic. Libpcap was a library implemented for Unix-like systems. For Windows operating systems, there is an implementation of libpcap known as WinPcap [47]. The core of both Libpcap and Winpcap was originally written in C, and it has been implemented in many other languages using a wrapper. Libpcap and WinPcap can save captured packets into a file or read files containing saved packets. A captured file saved in Pcap format can be read by applications, such as TCPdump, Wireshark, CA NetMaster, etc.

Built on top of Pcap, Tcpdump [48][49] is the most well known and powerful command-line packet analyzer on Unix-like systems. It allows the user to intercept and display TCP/IP and other packets being transmitted or received over a network to the attatched network interface controller. The Windows version of Tcpdump is called WinDump [50]. A user needs special privileges, such as root, on a system to run Tcpdump. For unencrypted traffic such as Telnet and HTTP, captured with Tcpdump, one can view login IDs, passwords, as well as other unencrypted information easily with the optionally applied Berkeley Packet Filter (BPF)-based [51] filter. A sample output of Tcpdump is shown in Figure 2.1. From the figure, one can see that the captured information can be divided into several categories: time stamp, protocol, sender's IP, receiver's IP, direction, sequence number, etc.

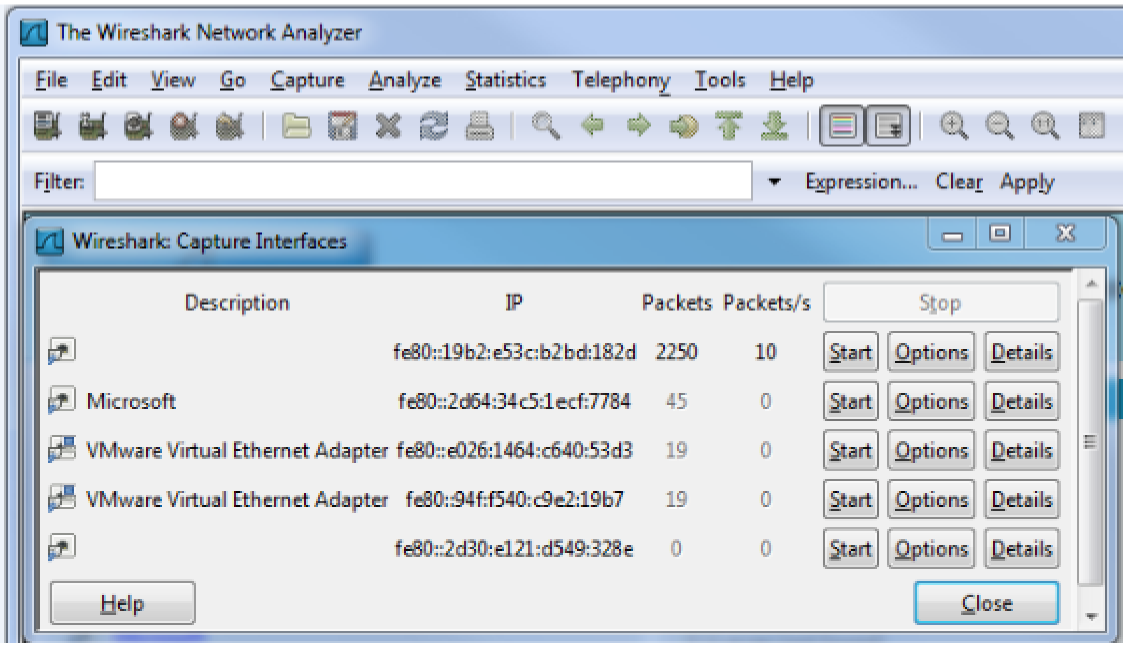


```
tcpdump: listening on eth0
16:44:53.126756 arp who-has 63.137.109.139 tell 63.66.9.60
16:44:53.126756 arp reply 63.137.109.139 is-at 0:50:3e:e3:ca:0
16:44:53.126756 63.66.9.60.1236 > 63.137.109.139.7006: S 2738976769
16:44:53.176756 63.137.109.139.7006 > 63.66.9.60.1236: R 0:0(0) ack
16:44:53.206756 63.66.9.60.1237 > 63.137.109.139.7006: S 2739040769
16:44:53.246756 63.137.109.139.7006 > 63.66.9.60.1237: R 0:0(0) ack
16:46:32.966756 63.66.9.60.1238 > 63.137.109.139.7006: S 2751840769
16:46:33.086756 63.137.109.139.7006 > 63.66.9.60.1238: R 0:0(0) ack
16:46:33.106756 63.66.9.60.1239 > 63.137.109.139.7006: S 2751968769
16:46:33.166756 63.137.109.139.7006 > 63.66.9.60.1239: R 0:0(0) ack
16:46:37.966756 arp who-has 63.137.109.139 tell 63.66.9.60
16:46:37.966756 arp reply 63.137.109.139 is-at 0:50:3e:e3:ca:0
```

Figure 2.1: Sample Output of Tcpdump

2.1.2 Wireshark and Tshark

Wireshark is an open-source cross-platform packet analyzer with graphic user interface (GUI) [52][53][54][55]. It was originally named Ethereal. In 2006, the project was renamed Wireshark due to trademark issues [56]. Wireshark is widely used for network troubleshooting, analysis, software and communications protocol development, and education. Our research mostly uses Wireshark to capture and analyze network traffic. Wireshark utilizes Pcap library to capture packets. The newest version of Wireshark re-implemented its GUI with the Qt toolkit. There is also a command-line based version called TShark which doesn't have a GUI. Tshark can be used flexibly through a SSH connection chain. The basic Wireshark operation interface is shown in Figure 2.2. One can use the capture option to choose the network interface to be monitored. After choosing the interface, one can set up Wireshark capture options as shown in Figure 2.3. After starting, any incoming and outgoing packets on the chosen interface will be captured and displayed by Wireshark as shown in Figure 2.4.


The Wireshark Network Analyzer
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression... Clear Apply
Wireshark: Capture Interfaces
Stop
Start Options Details
Help
Close

Description	IP	Packets	Packets/s
	fe80::19b2:e53c:b2bd:182d	2250	10
Microsoft	fe80::2d64:34c5:1ecf:7784	45	0
VMware Virtual Ethernet Adapter	fe80::e026:1464:c640:53d3	19	0
VMware Virtual Ethernet Adapter	fe80::94f:f540:c9e2:19b7	19	0
	fe80::2d30:e121:d549:328e	0	0

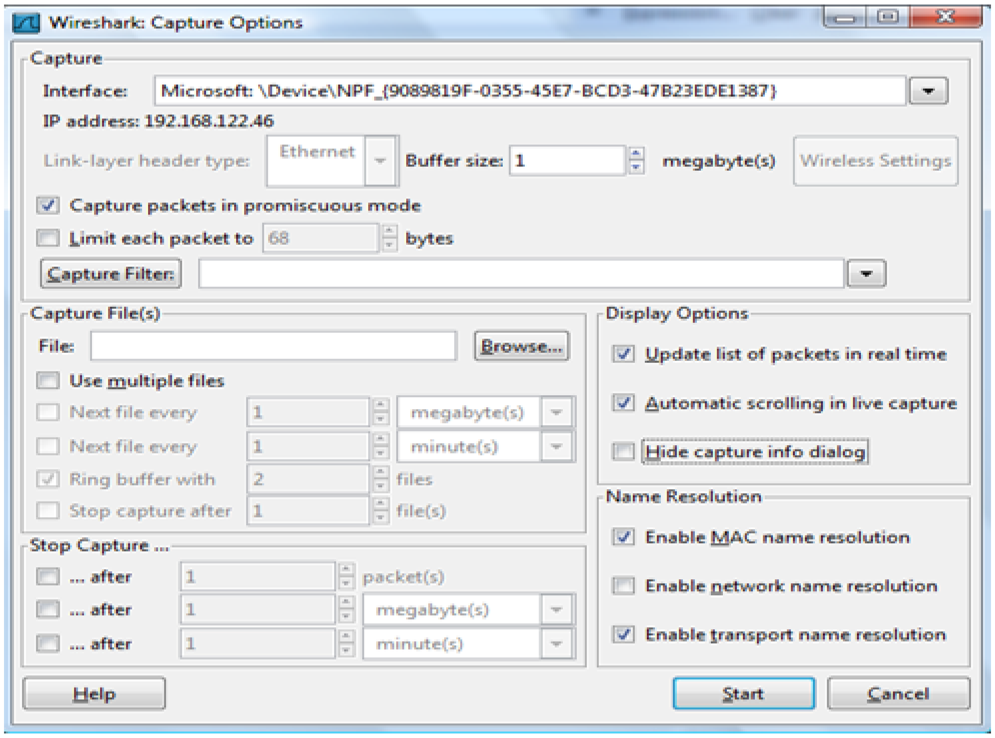
Figure 2.2: Screenshot of Capturing Packets with Wireshark

2.2 Intrusion Detection and Prevention Techniques

There are two types of intrusion detection: host based and network based. To detect intrusions, there are two major methods: signature-based intrusion detection and anomaly-based intrusion detection.

2.2.1 Anomaly-based Detection

Anomaly-based intrusion detection systems monitor system activity and classify it as either normal or abnormal [57][58][59][60][61][62][63]. The classification is based on statistical heuristics and rules, rather than signatures or patterns. This method will detect any type of misuse that falls outside of normal system operation. Normally,



Wireshark: Capture Options

Capture

Interface: Microsoft: \Device\NPF_{9089819F-0355-45E7-BCD3-47B23EDE1387}

IP address: 192.168.122.46

Link-layer header type: Ethernet Buffer size: 1 megabyte(s) Wireless Settings

[✓] Capture packets in promiscuous mode

[ ] Limit each packet to 68 bytes

Capture Filter:

Capture File(s)

File: Browse...

[ ] Use multiple files

[ ] Next file every 1 megabyte(s)

[ ] Next file every 1 minute(s)

[✓] Ring buffer with 2 files

[ ] Stop capture after 1 file(s)

Stop Capture ...

[ ] ... after 1 packet(s)

[ ] ... after 1 megabyte(s)

[ ] ... after 1 minute(s)

Display Options

[✓] Update list of packets in real time

[✓] Automatic scrolling in live capture

[ ] Hide capture info dialog

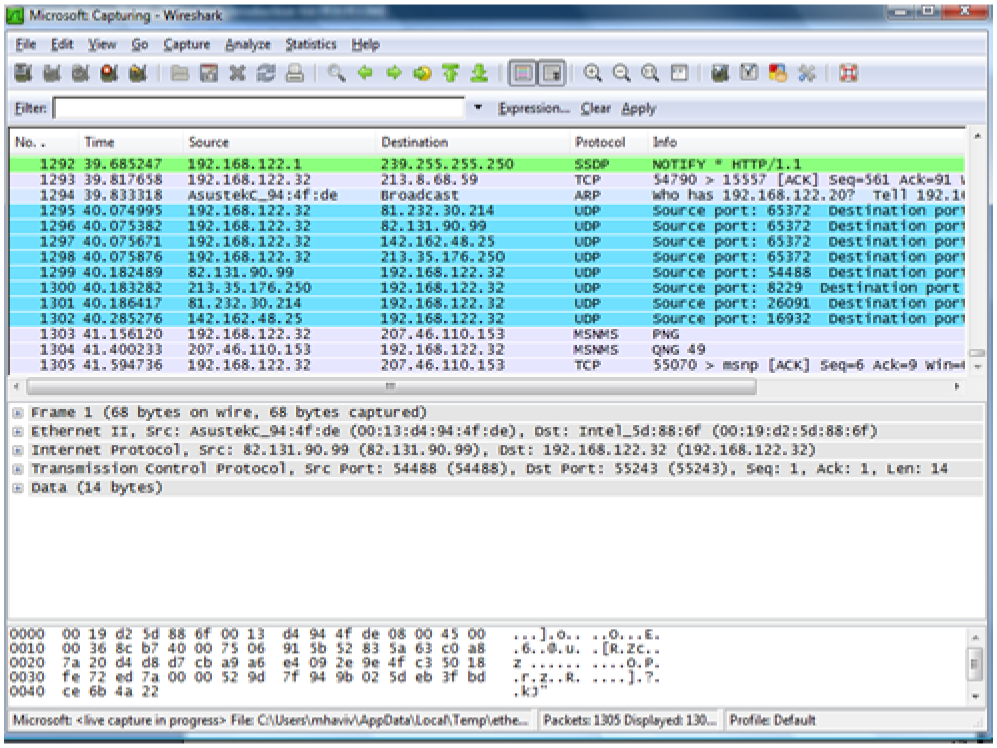
Name Resolution

[✓] Enable MAC name resolution

[ ] Enable network name resolution

[✓] Enable transport name resolution

Figure 2.3: Screenshot of Wireshark Capture Options


Microsoft: Capturing - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply

```
No. .  Time       Source              Destination       Protocol  Info
  1292 39.685247  192.168.122.1       239.255.255.250   SSDP      NOTIFY * HTTP/1.1
  1293 39.817658  192.168.122.32      213.8.68.59       TCP       54790 > 15557 [ACK] Seq=561 Ack=91 W
  1294 39.833318  AsustekC_94:4f:de   Broadcast         ARP       Who has 192.168.122.20?  Tell 192.16
  1295 40.074995  192.168.122.32      81.232.30.214     UDP       Source port: 65372  Destination port
  1296 40.075382  192.168.122.32      82.131.90.99      UDP       Source port: 65372  Destination port
  1297 40.075671  192.168.122.32      142.162.48.25     UDP       Source port: 65372  Destination port
  1298 40.075876  192.168.122.32      213.35.176.250    UDP       Source port: 65372  Destination port
  1299 40.182489  82.131.90.99        192.168.122.32    UDP       Source port: 54488  Destination port
  1300 40.183282  213.35.176.250      192.168.122.32    UDP       Source port: 8229  Destination port
  1301 40.186417  81.232.30.214       192.168.122.32    UDP       Source port: 26091  Destination port
  1302 40.285276  142.162.48.25       192.168.122.32    UDP       Source port: 16932  Destination port
  1303 41.156120  192.168.122.32      207.46.110.153    MSNMS     PNG
  1304 41.400233  207.46.110.153      192.168.122.32    MSNMS     QNG 49
  1305 41.594736  192.168.122.32      207.46.110.153    TCP       55070 > msnp [ACK] Seq=6 Ack=9 Win=
```

```
⊞ Frame 1 (68 bytes on wire, 68 bytes captured)
⊞ Ethernet II, Src: AsustekC_94:4f:de (00:13:d4:94:4f:de), Dst: Intel_5d:88:6f (00:19:d2:5d:88:6f)
⊞ Internet Protocol, Src: 82.131.90.99 (82.131.90.99), Dst: 192.168.122.32 (192.168.122.32)
⊞ Transmission Control Protocol, Src Port: 54488 (54488), Dst Port: 55243 (55243), Seq: 1, Ack: 1, Len: 14
⊞ Data (14 bytes)
```

```
0000   00 19 d2 5d 88 6f 00 13  d4 94 4f de 08 00 45 00   ...].o.. ..O...E.
0010   00 36 8c b7 40 00 75 06  91 5b 52 83 5a 63 c0 a8   .6..@.u. .[R.Zc..
0020   7a 20 d4 d8 d7 cb a9 a6  e4 09 2e 9e 4f c3 50 18   z ...... ....O.P.
0030   fe 72 ed 7a 00 00 52 9d  7f 94 9b 02 5d eb 3f bd   .r.z..R. ....].?.
0040   ce 6b 4a 22                                        .kJ"
```

```
Microsoft: <live capture in progress> File: C:\Users\mhaviv\AppData\Local\Temp\ethe...  Packets: 1305 Displayed: 130...  Profile: Default
```

Figure 2.4: Screenshot of Wireshark Packet Analyzer

the first step is to train the system to recognize normal behavior. This requires a lot of statistical data for training. Machine learning and artificial intelligence techniques are widely used in anomaly based intrusion detection system. Anomaly-based detection methods can detect unknown attacks whenever the attack doesn't fall into normal behavior. However, this method may suffer a major shortcoming with a high false positive rate.

2.2.2 Signature-based Detection

Compared to anomaly based intrusion detection systems, signature-based systems normally can have a lower false positive rate. Signature-based intrusion detection systems (IDSs) monitor the system or network activities, and compare them with pre-determined and pre-compiled attack signatures and patterns [64][65][66][67][68][69][70] [71]. Whenever getting a match, the system will flag the connection as an intrusion. However, signature based detection methods also have the drawback that they cannot detect unknown attacks, such as Zero-Day [72] attacks.

2.2.3 Intrusion Detection and Prevention System

There are a lot of open-source or commercial intrusion detection and prevention systems developed already [73][74][75][76], including products from major security service companies like Cisco, IBM, McAfee, Juniper, Sourcefire, TippingPoint, etc.. A traditional IDS can only detect suspicious behavior and notify the administrator. It doesn't make the further move to automatically stop the attack. Compared to

IDS, intrusion prevention systems (IPSs) can identify malicious activity, log information about the activity and report, block, or stop the malicious activity. Intrusion prevention systems are considered extended and enhanced intrusion detection systems.

Snort [77][78] is the most well known and widely deployed open-source intrusion detection and prevention system. It uses signature-based detection techniques, though it can combine the benefits of signature, protocol, and anomaly-based inspection. With millions of downloads and nearly 400,000 registered users, Snort has already become the de facto standard for IPS. Since Snort is a signature-based intrusion detection system, it is possible to combine our anomaly-based stepping-stone intrusion detection methods with Snort to build a more robust detection system.

2.3 Previous Work Related to Stepping-stone Detection

2.3.1 Detecting Stepping Stones

During the last decade, a lot of research has been done on stepping-stone detection. For different connections incoming to a host and outgoing from the host, these incoming and outgoing connections are managed to be matched into pairs. By successfully matching a incoming and outgoing pair, the host is detected as a stepping stone to other machines. These previous stepping-stone detection techniques can be illustrated in Figure 2.5. There are several connections incoming to and outgoing from the selected intermediate host shown in Figure 2.5. For these incoming and outgoing connections, algorithms can be used to match a incoming and outgoing pair based on time gaps or other information. If this pair is successfully found as shown in Figure 2.5, this host is detected as a stepping stone. Then the connection through this host becomes suspicious to be a stepping-stone attack connection utilized by attackers.

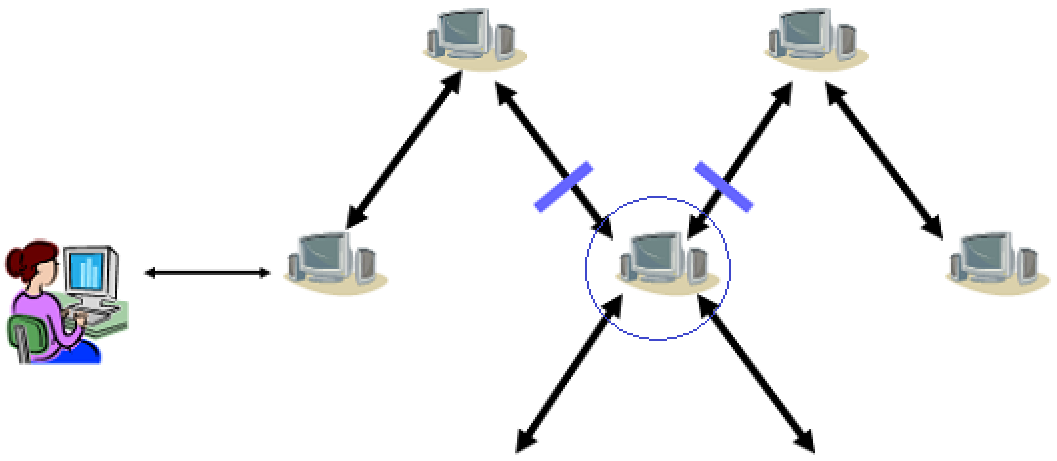


Figure 2.5: Previous Stepping-stone Detection Method

Staniford-Chen and Heberlein [30] are the forerunners in research that deal with the problem of detecting stepping-stone connections. Their early work [30] is based on the content of the traffic. They used thumbprints which are short summaries of the content of a connection, and compared these thumbprints to determine whether two connections contain the same text and are therefore likely to be part of the same connection chain. This method can only deal with unencrypted connections such as Telnet. However for encrypted connections like SSH, it is impossible to check the content of a connection, thus the thumbprint method is not valid anymore.

Wang and Reeves [79] used a more active way to detect intruders. Wang proposed a novel intrusion response framework, the "Sleepy Watermark Tracing (SWT)" framework which integrates a sleepy intrusion response scheme, a watermark correlation technique, and an active tracing protocol. SWT is called "sleepy", because it does not introduce overhead when no intrusion is detected. But, it is actually "active" in that when an intrusion is detected, the target will inject a watermark into the backward connection of the intrusion, and wake up and collaborate with intermediate routers along the intrusion path. Based on their experimental results, the SWT is able to provide a highly efficient and accurate source tracing on interactive intrusions through chained telnet or rlogin which are not encrypted during the transportation compared with SSH connections. For unencrypted connections, the SWT can trace back to the farthest trustworthy security gateway to the origin of intrusion even when there is just one keystroke entered by the intruder. Their work was very encouraging, because they made it workable on the real-time tracing of interactive intrusions that utilize connection chains to disguise their source. This real-time solution not only can stop or detect network-based intrusion, but also can help to detect DDoS by better protecting hosts from being compromised. This method is efficient, robust, and scalable to detect unencrypted intrusion connections which really can help a lot during the Telnet era.

Signal processing technology also can be used in stepping-stone intrusion detection. Zhao used correlation coefficients, such as Spearman Rank, Kendall Tau Rank, and Pearson Product-Moment, to correlate two sessions for identifying stepping-stone intrusions [80]. He is the first one to apply this correlation coefficient method to stepping-stone intrusion detection. His method does not need to monitor a session for a long time to conclude a stepping-stone intrusion. His experimental results showed that a stepping-stone intrusion can be detected while an intruder just inputs the username and password.

He and Tong [81, 82, 83] proposed the Detect-Match (DM) and Detect-Maximum-Variation (DMV) algorithms to detect stepping-stone connection pairs. In their paper [81], they used a perspective of signal processing to formulate the problem as a nonparametric hypothesis testing. They examined stepping-stone connection pairs at intermediate hosts, and allowed timing perturbation, proportional chaff rate and encrypted traffics. Their algorithms are proven to have exponentially decaying false alarm probabilities when normal traffic can be modeled as Poisson processes.

Besides these normal detection methods, He and Tong [84] proposed a way randomizing packet transmissions to defense against stepping-stone attacks. Their method can deal with encrypted and padded packets with perturbed timing and inserted chaff. But for time-sensitive applications, this method may not be desirable. To prevent such stepping-stone attacks, it is critical to detect the connection chains.

Motivated by the Longest Similar Subsequence (LSS) algorithm, Wan and Huang tried to detect intruders by comparing the similarity of two thumbprints [85]. A thumbprint is a summary of a connection that characterizes the connection. The packet gap thumbprint consists of sequences of nonnegative real numbers representing the time gaps between "request" packets. By defining the similarity between two nonnegative real number sequences, they introduced epsilon-similarity, partial sum, and longest epsilon-similar subsequence. By using a property of the partial sums, they proposed an algorithm based on the dynamic programming technique and reduced the time complexity on the matching to $O(mn(m+n))$.

With the increasing requirement of privacy, unencrypted remote login techniques are replaced by encrypted access techniques, such as SSH. Hence, the former contents-based intrusion detection methods are not valid any more. To deal with this change, Zhang and Paxson [31] used the timing correlation of ON/OFF periods of different connections to detect stepping-stone attacks. By leveraging the distinct properties of interactive network traffic, such as smaller packet sizes, longer idle periods than machine-generated traffic, their method is able to ignore the data contents of the connections to detect encrypted connections. By running their algorithm on a site's Internet access link, it not only can provide good accuracy, but also minimize the packet capture load by only recording packet headers with the packet filter. One shortcoming of their method was that it failed to separate legitimate stepping stones that users routinely traverse for a variety of reasons. One possible solution to this problem is to refine their security policies for addressing these legitimate stepping stones.

2.3.2 Detecting Downstream Stepping Stones

Yung [86] investigated request and response pairs between a client and a downstream server to estimate the round-trip time for outgoing connections. Yung's method of echo-delay comparison monitors an outgoing connection to estimate two important time gaps. The first monitored gap between the client request and the server delayed acknowledgment estimates the round-trip travel time between the client and the server. The second monitored gap between the client request and the server reply echo estimates how far downstream the final victim is away. These two time gaps were also used to provide information on the number of downstream hops. Compared with other approaches, this method works in isolation which means it doesn't match for similar sessions on the same connection chain. This strategy also allows benign, short connection chains common in practice. His method can be used on interactive terminal sessions, such as Telnet and SSH. The method showed good performance in identifying sessions with more than two downstream hops.

Yang and Huang [87] proposed an algorithm to detect the length of a downstream connection chain by monitoring packets of outgoing and incoming connections. The algorithm is able to compute the round trip time gap between a client's request packet and the server's response packet. By monitoring the changes in these gaps, the number of hosts in the downstream chain can be estimated. This technique can be used to stop network intrusions when intruders are connected to the target host. Their approach has many advantages compared with previous methods. It is able to detect intrusions in real-time, which is significantly important in practice. It shows better accuracy to estimate the downstream chain length. One of the restrictions of this algorithm is the assumption that one can monitor the packets from the beginning of the connection to the height of the connection chain which was not needed in Yung's [86] work. Furthermore, like some other approaches in this area, their algorithm can only detect the length of downstream chain.

After Yang and Huang's work [87], they proposed two algorithms [88] to match TCP Send and Echo packets. One is conservative and the other is heuristic. The conservative algorithm can accurately match a smaller number of packets, while the heuristic algorithm can match a higher number of packets, with with less accuracy. The authors also prove that their conservative algorithm matches packets correctly. By applying these two algorithms on the Internet to detect long connection chains, their experiments show that the two algorithms can get the same results. If the conservative algorithm failed to produce enough data, the heuristic one will be used to approximate it. The combined method has the advantage of the ability to detect intruders in real-time. In addition, it can handle encrypted sessions with accurate chain length estimation, as well as tolerating the network traffic fluctuation and the network load. The major disadvantage of this method is the requirement of being able to monitor the packets throughout a connection session.

Because the previous work [88] cannot find RTT for too many send packets by using a conservative algorithm, Yang and Huang proposed a new clustering partitioning algorithm [89] to find a TCP packet RTTs from timestamps of the send and echo packets of a connection chain. The former method [88] matches send and echo packets locally; their new method looks at all packets together to produce TCP packets matches which is called a global approach. By capturing all the send and echo packets of a connection chain in a certain time interval and computing the difference between each send packet and echo packets received after it, it can be sure that the correct RTT will be among these differences. Based on this observation, their approach finds the subset which can truly represent the RTT. The experimental result showed that this algorithm can estimate the length of a connection more accurately and has a largely decreased false positive error and false negative error in detecting stepping-stone intrusion compared with the methods proposed before in [31][86][87][88].

2.3.3 Stepping-stone Intrusion with Evasions

Donoho [90] considered evasions which may defeat normal stepping-stone detections. These evasions include local jittering of packet arrival times and the addition of superfluous packets like chaff. Their method makes the assumption that the intruder can only tolerate the delay to some degree. Donoho used wavelets and similar multi-scale methods, and the results show that they can separate the masked correlations by jittering or chaff from long-term packet streams which have the correlation remains.

To deal with a high chaff rate, Kuo and Huang [91] [92] proposed an effective algorithm for encrypted stepping-stone detection by using a mapping-based detection method. The algorithm [91] is able to rule out the independent (Normal) pairs and flag the stepping-stone pairs for examination. By using the order preserving mapping, the algorithm can run in linear time. This algorithm DMIM [91] guarantees mapping whenever there exists such a mapping. Their experimental results show that the DMIM algorithm is efficient and highly accurate to detect the stepping-stone pairs. To further increase the accuracy of detecting stepping stones when network traffic is corrupted or injected with chaff packets, the authors improved their algorithm and used several factors like maximum delay, number of incoming packets, and the acceptable mismatched rate to measure their methods. The experimental results showed their algorithm can detect abnormal connections correctly even with a chaff rate of up to 400% on incoming and outgoing streams.

Besides Kuo's work [91][92] on detecting stepping stones with the chaff perturbation, Yang, Lee, and Huang [93] proposed another method to detect stepping-stone intrusions based on random walk theory. Their theoretical analysis shows that the proposed method is more effective than Blum's approach [94] in terms of resisting intruders' chaff perturbation.

Zhang [95] also proposed their method to detect stepping-stone connections with delay and chaff perturbations introduced. Under the assumption that the connections are long enough, their method can detect stepping-stone connections with limited and independent delay and chaff perturbations effectively.

Most of the existing malicious stepping-stone chain detection research has been concentrated on detecting intermediate stepping-stone hosts. A general method for detecting stepping stones is to identify traffic characteristics which are highly correlated across stepping-stone connection pairs [31]. There are some potential characteristics which include connection contents of unencrypted connection, inter-packet time gap, ON/OFF patterns of activity, and traffic volume or rate. These features can be combined to detect stepping stones. However, these stepping-stone detection methods do not solve the problem completely. First, most of the benefits of the detection go to the host at the end of the chain (the victim host), an unknown third party to the monitoring site. Secondly, the stepping stone is only able to gauge the maliciousness of a connection by the number of downstream hops it detects [87] instead of the complete chain. If the stepping stone is very near the victim in the connection chain, it may not be unable to distinguish a malicious chain from a benign connection.

2.4 Tools Used to Tamper with Network Packets for Evasions

We already introduced some network traffic capturing and analysis tools in the previous section. We have reviewed feasible ways to intercept network traffic and decode it into useful formats. To invalidate stepping-stone detection algorithms, intentionally tampering with one part of the traffic is a feasible solution which can be used by hackers. To modify a network packet, in addition to intercepting packets, one needs a store and forward way to get the packet, change it, and send it out. In this section, we will introduce some tools can be practically used to tamper with network packets.

To tamper with network packets of client-based and server-based communication, one often makes independent connections with the client and the server, then relays messages between them. Furthermore, by using well developed tools, one can gain the ability to modify the traffic stream between the client and the server on the fly.

There are some widely used network traffic proxy tools can fulfill the store-and-forward functionality, such as Mallory, MiTM Proxy, Burp Proxy, Scapy, etc.

[96][97][98][99][100][101]. Those proxy tools are designed to analyze the communication of client-server-based applications. This can help researchers to easily find more bugs in all types of client-server applications. However, one can also use these tools to tamper with a network packet for the purpose of conducting an evasion attack.

Burp Proxy [98] is an intercepting proxy server for security testing of web applications. It stands in the middle between your browser and the target application which allows you to intercept and modify all HTTP/S traffic passing in both directions. Burp allows you to apply rules to determine which requests and responses are intercepted for manual testing, and also let you define rules to automatically modify requests and responses without manual intervention. However, Burp only supports HTTP and HTTPS protocols. Besides, it is not extensible to other protocols.

Scapy [101] is a Python-based packet manipulation program for computer networks. It can forge, decode, capture, and send packets of most protocols. It can also be used to scan, trace, probe, test, or even attack a targeted network. Scapy provides a Python interface based on Libpcap or Winpcap on Windows. It can be used as an interactive tool, or as the API for other Python programs.

Matasano's Port Forwarding Interceptor (PFI) [99] is a proxy tool that can be used to deal with TCP streams. It saves time from writing codes to deal with raw TCP connections which Burp can't help with. It intercepts TCP connections and allows you to modify the traffic manually. Besides, you can extend it with other TCP based protocols. However, PFI doesn't support a pre-set rule-based mechanism to let you modify your TCP stream in a programmatic way.

Cain & Abel [100] was a password recovery tool for Microsoft Operating Systems. Its main purpose was to perform the simplified recovery of passwords and credentials from various sources; however it also supports some other utilities in recent versions. The latest version of Cain & Abel contains new functions like man-in-the-middle (MiTM) attacks and APR (ARP Poison Routing) attacks. Its sniffer supports HTTP, HTTPS, and SSH protocols. However, it can only capture these traffics and perform several authentication attacks; you can't use it to modify the traffic, nor extend it to support other protocols.

The Middler [102] is a standard tool to demonstrate protocol middling attacks. It allows you to intercept and modify the traffic in a programmatic way to the HTTP protocol, while you can't manually modify the stream on the fly. The plugin system of the Middler gives the flexibility and extensibility to the HTTP protocol. However, you can't add other protocols to it.

Ettercap [103] is a network traffic tampering tool for LAN. It can sniff live connections, filter contents on the fly and support dissection of multiple protocols. It supports most common platforms, such as Linux, Windows, Solaris, and Mac OS X. Ettercap is well supported in extensibility, and it allows traffic streams to be modified in a programmatic way. Unfortunately, it doesn't support manual traffic stream modification.

Paros [104] and WebScarab [105] are two Java-based tools used to analyze and evaluate web applications which communicate by HTTP and HTTPS protocols. These two perform as an intercepting proxy, allowing you to intercept and modify requests and responses sending between the client and the server. They don't support any protocol other than HTTP and HTTPS, but you can enrich their functions by implementing plugins in a programmatic way.

There are some other tools, such as Netsed [106] and Squid [107] which either only support the HTTP protocol or just allow the traffic stream modification in a programmatic way. Neither of those tools introduced above supports as many features as Mallory does in terms of extensibility, multiple protocols supporting, and manual or programmatic traffic modification.

Mallory is a Python based tool implemented as a transparent TCP and UDP proxy, which can perform the task of intercepting and modifying network traffic streams between clients and servers. It integrates many features of existing tools, and focuses on the transport and application layer aspects of MiTM. Mallory is easy to use, it doesn't require pre-configuration, and it has very good extensibility. Currently, it's capable of intercepting several application layer protocols, such as HTTP, HTTPS, SSH, DNS and SSL, and injecting data into connections on the fly. Mallory can be a good network stream debugging tool to analyze malware and network-based applications.

To use Mallory implementing MiTM, one needs to make Mallory the gateway of one's application under testing first. All traffic from the application to the destination server will go through Mallory in a transparent way. This procedure can be illustrated by Figure 2.6. To illustrate how much one can tamper with the original packets during a network connection session, we use an example which tampers HTTP traffic on the fly by modifying requesting and responses between the web browser and the server. In this example, we flip all figures on the web page sent from the Flickr server to the user. The rendered web page on the user's browser has all figures flipped as shown in Figure 2.7.

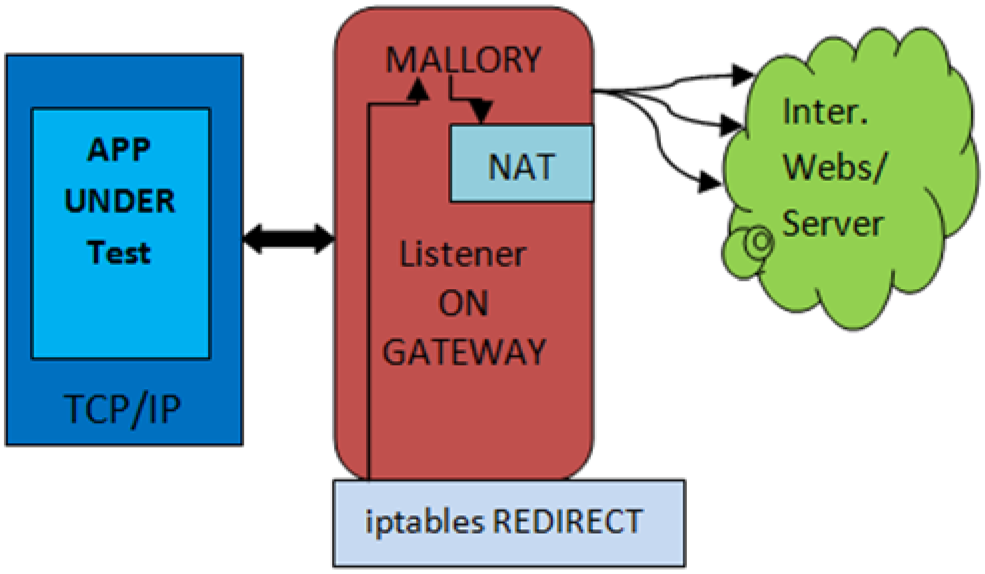
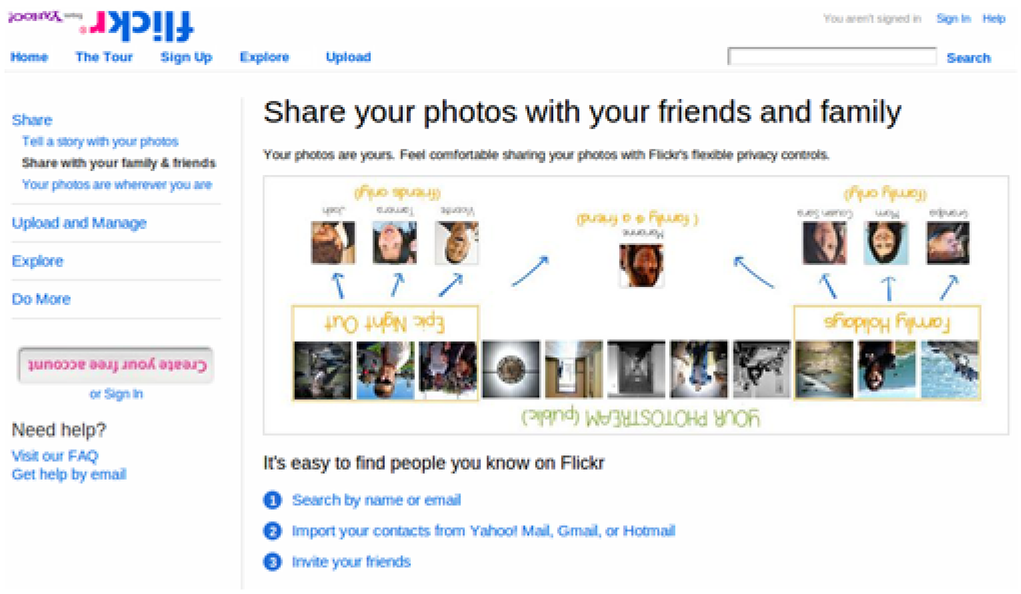


Figure 2.6: Using Mallory to Tamper with Network Traffic

2.5 Problems Remained to be Solved

Most research already done by others focused on intermediate host-based stepping-stone detections. Previous stepping-stone detection methods relied upon the time difference between the attacker sending data downstream, and a response from the server passing back upstream ("reply echo time"), which forms a "closed loop" along with the downstream connection chain. Yang and Huang [87] used the closed loop to detect a downstream connection chain in real time. Their research is trying to protect the host at the end of the connection chain, which is mostly an unknown



Share your photos with your friends and family

Your photos are yours. Feel comfortable sharing your photos with Flickr's flexible privacy controls.

Need help?
Visit our FAQ
Get help by email

It's easy to find people you know on Flickr

1. Search by name or email
2. Import your contacts from Yahoo! Mail, Gmail, or Hotmail
3. Invite your friends

Figure 2.7: Flipping Figures of HTTP Traffic by Mallory

third party. Hence, it is meaningful to carry on research on detecting intruders and preventing systems of being compromised based on the victim host.

Detecting a malicious connection chain is much more challenging from a victim's perspective than at a stepping-stone host. Victim-based detection has many difficulties of its own. There is no straightforward method of estimating the full Round-Trip Time (RTT) for the length of the connection chain. This is primarily due to the nature of tunneled SSH connections, and the fact that SSH is an interactive terminal session. This means that over the course of an SSH session, there is no point in time at which the server sends data to the client and the client's machine automatically sends a reply back to the server.

For interactive terminal sessions, the client-response would come from the nearest

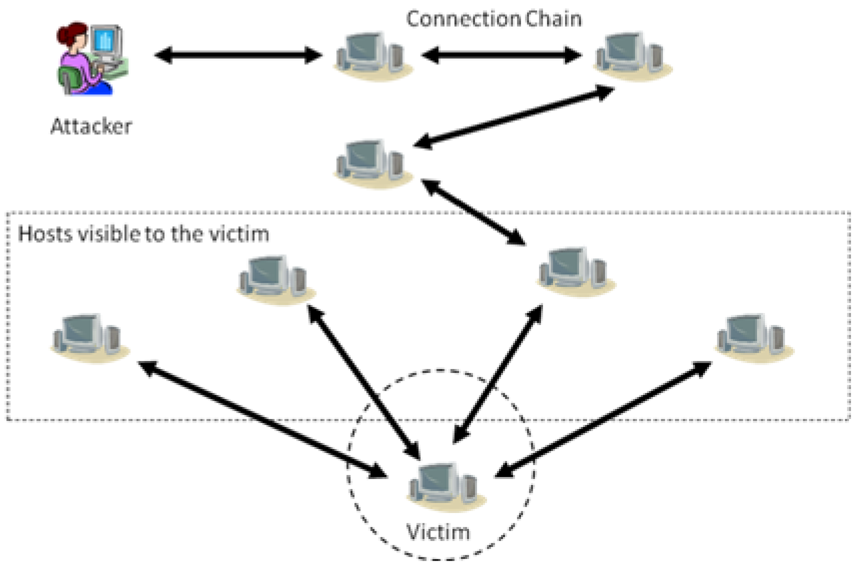


Figure 2.8: Stepping-Stone Attack with a Long Connection Chain

host in the connection chain. From Figure 2.8, one can see that all visible hosts to the victim are those directly connected to the victim machine which called the nearest neighbors. Other hosts on the long connection are not visible from the victim's host. Note that information passed between each pair of hosts is enclosed in a different packet. To find out if there is a long connection chain based on the view of the victim's host, clearly it needs to get a big picture of the connection chain out of the nearest neighbors area.

To detect intruders who always intend to evade being caught, it's reasonable to find if there are long connection chains being used. When a long connection chain with many stepping-stone hosts is being used, it's very suspicious that this chain is used by intruders. In the following two chapters, we solve this problem by implementing intrusion detection algorithms based on the victim host, which is at the end of the connection chain, as shown in Figure 2.9. By separating long connection chain from short connection chains, the suspicious intrusion connection chain can be detected and prevented.

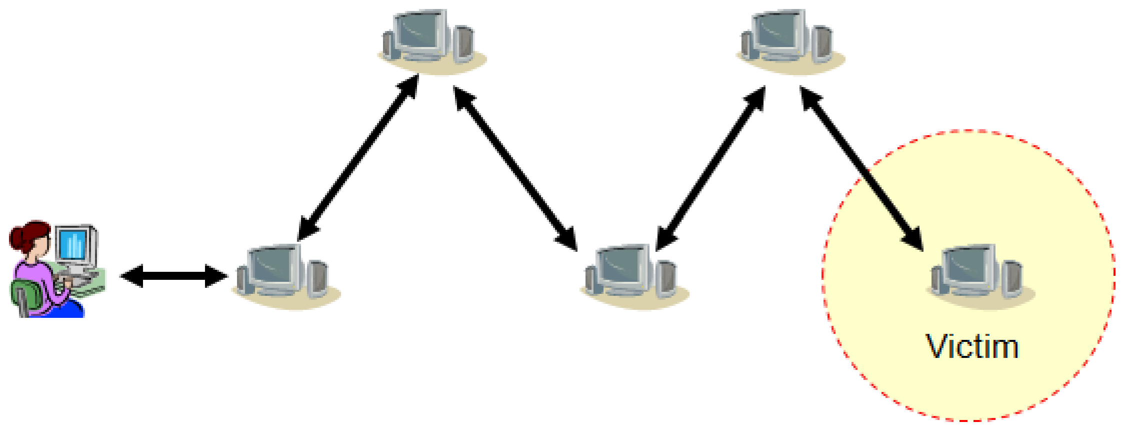


Figure 2.9: Detecting Intruders at the Targeted Host

Chapter 3

Detecting Intruders at the Targeted Host: A Nearest Neighbor Approach

There is no valid reason for a legitimate user to use a long connection chain for remote login to a computer host. If we can discriminate long connection chains from short connection chains, then we can identify potential intruders from normal users. Fortunately, the long chain will leave some trace of information for us to explore.

The work described in this chapter has been previously published: Wei Ding, M.J. Hausknecht, S. Huang, and Z. Riggle. Detecting stepping-stone intruders with long connection chains. In Proc. of the 5th Intl. Conference on Information Assurance and Security, pages 665-669, 2009.

Our approach is to examine the packets at the victim host and determine whether a connection is long or short.

3.1 Measuring Round-trip Time

A packet Round-Trip Time (RTT) for a connection is the sum of processing delay, queuing delay, transmission delay, and propagation delay of the connection [108]. It's not hard to detect the downstream round trip time. To measure the downstream round trip time at the originating host, we need to measure the time lapse between sending a request packet out to the downstream end and getting the reply packet back. As illustrated in Figure 3.1, a packet is sent from Host 1 to Host 4, via Host 2 and Host 3. After Host 4 gets this request packet, it sends a reply packet back to Host 1 via Host 2 and Host 3. At Host 1, the time difference between sending the request packet and receiving the reply packet is defined as the downstream RTT. There are various algorithms to match "echo" packets with the "send" packets with high confidence [87].

RTT of a connection chain may vary depending on network traffic speed and CPU load at intermediate hosts. The measurement of RTT can be done by using socket programming techniques in practice. The basic idea of this method is to establish a connection between two hosts first. Then based on one host called client, we can send a packet to another host called server through designated routes. As soon as the server receives the packet, it sends another packet (as the response to the packet it received) back to the client. The client needs to measure the time difference

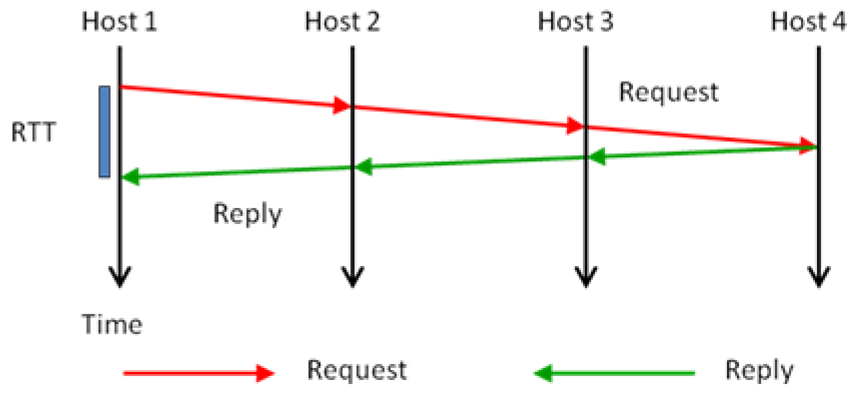


Figure 3.1: Round-trip Time (RTT) as Measured at Host 1 of a 4-hop Chain

between its sending and the receiving of the response packet, then the RTT can be calculated. An example of socket programs including client and server parts is shown in the Appendix A.

Our approach of estimating the RTT is to take an echo packet sent back and the next packet received as a round-trip which is called upstream round-trip time (uRTT). The difficulty of the task is to determine the upstream round-trip time with some certainty, even though it is not as accurate as the downstream one. This time difference usually represents the full round-trip time plus the time it takes the user to generate the next packet (via keystroke).

$$\begin{aligned} \text{Time Diff} = \quad & \text{Time to send an Echo packet} + \text{User Delay Time} + \\ & \text{Time to send the next packet} \\ \approx \quad & \text{Round-trip Time} + \text{User Delay Time}. \end{aligned}$$

In Figure 3.2, as soon as Host 4 receives the first request packet, it sends the reply packet back to Host 1. After an unknown delay at Host 1, it sends the second request packet to Host 4 via the same route. Then Host 4 sends another reply packet back when it received this new request packet. The echo time of the first reply and the send time of the second request combine to form an approximate round-trip time. Clearly this is not the most accurate way to measure the RTT, but this is what is available and we would like to see if this will lead to the identification of long connection chains.

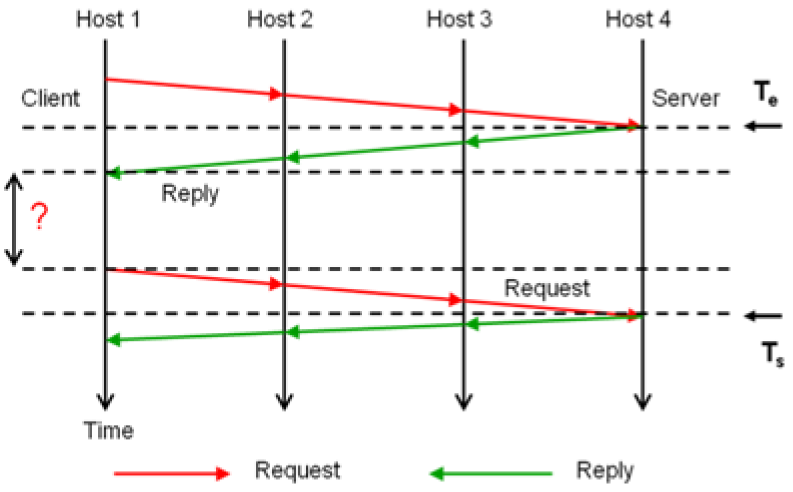


Figure 3.2: Estimating Upstream Round-trip Time at the End of the Chain

Therefore, if the time it takes for the user to press the next key, i.e., the user delay time, is subtracted from the time difference, the round-trip time remains. At the beginning, we tried to find the minimum upstream round-trip time to approximate the real RTT. But experiments showed that the measured minimum upstream round-trip time can be less than the actual RTT which means packet cross-over happened. This is illustrated in Figure 3.3. Figure 3.3 illustrates two cross-overs in a connection chain. Before the first reply packet sent from the host arrived at Host 1, Host 1 already sent the second request packet to Host 4. At the receiving end, the time difference of the first reply and the second request is very small indicated as "Gap 1" in the figure. These cross-over packets make it more challenging to estimate the real round trip time. In this situation, there is no user delay to subtract from uRTT, in order to get the left RTT.

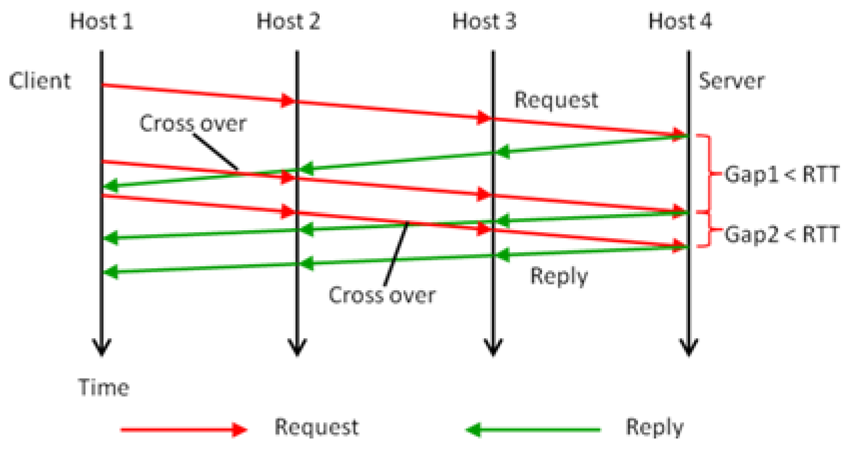


Figure 3.3: Example of Packet Cross-overs

Our approach seeks to estimate the proper user delay time in order to find the full round trip time. More specifically, our algorithm finds the time difference between the client's enter keystroke to submit a command and the client's next keystroke to start a new command. It then analyzes this time gap based on several other features of the connection and attempts to determine the full round-trip time and ultimately the length of the client's connection chain.

3.2 Separating Packet Gaps in SSH Connections

It is not an easy task to estimate the user delay since the time gap between typing varies significantly even for the same user. Furthermore, if cross-over happens in the middle, there would be no user delay to subtract. We chose to analyze not all but some special gaps. We selected the time gap between the end of a command and the beginning of the next command which we call "inter-command gap" as shown in 3.4(a) because of the expectation that a user will often need to see the command's output before starting to type the next command.

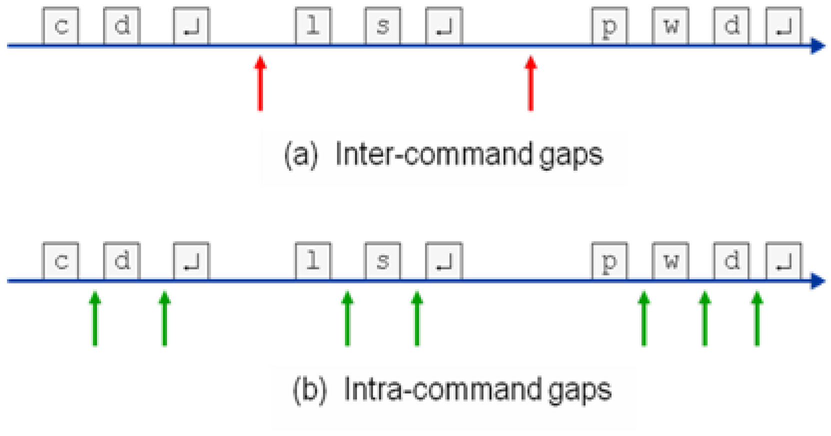


Figure 3.4: Two Types of Gaps between Packets

For example, it is often necessary to see the file listing returned by a directory listing ("*ls*") command before selecting which directory to change into ("*cd*"). Besides, people always want to use ("*ls*") command to list all contents of the directory after using ("*cd*") to get into this directory. Sometimes people don't stop to think when using these two commands consecutively. Other possible time gaps such as the gap between individual keystrokes would have been equally valid for analysis; however, our algorithm is designed to ignore these gaps because of the possibility that clients, when using a long connection chain, may not wait for each character to appear on their terminal before typing the next. This would result in no user delay. Imagine that a packet is echoed back from one end of the chain while a packet from the originating host is already on the way toward the end of the chain. The two packets may cross each other somewhere in the middle of the connection.

While it might seem that user delay times would be dramatically larger than full round trip times, we found that for long connection chains, the full round trip times can be on par with the user delay times. For example, using a ten hop chain of reasonably fast hosts (40 milliseconds round trip time), there would be a 400 millisecond delay which is approximately equal to the delay of certain pairs of commands for many of our recorded users.

Our algorithm seeks to detect a long connection chain by finding the full round-trip time. This is found by subtracting the estimated user delay time from the time gap between the enter key-press and the next keystroke. Shown in Figure 3.4(b), the user delay time is estimated to be the client's average typing speed which can be estimated by using keystroke gaps inside one command which we call "intra-command gap." While this estimation does not account for the time the user might spend reading/thinking before resuming typing, our approach seeks to target those pairs of commands which require minimal cognitive delay. By subtracting the estimated user delay from the total gap time, what is left is the estimated full round-trip time.

Unavoidably, some users will wait long periods of time between certain pairs of commands. For this reason, estimated full round-trip times greater than a threshold value of two seconds were discarded. In order to accurately detect measures such as the user typing speed, and the occurrence of a new command, it was necessary to examine certain characteristics of an SSH session. Despite the fact that all of the data in an SSH session are encrypted, much information can still be gathered simply by monitoring the quantity, characteristics, and timing of packets. Session characteristics which allow the detection of keystrokes, new commands, and nearest hop round-trip time will all be discussed. Note that all the data about user delay is taken at the victim host, not at the source of the chain.

3.3 Estimating Keystroke Gaps

The TCP header of all packets in TCP sessions (SSH packets included) holds information about, among other things, the source port, destination port, sequence number, and acknowledgment number of the packet among other pieces of information. We can use these header details, especially the latter two numbers to detect nearly all of the desired SSH session characteristics.

In a normal flow of information at the start of a TCP session, the client will connect to the server with a SYN packet, telling the server its sequence number. The server will then respond with a SYN ACK packet, providing its own sequence number and setting its acknowledgment equal to the client's next sequence number. The client will then (a) piggyback the acknowledgment (ACK) of the received packet within a reply packet, or (b) respond with an ACK packet if no reply is ready within a certain period of time. The sequence number of the request packet will be the same as the acknowledgment number of the corresponding reply packet.

After the three way TCP handshake has been completed, the following packets would be packets sent between the client and the server. In the context of a SSH connection each character typed by the client generates a series of packets, each request packet and its corresponding reply packet can be matched by request packet's ACK and reply packet's sequence number which should always be the same. If there are multiple reply packets for a single request packet, the sequence number of those reply packets should be the same. In this way, we are able to detect each keystroke generated request packet and its corresponding rely packets. Additionally, by analyzing the time gaps between individual keystrokes, we can find a user's average typing speed.

New command detection is slightly more difficult and less reliable than keystroke detection. The intuition behind command detection is that after the client enters a command, the amount of data that is sent back will be large enough to exceed one block size for the encryption algorithm being used. We observed that even an empty enter stroke can get two response packets back in tested Linux systems. Note that the encrypted packet length in bytes may vary among these values like 48, 64, 80, 128, etc., so all data will always be padded to take up an encrypted packed size. For example, an individual character can be padded and encrypted to a request packet with size of 80 bytes. Data larger than the normal block size, such as a directory listing, will be padded to the next block size, and will thus be reported as being a packet with a larger payload.

Additionally, these post-command packets exhibit the characteristic that each packet's sequence number is the same as the last packet's acknowledgment number. Because of this, we are able to distinguish the series of packets denoting the return of a command from the series of packets generated has the user continued typing new characters. The number of packets generated in response to a new command can be more than one. For example, "*ls /usr/bin/*" can generate 290 post command packets in our Unix system while simply pressing enter at an empty prompt may only generate two. Network latency and user typing speed can also cause the number of post command packets to vary. Our approach flagged a new command after observing two consecutive post command packets. In practice we found this to be quite accurate and reliable. Our sample program used to analyze keystrokes is shown in Appendix B.

3.4 Long Connection Chain Detection Algorithm

Since we can't calculate the downstream RTT at the end of the chain, we can't use it to indicate how long the current chain is. Instead, we try to measure the upstream round-trip time at the end of the chain, and use it to discriminate long connection chains from short ones.

3.4.1 Estimated Upstream RTT

We define a measure, called estimated upstream round-trip time, to distinguish a long chain from a short one. We start out with the time gap between a response packet and the next request packet received at the victim's host. We then subtract the average user delay (as defined earlier) from this quantity. Our objective is to see if the "estimated upstream round-trip time" can be used to differentiate the chain length. By choosing a representative value for each connection chain based on estimated uRTT, we want use it to separate long chains from short chains. This procedure is described by the algorithm shown below.

Algorithm: uRTT

Given a session of one connection from time T, calculate a representative value:

Step 1: *For each inter-command gap, compute* $gap(i) = T_s(i) - T_e(i)$*,*

where T_e is the time stamp of a response packet, and T_s is the time stamp of the following request packet.

Step 2: *Estimate average user delay* d *by averaging keystroke gaps inside commands.*

Step 3: *Calculate the estimated upstream round trip time gaps for each connection chain as* $uRTT = gap(i) - d$*.*

Step 4: *Calculate the representative value* $m = median(uRTT(i))$*, i=1,...,n.*

3.4.2 Nearest Hop Round-trip Time (nRTT) Detection

We already know from the previous chapter that all visible hosts to the victim are those directly connected to the victim machine, which is shown in Figure 2.8. The round-trip time to the nearest hop in the connection chain can be calculated due to the nature of the TCP session or by sending a ping. After each data packet sent by the client down to the server, the server replies with a response packet to the last host in the connection chain very shortly after getting the request packet. The last host then automatically sends an acknowledgment back to the server, completing the sequence of packets.

Shown in Table 3.1, packets between client and server can be matched by the Sequence number and the Acknowledgement number. After client sending the request packet to server, the response packet from the server will make the ACK number of the request packet to be its sequence number. Also, the ACK packet from client to server will have its sequence number the same as the ACK number of the response packet. After we match the request packet, response packet and the ACK packet by this way, the round trip time to the nearest host can then be found by examining the time difference between the server sending the response packet to the last host in the chain and the nearest host replying with its own acknowledgment. Since the nearest host upstream from the victim machine is communicating with the victim, the IP address can be found in the packet header. Thus, this is the only host on the chain that directly communicates with the victim host.

This nearest hop round-trip time does not necessarily reflect the actual RTT or

Table 3.1: An Example of Matching Packets with Sequence and Acknowledgment Numbers to Compute RTT

Time	**Direction**	**Packet Type**	**Seq. Num.**	**Ack. Num.**
9.927634	Client->Server	Request	3121	4593
9.927742	Server->Client	Response	4593	3201
9.956932	Client->Server	ACK	3201	4641

length from the origin of the connection to the victim when there is a long connection chain used. But it can help when we have an estimation of the RTT for the whole chain. It's useful to take the ratio of this nearest hop RTT and the estimated uRTT to estimate how many intermediate hops does this connection chain have, though it's maybe not an accurate way to measure this length.

3.4.3 Estimating Connection Chain Length

By modifying the uRTT algorithm a little, an estimated length of the connection chain can be calculated as shown in this algorithm following.

Algorithm: Length of the Connection Chain

Given a session of one connection:

Step 1: *Use algorithm of uRTT to calculate the representative value m.*

Step 2: *Calculate the nRTT of the connection chain.*

Step 3: *Calculate the length of the connection chain $L = m/nRTT$.*

When the RTT between each continuous host pair is close to each other, and the estimated uRTT can closely reflect the real RTT of the whole chain, this estimation of the chain length can be close to the real length of the connection chain. But under real circumstances, neither of these two conditions can be satisfied well.

3.4.4 Classifying Long Chains from Short Chains

In practice, initially we can compare the uRTT and the nRTT to see if the uRTT is much larger than the nRTT. If the uRTT of a chain is much larger than the nRTT, it's very suspicious that there are many intermediate hops before the connection getting to the nearest neighbor host. Theoretically, if the uRTT is close to the downstream RTT, and the nRTT is a good representation of the network, the ratio between uRTT and nRTT should be a good estimator of the chain length. However, in practice it's not true. Normally, RTTs between each neighbor hosts are not close to each other, and the nRTT can vary a lot. Hence, the ratio between uRTT and nRTT is not a good indicator for the length of the whole chain. It turns out that uRTT is a better measure to separate long connection chains from short connection chains. Hence, based on the algorithm of uRTT, we have the classification algorithm to determine long chains from short chains as given below:

Algorithm: Classification of Long Chains from Short Chains

Given a session of one connection from time T, find if this is a long connection chain:

Step 1: *For each inter-command gap, compute gap$(i) = T_s(i) - T_e(i)$,*

where T_e is the time stamp of a response packet, and T_s is the time stamp of the following request packet.

Step 2: *Estimate average user delay d by averaging keystroke gaps inside commands.*

Step 3: *Calculate the estimated upstream round trip time gaps for each connection chain as* $uRTT = gap(i) - d$.

Step 4: *Calculate* $m = median(uRTT(i))$, *i=1,...,n.*

Step 5: *For a predefined threshold t,*

if $(m > t)$ *then classify the session as a long connection chain*

else classify the session as a short connection chain.

We will discuss how to determine the threshold t in the following section.

3.5 Validation and Performance

3.5.1 Experiments Setup

Most of network traffic packets capturing programs are using the pcap (packet capture) library [109]. In the field of computer network administration, pcap consists of an application programming interface for capturing network traffic. Unix-like systems implement pcap in the libpcap library [109]. Libpcap provides an API to capture packets travelling over the network, and supports saving captured packets to a file for further analysis.

There are three major parts in our network packets capturing and analyzing programs, including packet capturing, session analyzing, and gnuplot exporting. The packet capturing program is written based on libpcap library with the main program "realtime". The "realtime" program allows for the collection and analysis of packet data in real time. These network packets data are recorded in a human readable way and stored in specified log files. The log file format is shown below as:

```
♯LOG TIME Thu Jul 24 14:29:40 2008
1216927631.371089 3790638033 0 131.215.17.73 42860 → 129.7.243.14 22 0 2
1216927631.371340 1448313446 3807415249 129.7.243.14 22 → 131.215.17.73 42860 0 18
1216927631.417100 3807415249 1465090662 131.215.17.73 42860 → 129.7.243.14 22 0 16
1216927631.424800 1465090662 3807415249 129.7.243.14 22 → 131.215.17.73 42860 20 24
1216927631.464867 3807415249 1800634982 131.215.17.73 42860 → 129.7.243.14 22 0 16
1216927631.465011 3807415249 1800634982 131.215.17.73 42860 → 129.7.243.14 22 31 24
1216927631.465590 1800634982 32607185 129.7.243.14 22 → 131.215.17.73 42860 0 16
1216927631.470028 1800634982 32607185 129.7.243.14 22 → 131.215.17.73 42860 784 24
```

In the recorded log file, it's easy to check the timestamp, tcp sequence number, tcp acknowledge number, source IP address and port, destination IP address and port, packet size, and tcp flags. By using different program flags, we can choose which network interface to be listened to, specify host IP address for collecting packets, enable different packet filters, change port to be listened, and set path of log files. A usage example of "realtime" program is "*realtime -i eth0 -p 22 -f 1 -a 3 -l data.log*". Since this program performs packet sniffing, it normally requires super-user privileges under Unix-like systems.

The first function of the session analyzer is to calculate the time difference between the current packet and the last packet. It can update time gaps in real time when the proper flag is set. Besides, the session analyzer can also calculate the average of the time gaps between each analyzable packet and output the standard deviation of them. Furthermore, there are some more complicated filters that can be set to work in conjunction with the analyzer. The responsibility of the filter is to decide how the analyzer should interpret each packet. The filter indicates the appropriate action by passing an integer to the analyzer which indicates different filter types. The "nearest hop" filter is designed to provide information on the packets which are direct communications between the host and the nearest hop machine. It ensures that no analysis is performed on packets which are influenced by user delay. This is useful for finding the RTT to the nearest hop. The "user delay" filter is only interested in the packets which are subject to user delay. It only allows analysis when the user types a character or presses the enter key. The "command" filter passes the first packet of each new command entered by the user. It attempts to detect new commands by listening for a long sequence of data packets which indicate a larger amount of data being transferred than simply a character. Some commands will generate more data transmission than others which will cause the filter to be unable to detect new commands with perfect accuracy. The "stream" filter attempts to identify streams of characters as when a user will hold down the backspace key for example. It does this by passing those packets in which the current sequence number matches the last ack number and the current ack number matches the last sequence number. Besides these, there are filters to separate encrypted incoming and outgoing packets.

Once we have the results from the algorithms, it often helps to be able to do some kind of filtering/manipulation on the results (for example, smoothing, chopping off the first half of the results, etc.). For this functionality, results filters were created. There are two possible ways to perform results manipulation: on a single result list, or a short/long chain comparison (useful, for example, in determining the high or low threshold of a set of connections). Result analyzers are specified with the -r flag on the command-line supported by the gnuplot exporting function. The "sort in order" filter (t) simply sorts the results in increasing order. The "smooth" filter (a) takes a floating-point argument specifying the weight of new data. The algorithm used for smoothing simply takes a weighted average between the last returned value, and the newest value. The weight is determined by the floating-point argument. Note that smaller values increase the smoothing amount. As the smoothing argument approaches zero, the returned results will be more uniform. As the smoothing argument approaches infinity, the results will be closer to the original values. An example is shown below:

Original Data: [100 90 120]

Smoothing argument: 0.1

First result from algorithm: 100

First result returned: 100

Second result from algorithm: 90

*First and second result composite: 100 + (0.1 * 90) = 109*

Second result returned: 109 / (1 + .1) = 99

*Third result returned: [(99) + (0.1 * 120)] / (1 + 0.1) = 101*

Filtered result with arg (0.1): [100, 99, 101]

The "Chomp" filter (c) takes a floating-point argument specifying the percentage of the data that should be omitted. Positive values take data off the front of the results, and negative values take data off the end of the results. These processes are shown as:

Original Data: [1 2 3 4 5 6 7 8 9 10]

Filtered result with arg (0.1): [2 3 4 5 6 7 8 9 10]

Filtered result with arg (-0.2): [1 2 3 4 5 6 7 8]

The "Median" Filter (m) returns a list containing a single value, the median of the original results. The "Mode" Filter (d) takes the mode of the data, where the floating-point argument specifies how much of the overall data must fit inside of the band. An argument value of 0 will provide the true mode, which is not likely the desired result, as almost all of the results will be entirely unique. Specifying an argument (for example, 0.1) will repeatedly round all of the results up by one decimal level until the the mode value consists of the specified percentage of the results. A "representative value" filter (r) chops off the first half of the results, and gets the median. The "Criss Cross" Filter (i) creates one long array with all of the results of all of the long logs sorted in decreasing order. It also creates one short array consisting of all of the results of all of the short logs sorted in increasing order. The recommended use of this filter is with the representative value and gap filters. This filter is useful for visually determining where the long logs overlap the short logs. The "Separation Comparator" (g) is meant to find the lowest value of the long logs and the highest value of the short logs. These values are then plotted to the graph as horizontal lines. This functionality is useful for determining the gap between the high and low logs.

The gnuplot-export program [110] is used to interface with the "Gnuplot" graphing program. Gnuplot is a portable command-line driven graphing utility for Linux and many other platforms which can generate 2-D and 3-D plots of functions, data, and data fits. It's copyrighted but can be distributed freely. Based on Gnuplot, our gnuplot-export program is designed to review a previously recorded session log and apply a specific filter algorithm combination to the session, ultimately producing a data file and a script file which allow gnuplot to load and chart the data. The gnuplot-export program prepares the required external files to graphically plot analysis results by using the gnuplot program. By giving log files, we use "*gnuplot-export [args] log1 ... logn*" first. After running the program, a data file with the name specified by the ‘o’ flag will be created (default name “data.dat”). Additionally, a gnuplot script file named “auto.p” will also be present. Then we can use the Gnuplot program to interactively format the data into a figure by using the following sequence of commands:

```
gnuplot
load ‘auto.p’
♯ View the results
exit
```

Besides, we can use specified arguments to filter the result and output the graph directly. A detailed usage of our gnuplot-export program is shown below:

```
Usage: gnuplot-export [args] logfiles
Supported Arguments:
-a n      Log analyzer to use. Default ’0’
             0: No Analysis
             1: Time Diff
             2: Average
             3: Standard Deviation
             4: Uber (use Filter 0)
-d Disable key/legend
-e Export to svg file.
-f n      Session filter to use. Default ’0’
             0: No Filter
             1: Nearest Hop Filter
             2: User Delay Filter
             3: Encrypted In Filter
             4: Encrypted Out Filter
             5: Command Filter
-r        List of results filters (comma deliminated)
```

c: Chomp Results

d: Mode of results

i: Criss Cross

g: Gap analysis

m: Median Result Value

r: Representative Result Value

s: Smooth Results

t: Sort Results

-o path   Path to output file. Default 'data.dat'

-n Normalize xValues. Default '100'

-p Include points in graph.

-v Verbosity. Specify twice for extra-verbose.

More than one result filter can be specified, and the same result filter can be specified more than once. The order in which result filters are specified is the order in which they are executed. For example, "*gnuplot-export [args] -r c=0.1,t,c=0.3 [logfiles]*" will chop off the first 10% of the results, then sort the remainder, and then chop off 30% of the remaining sorted results.

A total of seven computers were used to build chains and collect data. The computers ran various versions of Linux and all were connected to the Internet via high speed connections. As shown in Figure 3.5, two computers located at the University of Houston were used to listen for incoming connections (victim) and to initiate connection chains (attacker). The other five computers were located in various regions of the U.S. and primarily served as intermediate hosts in the building of connection chains. To simulate long connection chains, sometimes these distant computers may be used more than once in a chain.

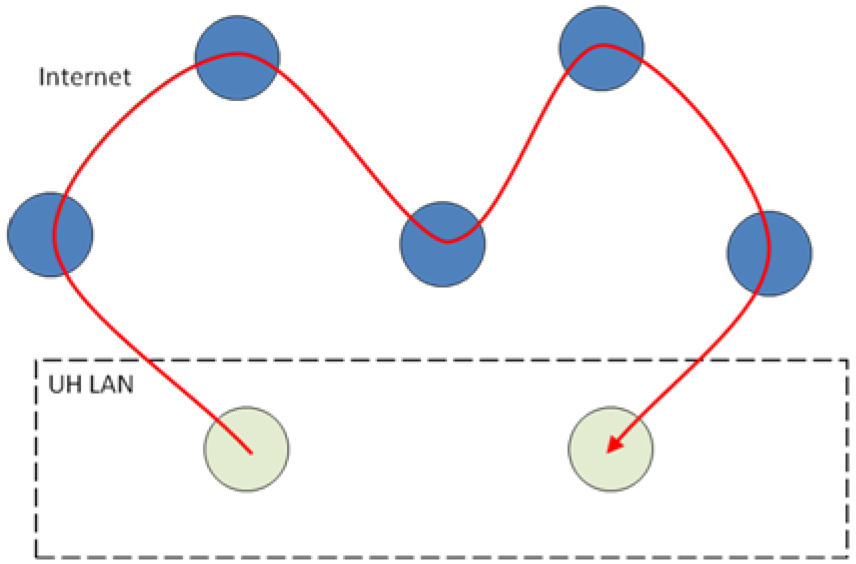


Figure 3.5: Experimental Connection Chains of 5 Hops

To increase the efficiency of our experiments, several connection establishment programs were developed in Python which can automatically establish a connection chain of variable length by using any number of routes and any number of hosts we already set in a configuration file.

We use public-key authentication among all hosts. The first step is to use the command "*ssh-keygen -t rsa*" to produce the public key for each host. Then the file " */.ssh/id_rsa.pub*" among all hosts is copied to one place. After that all these public key files of all hosts are combined to one file named "*authorized_keys2*", and this file is distributed into each host inside the directory of " */.ssh*". By finishing these steps, any host can be accessed by SSH without requiring the password authentication. By using a different chain name for each connection chain from the configuration file set before, we can use the "connect.py" program to make the SSH connection through all hosts sequenced in the chain automatically. An example of using the connection program is "*python connect.py -c myRoute -n 3*".

SSH session data was collected on a machine running the Slackware 12.1 Linux distribution. A total of 61 SSH sessions contributed by four different users were recorded and analyzed. These sessions consisted of the user activities of executing a series of commands through an SSH tunnel containing from zero to eight intermediate hops. Five different routes or chains of hosts were used. Each route employed a unique combination of hosts.

Gathering large amounts of data over various connection chains is very time-consuming. If each connection session can be recorded and replayed through other connection chains, it would be a better way to let us study the different results of various length of chains by using the same session data. To do this, a SSH session recording program was developed by using Python. The syntax is similar to "*connect.py*" program with the addition of a file to record to and play back from. The usage of this session recording program is shown below:

```
Usage: sshauto.py [options]

Options:

    -h, –help      show this help message and exit

    -c CHAIN        name of the chain (syntax: a,b,myChain)

    -f FILE         input/output files (syntax: file1,file2,file3)

    -l LENGTH       connection length

    -p              playback session

    -r              record session
```

When recording, the software records the exact moment that the user presses each individual key, and records that information. It also records the exact moment that it receives a prompt in the form of username, where the username is defined in the configuration file "*config.py*". When playing back the session, the software does not just play back the keystrokes with the same typing intervals. It is slightly more intelligent than that, and knows how long after a prompt a user waited to start typing a command.

Once a session is recorded, we can play it back through different connection chains with different lengths. This procedure is simply done by "*python sshauto.py -l 1, 3, 5-7 -c chainOne, chainTwo -f sess1,sess2 -p*". This command would play back both recorded sessions sess1 and sess2 through connection chains chainOne and chainTwo by using connection lengths 1, 3, 5, 6, 7 for a total number of 20 sessions in order. After triggering the packets capturing program, we can record several traffics coming in concurrently by playing back sessions recorded. Then these recorded log files will be processed for analyzing.

To simulate natural typing, users were given "objectives" to accomplish rather than a list of commands to type. Some example objectives were writing a short program, searching for text in a group of files, and creating a "tar" archive. Sessions generally tended to last around five minutes although some took up to fifteen. Clearly the users were slow at the beginning since they may be unclear of the intention of the instruction. After a few attempts, their speed stabilized. We discarded the first few data collections since they don't represent a true user behavior.

3.5.2 Comparison Between Short Chains and Long Chains

The first purpose of the experiment is to see if short chains and long chains exhibit any difference. Five length-1 short chains are used to compare with five length-8 long chains. Graphs of the uRTT of these short chains and long chains are shown in Figure 3.6.

Each marker represents a new command. The uRTT of each chain is determined by our algorithm described before. In each experiment, the number of packets collected varies from 31 to 100. We took the last 20 packets from each experiment since they are more stable than the beginning of the packet stream. As we expected, normally long chains have the uRTT larger than short chains. The top five series in the figure represent chains of length 8 and the bottom five represent chains of length 1. On average, the uRTT rating of a length-8 chain is about 200% higher than that of length-1 chains. To quantitatively check the difference between short chains and long chains, median and standard deviation of each chain is calculated. Results are summarized in Table 3.2.

3.5.3 Length Estimation

It is very difficult for us to correctly determine the length of a chain, because it's hard to give a very close estimation of the RTT for the whole chain, as well as the nRTT can vary a lot. By using the length estimation algorithm, we consider it's a success if the calculated length L is within a difference of one of the actual length of the chain. By further considering a success of length estimation within a difference

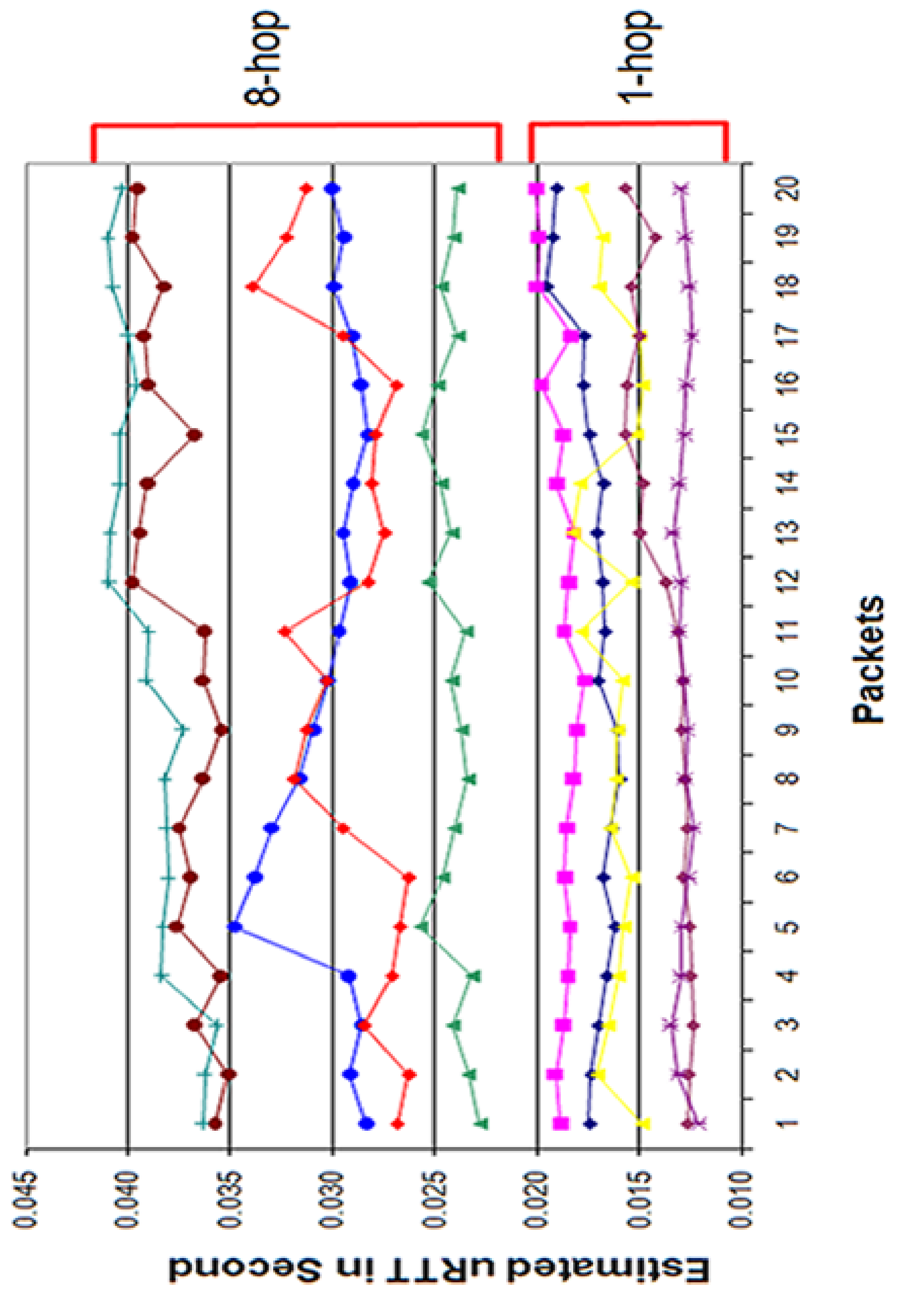


Figure 3.6: Estimated uRTT between Length-1 and Length-8 Chains

Table 3.2: Median and Standard Deviation of uRTTs (Short Chains vs. Long Chains)

Chain	Length	Median	Stdev
1	1	0.0171	0.0010
2	1	0.0187	0.0007
3	1	0.0161	0.0011
4	1	0.0130	0.0013
5	1	0.0129	0.0003
6	8	0.0373	0.0016
7	8	0.0391	0.0017
8	8	0.0295	0.0018
9	8	0.0242	0.0008
10	8	0.0284	0.0023

of two of the actual length of the chain, the success rate of most hops can increase in a certain percentage. These results are summarized in Figure 3.7.

3.5.4 Rank Checking

The length estimation of chains may not be accurate enough at most times. It is, however, much easier to determine which one of many chains is a long chain. In a real situation, the chance of having two intruders attacking the same host is low. The purpose of rank checking is to see if our algorithm can correctly identify the longer chain when given chains of various lengths. The objective of this analysis was to compare two sessions of different lengths and correctly rank the longer chain over the shorter. This ranking was accomplished by finding a representative value for each of the sessions, higher values indicating more suspicious connections. The median of the uRTT of each session is used to represent the session. By using this approach, each of the user sessions was compared to all other sessions of all other lengths. Each

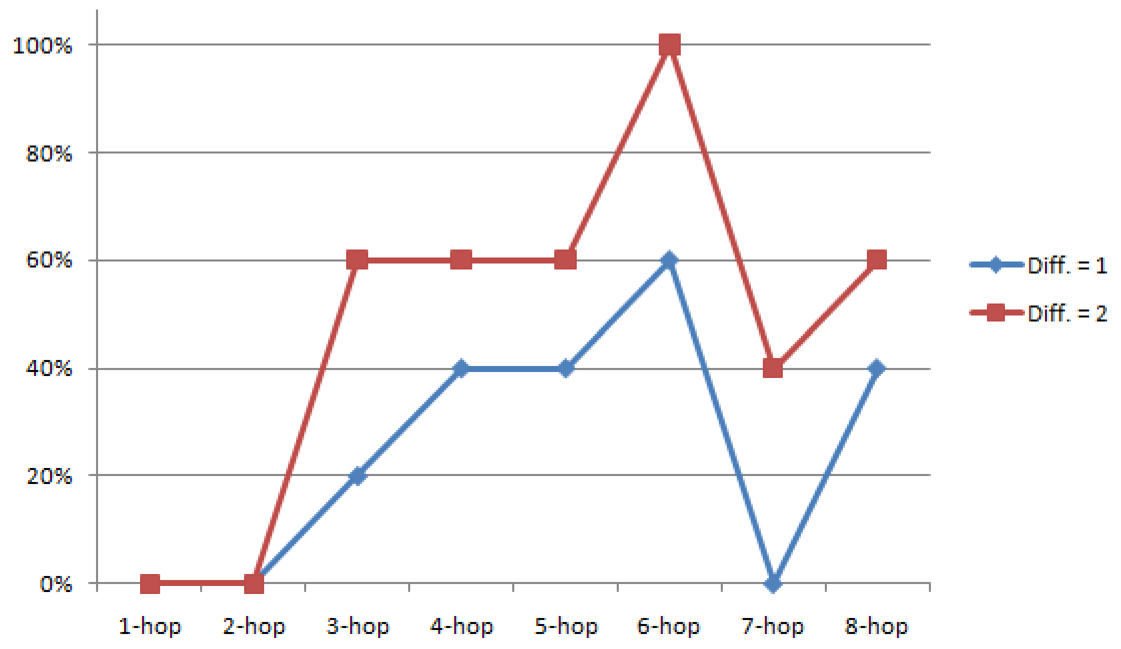


Figure 3.7: Length Estimation Success Rate within Difference one and two

column in Table 3.3 represents the number of intermediate hosts in the connection chain. Each cell contains the percentage of correctly ranked sessions. For example, the cell at Row 2 and Column 5 represents the success rate (84%) when one chain of 2 hops is compared with one chain of 5 hops. Each cell is the average of at least 5 experiments [111][112]. It might be noted that each of the zero-hop chains had a 100% success rate. This is because all of the zero chains (those with no intermediate stepping-stone hosts) were direct connections over the local area network.

The local connection is much faster than a chain connected to a host off campus. For this reason they are omitted from Figure 3.8 below. Also note that a chain of i hops consists of (i+1) SSH logins. Here we average the success rate of detection based on the difference of the chain of 1 to 7 hops. The figure indicates that the

Table 3.3: Ranking Percentage of Sessions from Length-1 to Length-8

Length	1	2	3	4	5	6	7	8
0	100	100	100	100	100	100	100	100
1	-	84	72	92	100	100	84	100
2	-	-	44	64	84	92	72	88
3	-	-	-	64	84	92	80	96
4	-	-	-	-	68	88	68	88
5	-	-	-	-	-	76	56	68
6	-	-	-	-	-	-	44	48
7	-	-	-	-	-	-	-	56

success rate for a difference of 3 is about 84%. Even if the difference is 1, the success rate is about 67%.

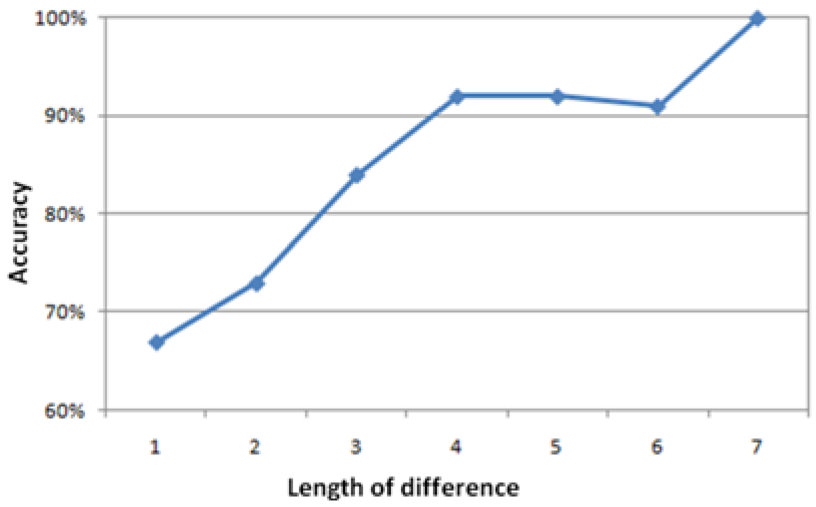


Figure 3.8: Percentage Correctly Ranked by Length Difference

3.5.5 Degree of Separation

Ranking analysis fails to indicate the degree of separation between two different incoming SSH sessions. In order to truly determine the success of our approach, it is necessary that the long chains are substantially separated (easily differentiable) from the short ones.

Short chains were (somewhat arbitrarily) defined as those connections containing from zero to two intermediate hops and long chains were defined as connections containing from six to eight intermediate hops. Representative points for all long chain sessions were plotted in decreasing order while representative points for all short chains were plotted in increasing order. The result is summarized in Figure 3.9. In this figure, we collected data from 15 long chains and 15 short chains. The data are arranged in decreasing order for the long chains and increasing order for the short chains.

By setting an absolute threshold, chains with representative value larger than the threshold are classified as long connection chains, and chains with representative value less than the threshold are classified as short connection chains. Ideally, we want to get a threshold yielding zero misclassification rate. Furthermore, if there is any misclassification, we want the misclassification rate minimized and these misclassified points close to the threshold as possible. We use two criteria to measure how good the threshold is. In practice, either we want to find an optimized threshold with *(FP+FN)* minimized, or we can minimize the sum of distance $\sum_P |p(i) - \tau|$ to find an optimized threshold. Here, P is the set of all misclassified data points. This

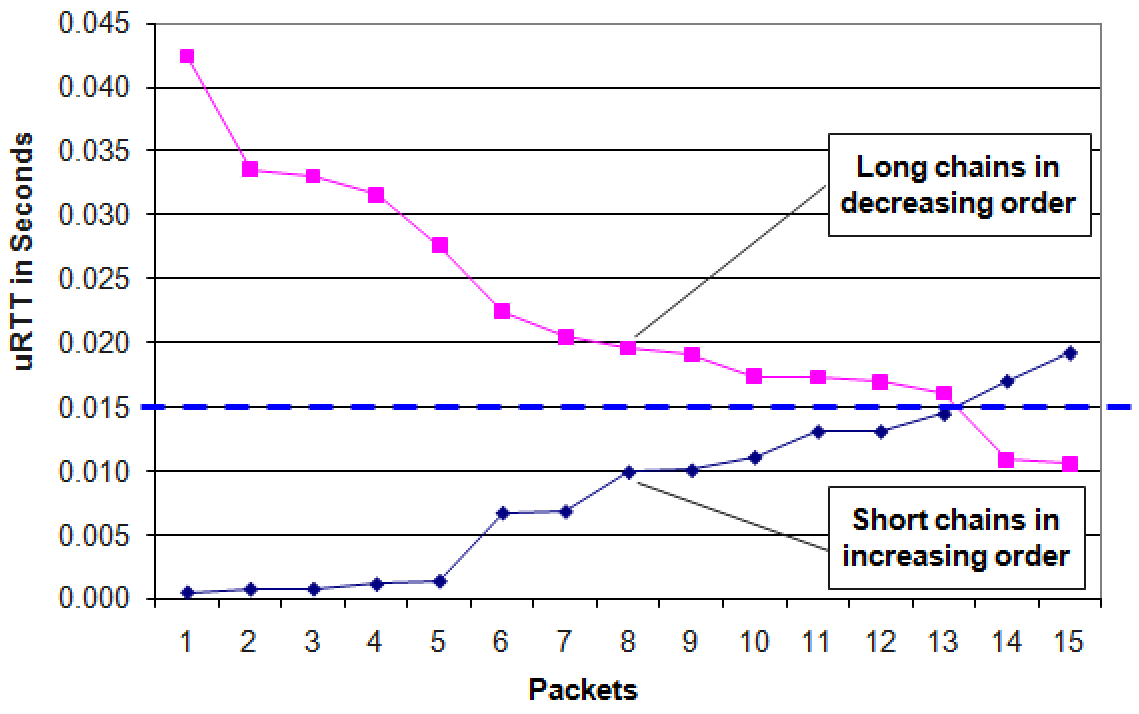


Figure 3.9: Comparison of uRTT of Short vs. Long Chains

process can be described as an optimization problem of τ subject to either the error rate $min_\tau\{FP + FN\}$ or the total error distance $min_\tau\{\sum_P |p(i) - \tau|\}$. To get the optimized τ, a search algorithm can be given as follows:

Algorithm: Threshold Calculation

Given N long connection sessions and M short connection sessions:

```
Step 1: For each short chain, compute the representative value $r(i), 1 \leq i \leq M$.
        For each long chain, compute the representative value $R(j), 1 \leq j \leq N$.
Step 2: Sort all $r(i)$ by increasing order. Sort all $R(j)$ by decreasing order.
Step 3: If $Min(R(j)) \geq Max(r(i))$, $\tau = (Min(R(j)) + Max(r(i)))/2$.
        Else if $Min(R(j)) \leq Max(r(i))$, pick and merge sort values into a new set
        $D = \{R(j)$ and $r(i) \mid R(j) \leq Max(r(i)),\ r(i) \geq Min(R(j))\}$ where
        $i \in [1, M],\ j \in [1, N]$.
Step 4: Set $d = |D|$. For each k, $2 \leq k \leq d$, $\tau = (D(k-1) + D(k))/2$,
        calculate misclassification error rate or error distance.
Step 5: Pick the $\tau$ when there is the minimal misclassification error rate or the
        minimized error distance.
```

Besides, we can assign false positive and false negative points different weights for computing the error rate or error distance when we consider false positive and false negative are unequally important. By using the algorithm above, we can get the threshold (dotted line) as 0.0153, and our algorithm would have correctly identified 13 out of 15 long chains and 13 out of 15 short chains, yielding a false positive rate of 13% and a false negative rate of 13%. However, the consequences are higher for every false negative than for each false positive detected. More harm is done in preventing one legitimate user from accessing the system than from allowing one illegitimate user access. For this reason, if a threshold was set just above all of the short chains (.0195), the algorithm would have correctly identified 8 of 15 long chains while keeping all short chains safe. This would be associated with a 47% false negative rate and a 0% false positive rate. Each host probably has to conduct experiments to determine the best threshold value to separate long and short chains. The system administrator may have to determine the tolerable level of false positive and false negative.

3.6 Conclusion

Attacking through stepping stones is a widely used technique by network attackers to attain anonymity and prevent to be traced back. To reinforce the safety of our systems, a lot of research in stepping-stone detection has been done. However, previous research focused on intermediate host-based stepping-stone detections, which is mostly trying to protect an unknown third party host at the end of the connection chain. It is meaningful to carry out research on detecting intruders and preventing systems of being compromised based on the victim host. We have proposed a new approach to detecting long SSH connection chains at the victim host. Our approach differs from previous work in that previous methods could only detect long downstream connections from the perspective of a stepping stone. Our method of detection centers around analyzing the delay between the time a user presses enter to finish a command and the time that the user types the next character. Taking into account the user's typing speed, it is possible to estimate if the user is connected through a long or a short chain. Our experiment results show that 86% of long chains can be correctly identified with a false positive rate of about 13%.

There are however several limitations to our approach. First, while 61 total sessions were recorded and analyzed, they were contributed by only four different users. A much broader sampling of users would have been desirable. Also, because our algorithm hinges on the assumption that a human is ultimately typing into the terminal, none of our chain length predictions would remain valid if a computer or script were entering commands. Additionally, our approach assumes that the user is typing normally. It is quite possible that an intruder, being aware of the specifics of our detection algorithm, would be able to intentionally alter his typing speed or command delay in order to hinder accurate detection. Lastly, our approach flags all incoming connections from the LAN as legitimate which would present a weakness if an attacker was able to successfully compromise another computer on the LAN and then attack our machine.

Chapter 4

Detecting Intruders at the Targeted Host: An Anomaly Detection Approach

A common technique hackers use to break into a computer host is to route their traffic through a chain of stepping-stone hosts. There is no valid reason for a legitimate user to use a very long connection chain to remotely login to a computer host. Intruders always connect through stepping stones with a long chain. To prevent

The work described in this chapter has been previously published: Wei Ding and Stephen Huang. Detecting intruders using a long connection chain to connect to a host. In Proc. of the 25th IEEE Intl. Conference on Advanced Information Networking and Applications, pages 121-128, 2011.

stepping-stone attacks, it is critical to detect the connection chains. Most of the existing stepping-stone chain detection research has been concentrated on detecting intermediate stepping-stone hosts. However, these stepping-stone detection methods do not solve the problem completely. First, most of the benefits of the detection go to the host at the end of the chain (the victim host), an unknown third party to the monitoring site. Secondly, the stepping stone is only able to gauge the maliciousness of a connection by the number of downstream chain instead of the complete chain. If the stepping stone is very near the victim in the connection chain, one may not be unable to distinguish a malicious chain from a benign connection.

Detecting a malicious connection chain is much more challenging from a victim's perspective than at stepping stones [113]. This is due to the fact that a stepping stone can perform timing and correlation analysis with all of the information sent between the attacker and victim. The packets from the intermediate host to the victim and back form a closed loop. Previous stepping-stone detection methods relied upon the time difference between the attacker sending data downstream, and a response from the server passing back upstream (reply echo time).

Victim-based detection has many difficulties of its own. First, there is no straightforward method of estimating the full round-trip time (RTT) for the length of the connection chain. This is primarily due to the nature of tunneled SSH connections, and the fact that SSH is an interactive terminal session. This means that over the course of an SSH session, there is no point in time at which the server sends data to the client and the client's machine automatically sends a reply back to the server. In the following part of this chapter, we introduce our method of detecting long connection chains from short connection chains in a victim host based way.

4.1 Filtration of Upstream RTT

As mentioned earlier, on the victim end, for each connection, all incoming request packets will give both inter-command time gaps and intra-command time gaps. In order to analyze the difference between short connections and long connections, all time gaps of each connection are sorted in increasing order. This (increasing) sequence of uRTTs distribution represents a characteristic of a connection. If there is a significant difference between short and long time-gap distribution, we can use quantitative methods to formally quantify the difference between them. A comparison between short and long uRTT distribution is shown in Figure 4.1.

In Figure 4.1, there are about 700 time gaps for each of the two connections. These time gaps include both inter-command gaps and intra-command gaps introduced earlier. From Figure 4.1, we cannot see a significant difference between short connection and long connection. Because for an incoming connection with a lot of command request packets, there will be many more intra-command time gaps than inter-command time gaps. For intra-command time gaps, it mostly changes from 0 to actual round trip time. After hundreds of intra-command gaps been sorted in increasing order, it will be very hard to see any statistical difference between two difference curves. Thus, a better way is to analyze and utilize these two types of time gaps separately. Intra-command gaps and inter-command gaps of one short and one long connection are compared separately in Figure 4.2 and 4.3.

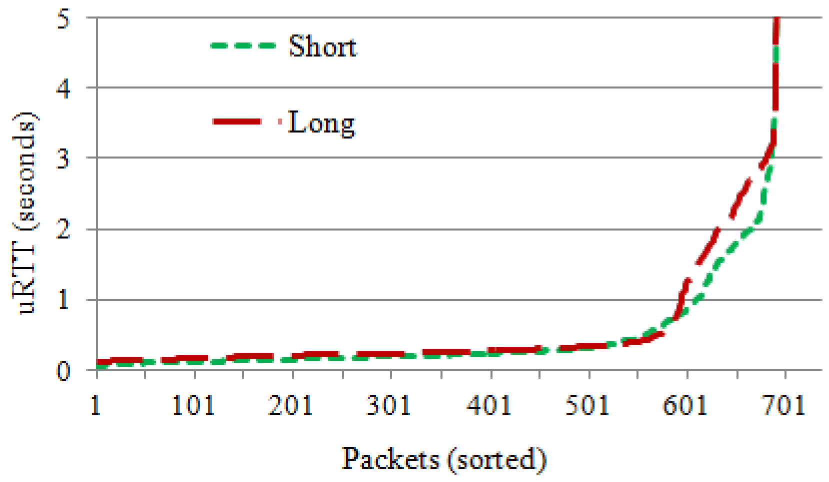


Figure 4.1: Comparison between Short and Long Connection Distribution

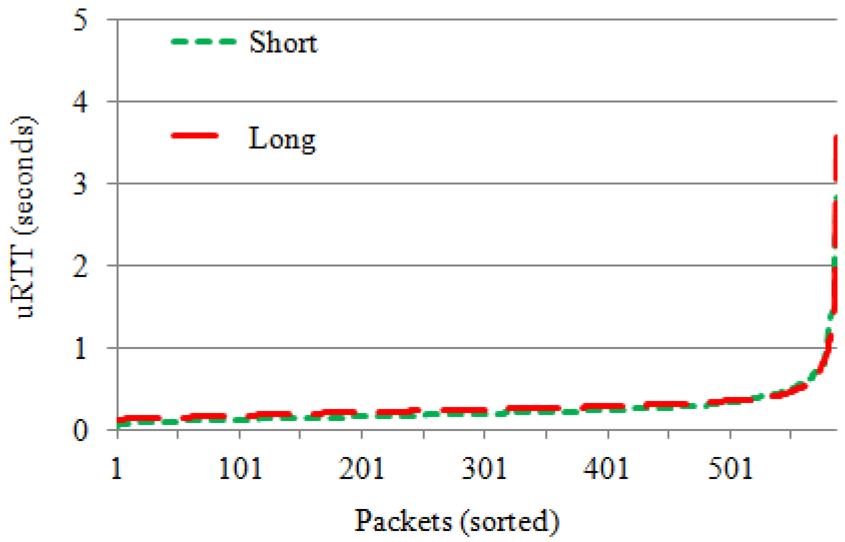


Figure 4.2: Distribution of Intra-command Gaps only

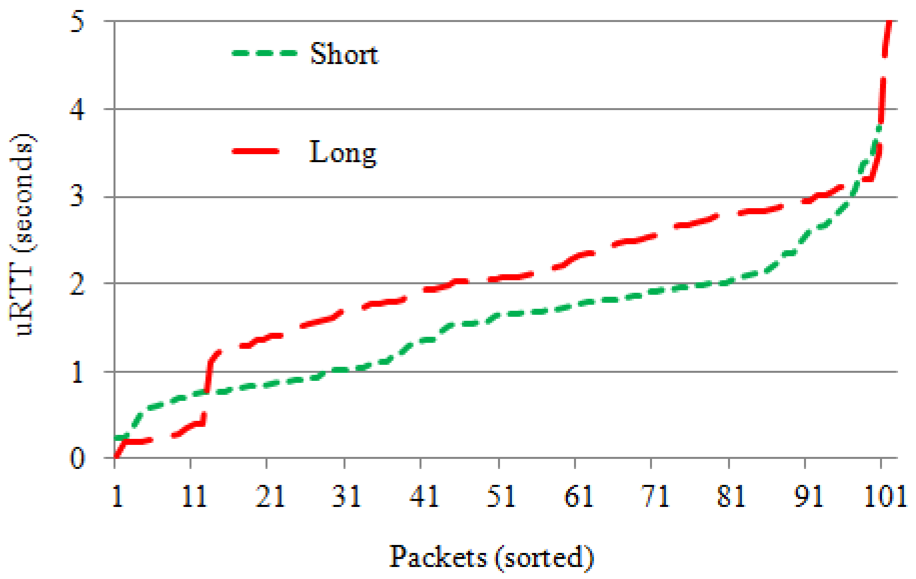


Figure 4.3: Distribution of Inter-command Gaps only

From Figure 4.2 and 4.3, one can see by using intra-command time gaps, the two curves of short and long connections are very close and it is hard to tell the difference. However, by using the inter-command time gaps, the difference between short and long curves is evident. Thus in the rest of this chapter, we will examine uRTT streams of inter-command gaps.

4.2 Algorithms of Measuring Distance between uRTT Streams

To detect long-connection chains from short-connection chains, the first step of our method is to use several collected one intermediate hop connections to build a profile of uRTT distribution with enough (say, 100) inter-command time gaps which is shown as a solid line in Figure 4.4. After that any extracted curves from a new collected connection packet stream will be compared with this profile distribution.

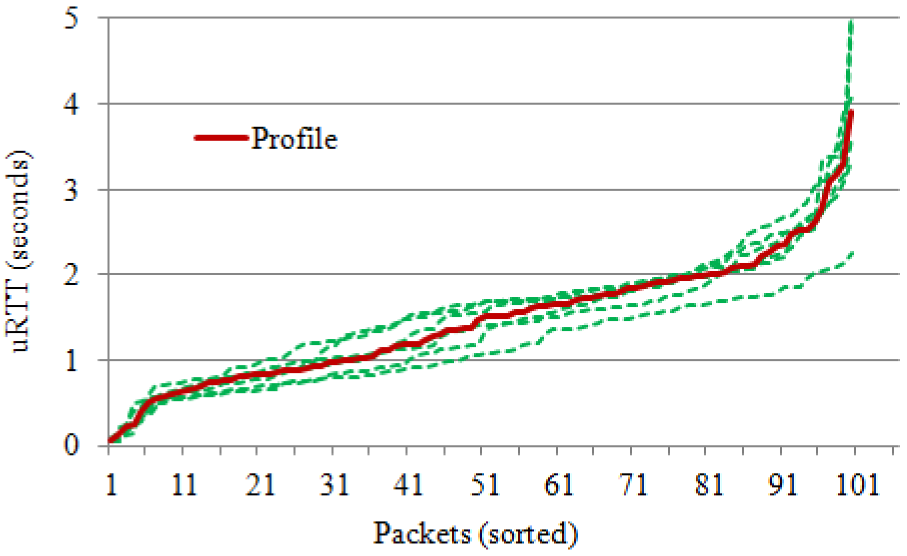


Figure 4.4: Using uRTT's of Short Chains to Build a Profile

4.2.1 Absolute Difference

Based on the profile curve built before, the next step is to quantitatively calculate the difference between the profile curve and the new connection distribution. The first distance measure is the absolute distance between the profile distribution and any test connection distribution based on inter-command time gaps. This is the most straight forward way to calculate a distance between two curves which is called absolute difference shown below. Gap g_p is the distribution of uRTT gaps of the profile chain and g is the test connection's distribution.

$$\begin{aligned} D\left(g, g_p\right) &= \sum_{i=1}^{N} \frac{\left|g\left[i\right] - g_p\left[i\right]\right|}{N} \\ g &= \left\{g\left[i\right] \middle| \, i : 1 \ to \ N\right\} \\ g_p &= \left\{g_p\left[i\right] \middle| \, i : 1 \ to \ N\right\} \end{aligned} \tag{4.1}$$

By using this straightforward distance measure, we can compute the difference of a distribution from the profile as illustrated in an example in Figure 4.5. However, there are some problems regarding this distance calculation method. Different users may have different typing speed resulting in a change of the uRTT distribution. Thus a short-connection chain may have a lot of small uRTT gaps, and it will get a much lower distribution under the profile curve. The calculated distance D will be very large, even larger than a distance between an actual long chain and the profile chain. In Figure 4.6, the calculated distance by formula one between the short chain and the profile chain will be larger than the distance between the long chain and the profile chain.

4.2.2 Distance with Median of the Ratio Adjustment

To deal with the problem illustrated in Figure 4.6, an adjustment method is used when users' typing speeds vary. This adjustment uses a ratio R to adjust and compensate for distribution with different average typing speeds. For some short connection curves, when they are below the profile curve, the calculated absolute difference will be large and may exceed the threshold. Hence, it will give a high false positive rate by the absolute difference method described above. For short connection curves under

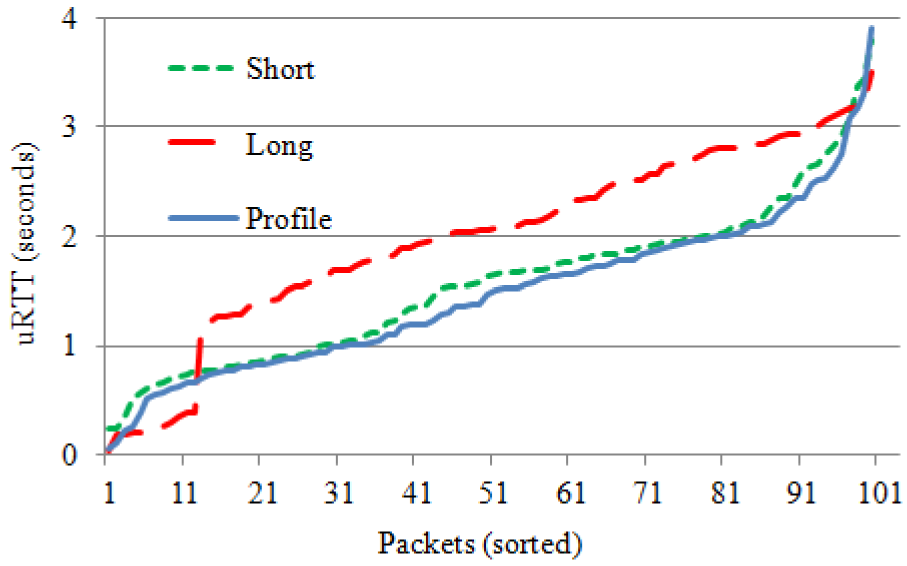


Figure 4.5: A Successful Detection by Absolute Difference

the profile curve, the ratio R will be greater than one which will shift the distribution up slightly. By doing this, the distance D_R will be decreased a little for these short connection chains. Besides, for long connection chains, there should be a significant difference between time gaps with packet crossover happening as shown in Figure 4.6 before and after packets number 25. Thus, the distance calculated by D_R may not decrease a lot, even though the ration R will make the connection distribution adjust up or down.

The distance D_R is calculated by Equation (4.2). In Equation (4.2), the gap g_p is the distribution of uRTT gaps of the profile chain and g is the test connection's distribution. R is the median of ratios between g_p and g by each gap. By using the ratio R multiplying the gaps of g, we can get the distance D_w to deal with the

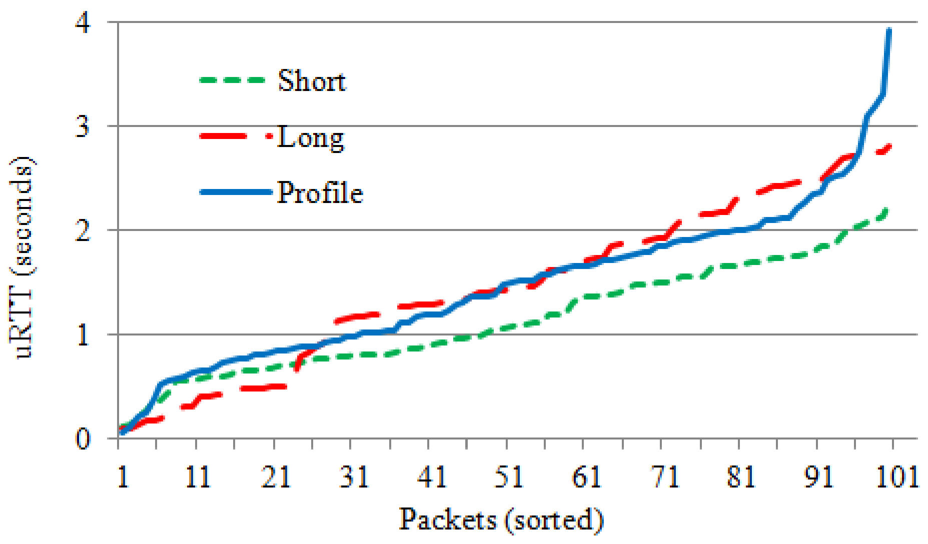


Figure 4.6: Short uRTT Curve with Large Distance by Absolute Difference

problem when user's typing speeds vary a lot.

$$\begin{gathered} D_R\left(g, g_p\right) = \sum_{i=1}^{N} \frac{\left|g\left[i\right] \cdot R - \ g_p\left[i\right]\right|}{N} \\ where\ R = Median\left\{\frac{g_p\left[i\right]}{g\left[i\right]} \middle| i = 1, 2, \ldots,\ N\right\} \end{gathered} \tag{4.2}$$

4.2.3 Distance with the Weighted Ratio Adjustment

The previous adjustment solves some problems when a short chain is lower than the profile chain. However, the long chain may be moved down, sometimes it will yield a small calculated distance. In some cases, this gives a new problem of missing the detection of some long chains. Hence, we try to use another characteristic of these uRTT distributions to deal with this problem. When long connection distributions have significant differences between time gaps with crossover, the linear regression slope of this uRTT distribution will be normally larger than the short connection distribution and the profile distribution. Hence, we implement the second adjustment method by giving a higher weight to the distribution which has a larger regression slope. Other distribution with slope less than the profile will have a weight of 0 as shown below.

$$\begin{aligned}
&W = \begin{cases} 1 - \frac{S_p}{S}, & if\ S_p < S \\ 0, & if\ S_p \geq S \end{cases} \\
&R_w = (1 + W) \cdot R \\
&D_w\left(g, g_p\right) = \sum_{i=1}^{N} \frac{\left|g\left[i\right] \cdot\ R_w -\ g_p\left[i\right]\right|}{N}
\end{aligned} \tag{4.3}$$

This adjustment improves the previous “median of ratio” adjustment method by keeping curves with larger slopes a larger distance D_w as shown in Equation (4.3). In Equation (4.3), we first calculate the slope of each uRTT distribution by their linear regressions ($y = S \cdot x + c$). Then for each connection having a slope S larger than the slope of the profile distribution S_p, we will give a weight W between 0 and 1. Otherwise, we give it a weight equal to 0. After we have weights of all distributions, we are going to calculate a new ratio R_w. For any distribution with a weight larger than 0, their ratio will be increased. Then by using the ratio R_w multiplying the gaps of g, we can get the new calculated distance D_w. Based on our experiment, most long connection chains will get a weight larger than 0, then they will have increased distance D_w by using this adjustment. By using this adjustment method, most long chains will have a bigger chance to be at an increased distance from the profile distribution which can increase the successful rate of classifying these long chains.

We summarize the two adjustment methods using one example in Figure 4.7. There are four chains in this figure including the original long chain and the profile chain. In this case, by using the ratio of the first adjustment method to adjust the curve, the curve is moved down and is very close to the profile curve. This makes the calculated distance between this long chain and the profile chain very small. To further increase the correct detection rate, the slope of the curve is utilized. By assigning a weight by the second adjustment method, the distribution curve is moved up which gives a large enough distance to detect this long connection chain. For most short chains with slope less than the profile chain, these short chains will not be adjusted by the weight which keeps their distance to the profile chain unchanged.

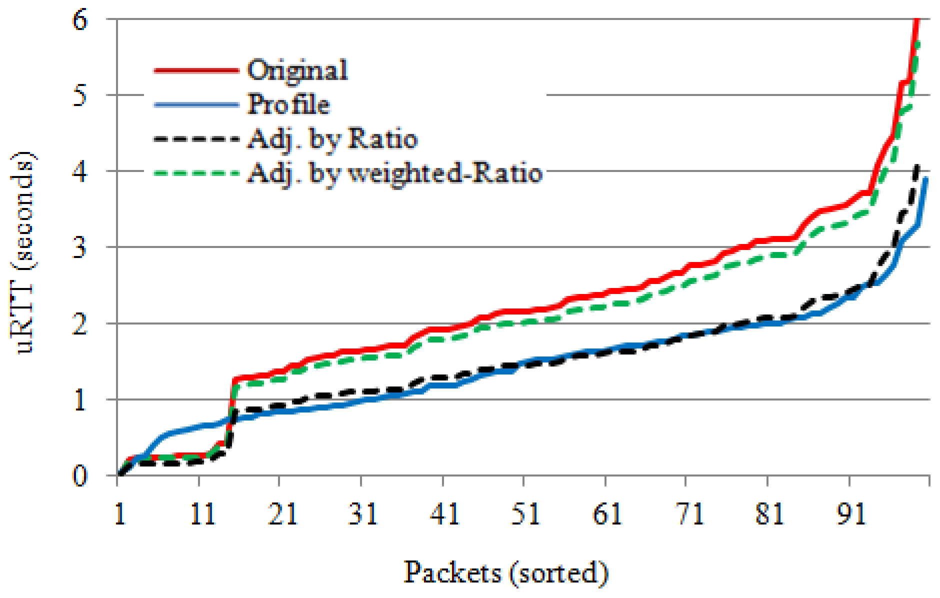


Figure 4.7: Comparison of Three Distance Measurements

4.3 Experiments and Validation

To demonstrate the validity of our claim and to see which one of the three distance calculation algorithms works better, we carried out an extensive study using data collected on the Internet. Several computers were used to build connection chains and data were collected. The computers ran various versions of Linux and all were connected to the Internet via high speed connections. A computer located at the University of Houston campus was used to monitor incoming connections and serve as a victim machine. A computer located off campus was used to initiate connection chains (attacker). The other computers were located in various regions of the US and primarily served as intermediate hosts in the building of connection chains. This enables us to collect packet data of connection chains of various lengths. We conducted twenty 1-hop sessions representing short-connection chains. To collect data for longer connection chains, remote computers may be used more than once in a chain because we have only a limited number of hosts off campus. These sessions consisted of users login into an SSH tunnel containing two, four, and six intermediate hops and executing a series of Unix commands. Twenty sessions of 2-hop, 4-hops, and 6-hops representing long connection chains were collected.

After filtering out the inter-command gaps, we derived 20 sets of 1-hop short connection chain distributions. We then built a profile using thirteen of these short chains. Then we cross validated the test data (4-hop and 6-hop chains) with the short chains. In each method, the three measures D, D_R, or D_W were used to see how the test chains differ from the profile. If the difference was greater than a threshold, then we classified the test chain as a long one. Otherwise, the chain is considered as a short one. In each case, we used the Leave-One-Out cross validation technique [15] to select a threshold for separating the short chains from the long chains. Several thresholds were used to with different level of false positive (i.e., short chains misidentified as long one) rate. As expected, by varying the threshold, there is a trade-off between the false-positive rate and the false-negative rate of the detection. The results from our experiment confirm this trade-off. We summarize the false rate in a receiver operating characteristic (ROC) curve shown in Figure 4.8.

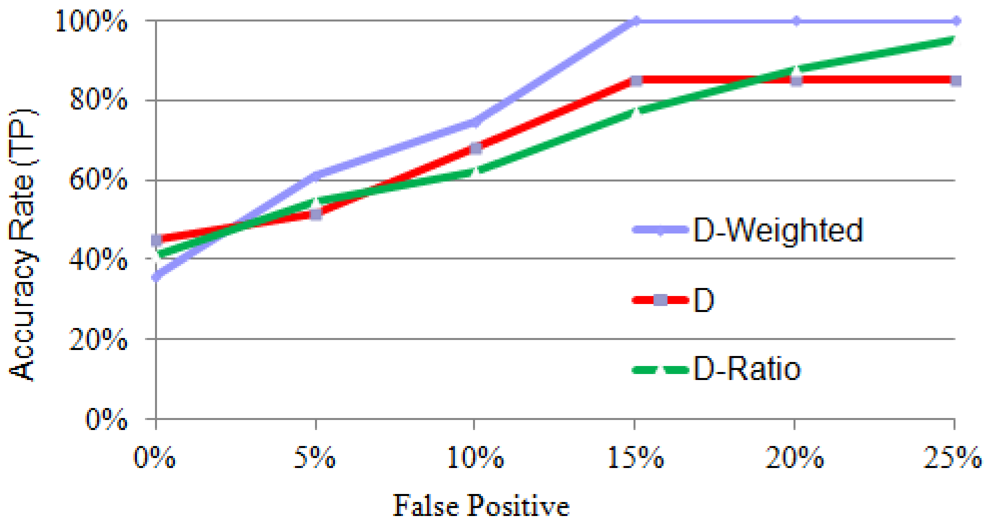


Figure 4.8: Successful Rate of Classifying 4-hop Chains by Three Distance Measurement with Different False-positive Rate

By using three distance measurements, we calculated the successful rate (true positive) of determining long chains by adjusting the false-positive rate. We can see the weighted ratio distance measurement is the best one among the three measures used. By using the weighted ratio adjustment method, we can successfully classify all 4-hops and 6-hops chains if we are willing to accept a FP rate of 15% as shown in Figure 4.9. Typically, there are many fewer intruders than normal users, so we may prefer to keep the FP rate relatively low. Figure 4.9 also confirms our belief that longer chains (6-hop) are easier to detect than shorter ones (4-hop in the figure). The result of comparing a 2-hop chain with the profile is not good as expected. Fortunately, most stepping-stone intruders use more than 2 hops to connect to the victim host, so we don't consider this a restriction on our result.

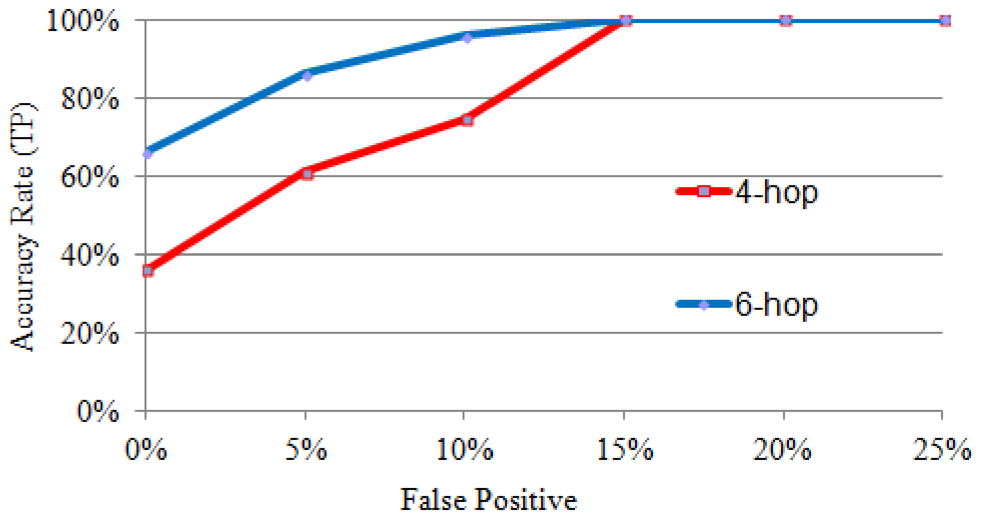


Figure 4.9: Successful Rate of Classifying 4-hop and 6-hop Chains by Weighted Ratio Distance with Different False-positive Rate

4.4 Conclusion

We have proposed a new approach to detect long stepping-stone connection chains from short connection chains at the victim host. Our method of detection centers on utilizing the packet stream of incoming connections to build an inter-command gaps curve. By using a new connection distribution compared with a profile of short connection chains, it is possible to detect long connection chains with a certain threshold. Our experiments show that by tolerating a false-positive rate of 15%, 100% of the test cases (4-hop and 6-hop) can be correctly detected with our weighted ratio distance measurement. This is somewhat surprising because we know clearly that the uRTTs are not a good estimate of the true RTTs. Our approach differs from previous work in that previous methods could only detect long downstream connections from the perspective of a stepping stone. All benefits of detecting such suspicious connections would go to the end hosts, who are likely unaffiliated with the implementer of the software. There are however several limitations to our approach. First, a limited number of sessions (60 in total) were analyzed which may not reflect all statistical characteristics correctly. A much broader sampling of users would have been desirable. Also, because our algorithm hinges on the assumption that a human is ultimately typing into the terminal, command streams intentionally conducted by programming scripts may not be detected correctly.

Chapter 5

Detecting Stepping Stones under the Influence of Packet Jittering

Cybercrime has become a serious problem all over the world. Based on the IC3 (Internet Crime Complaint Center) Internet crime report [114] for 2013, there were 48.8% more complaint submissions in just one year from 2012 to 2013. However, this is just the tip of the iceberg. Many more cases are undetected or unreported. Identity theft, data loss, and privacy leaking have been a great concern to all of us. Hackers normally use previously compromised computers as intermediate hosts to route the

The work described in this chapter has been previously published: Wei Ding, Khoa Le, and Stephen Huang. Detecting stepping-stones under the influence of packet iittering. In Proc. of the 9th Intl. Conference on Information Assurance and Security, pages 31-36, 2013.

traffic rather than directly connecting from their own host as shown in Figure 5.1. These intermediate hosts are called “stepping stones”. Connecting through stepping stones is a widely used technique by network attackers to attain anonymity and avoid being traced. At the target of the attack, the only visible machines are those directly connected to it, i.e., those exchanging packets with it.

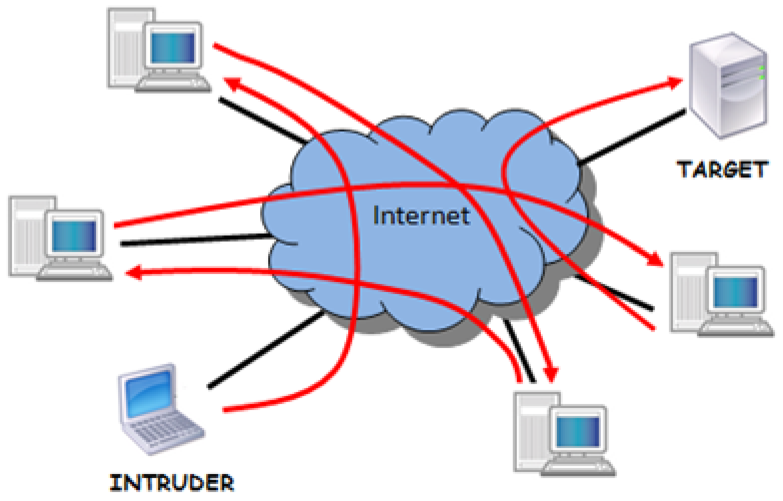


Figure 5.1: Attacking a Target through Compromised Stepping-stone Hosts

5.1 Evasion Techniques against Correlation-based Detection

Stepping-stone attacks can be detected by applying correlation-based algorithms on the connections in and out of an intermediate host. In other words, if one can identify whether a host is being used as a stepping stone, then it is possible to break the stepping-stone connection chain, in order to stop this suspicious intrusion.

By utilizing stepping-stone correlation algorithms, based on dynamic time warping algorithms [115][116][117][118][119][120][121][122], if an incoming stream and an outgoing stream of the chain can be matched upon similarity, we can successfully stop the stepping-stone intrusion. The correlation of incoming and outgoing streams is shown in Figure 5.2.

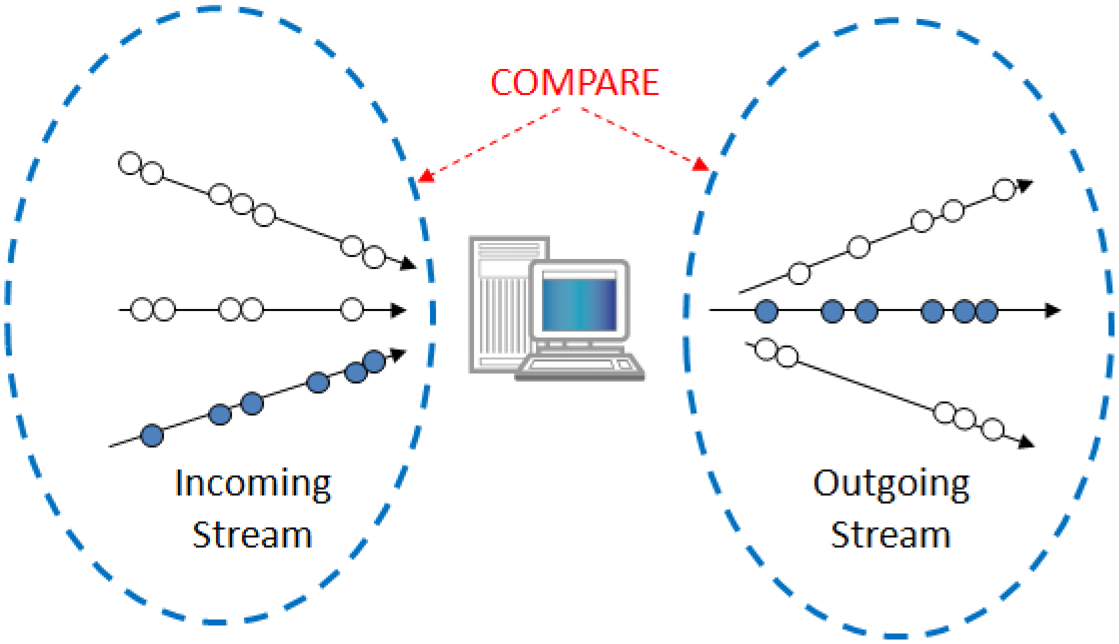


Figure 5.2: Match Incoming and Outgoing Streams

However, hackers have developed new evasion techniques by adding chaff packets into the traffic or jittering packets of the traffic. The chaffing methods intentionally injects bogus packets (so called chaff packets) at a stepping-stone host. These chaff packets can be removed later manually or automatically. If a significant amount of chaff packets are injected into a stream, the time-arrival gaps pattern between this stream and the stream before it will be noticeably different. The chaffing evasion technique can cause the stepping-stone detection algorithm to fail. In addition, chaff packets can be very hard to distinguish from normal packets, especially when traffic is encrypted [123]. Jittering is another evasion technique against stepping-stone correlation algorithms. By intentionally delaying some packets, the time-arrival gaps will change compared with the original stream. This pattern change can destroy the stepping-stone correlation between the jittered stream and the original stream before it [124]. Those two techniques are implemented on a stepping-stone host as illustrated in examples in Figure 5.3.

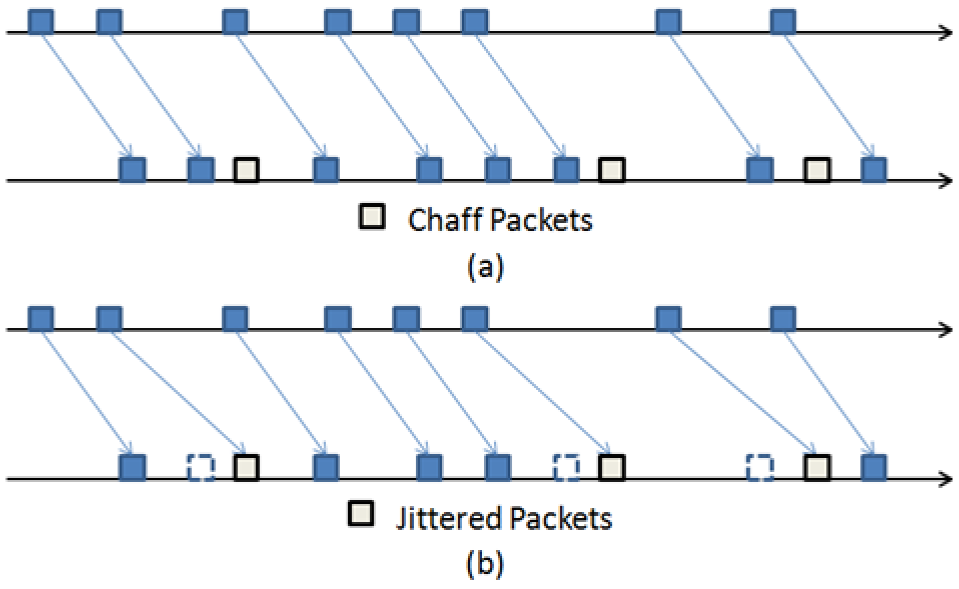


Figure 5.3: Chaffing and Jittering: (a) Green squares represent the original packets and black squares represent extra injected chaff packets; (b) Green squares represent the original packets, green dashed squares represent the original packets if not jittered and black squares represent packets been jittered

The purpose of adding chaff packets or jittering original packets is to change the pattern of an outgoing connection of a host and make it sufficiently different from the incoming connection. This difference may invalidate the correlation-based stepping-stone detection algorithms. Previous work has been limited to overcoming chaffing evasion [92][123]. It has not yet been known whether we can overcome the jittering evasion.

We are going to give a solution to counter packet jittering evasion technique of stepping-stone detection. Our approach is parallel to that of Huang and Kuo [123]. For chaffing, it is relatively easy to define a chaff rate which is the ratio of the number of added packets and that of the original packets. We also have to worry about the probability distribution of the inter-arrival time of the added packets. For jittering, there is a similar jittering rate (ratio of delayed packet vs. total packets) and a probability distribution of the amount of added delay. The net effect of chaffing is relatively easy to see. Many packet intervals are split due to chaffing. So there will be an increase of small interval gaps. For jittering, the effect is not that obvious. In fact if two consecutive packets were delayed by the same amount of time, the net effect on the gap is none.

This research focuses on finding a solution to counter packet jittering evasion technique. If an attacker jitters a significant percentage of packets at one of the stepping-stone hosts as shown in Figure 5.4, then the pattern of outgoing packet traffic can be different from that of the incoming traffic thus causing timing-based correlation detection algorithms to be no longer reliable. Based on our observations, if a connection is being jittered enough, the pattern of the jittered flow will be different from the un-jittered traffic flows. Thus we take a completely new approach based on the above hypothesis. If one can determine that a connection flow is being jittered, then one will be able to conclude that the host is being attacked (i. e., being part of the stepping-stone chain) [125].

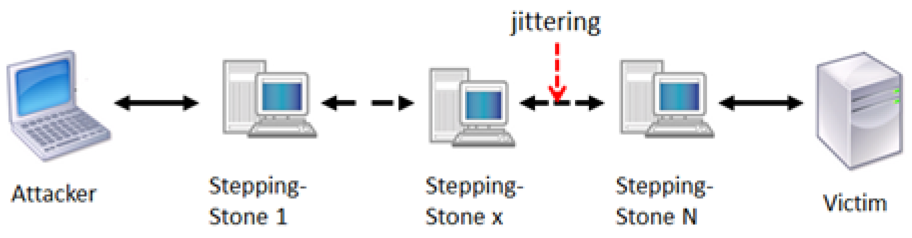


Figure 5.4: Jittering the Traffic at a Stepping-stone Host

We assume hackers may impose extra delay on selected packets on a connection so that the packet traffic patterns before and after the jittering may looked somewhat different. Since the pattern of packets inter-arrival time gaps is manipulated, the pattern correlation-based method would not be effective anymore. Thus we take a completely new approach based on the above observation. If we can detect that a connection's distribution is different from non-jittered ones, it is highly likely that this connection is being jittered. If one can determine which connection flow is being jittered, and then one will be able to conclude that the host is most likely being attacked (i. e., being part of the stepping-stone chain). This method can be combined with correlation-based detection algorithms which provides a detection method with or without jittering. This approach can complement correlation detection algorithms directly to yield a more robust method.

5.2 Modeling Traffic Streams

If a pair of incoming and outgoing connections of a host is strongly correlated by pattern matching algorithms, one can reasonably conclude that the host is being used as a stepping-stone. To counter those detection methods, intruders may try to change the pattern of one outgoing connection stream at a host which can cause the correlation between previous connection streams and the stream manipulated failed. To manipulate a connection stream, one can inject chaff packets or jitter certain amount of packets. By using those methods, the pattern correlation algorithm will fail in certain degree, and the stepping-stone detection rate will decrease.

Our approach is to extract certain features of the jittered and non-jittered packet traffic and look for the differences in these features. Once we have identified the features, we then extract the distinguishing features from many test cases and use the support vector machine (SVM) algorithm to separate the two clusters. After we find the separating hyper-plane, we can use that to determine whether a testing case is from the jittered or un-jittered stream. Our strategy utilizes statistical distributions to fit the normal traffic flows based on packet-arrival time gaps and by using the normal model, we can identify anomalous connections with intentional jittering. This "fitness" turns out to be a feature that was able to separate a jittered traffic from un-jittered one. The first step of the strategy is to decide the model of fitting normal traffic packets gap flow.

5.2.1 Modeling Normal Traffics

Network packet traffic flows have been widely studied for more than a decade. The oldest traffic model for analyzing traffic packets arrival time gaps is the Poisson distribution [126][127][128]. Further research on packets arrival time gaps suggests the aggregation of those measurements fit better with the General Pareto and Lognormal distributions [129][130][131][132].

Studies of Internet traffic have found that keystroke inter-arrivals can be very well described by those distribution models. Hence, we can use those distributions to fit normal traffic streams. The Probability Density Function (PDF) of the General Pareto distribution is shown in Equation (5.1), where k is the continuous shape parameter, σ is the continuous scale parameter ($\sigma > 0$), and μ is the continuous location parameter. The PDF of the Lognormal distributions is shown in Equation (5.2), where σ is a continuous parameter ($\sigma > 0$), μ is a continuous parameter, and γ is a continuous location parameter.

$$f\left(x\right) = \begin{cases} \frac{1}{\sigma}\left(1 + k\frac{(x-\mu)}{\sigma}\right)^{-1-1/k} & k \neq 0 \\ \frac{1}{\sigma}exp\left(-\frac{(x-\mu)}{\sigma}\right) & k = 0 \end{cases} \tag{5.1}$$

$$f\left(x\right) = exp\left(-\frac{1}{2}\left(\frac{\ln\left(x-\gamma\right)-\mu}{\sigma}\right)^2\right) / \left(\left(x-\gamma\right)\sigma\sqrt{2\pi}\right) \tag{5.2}$$

Given the traffic model and normal traffic data, the first step is to find distribution parameters by using Maximum Likelihood Estimation (MLE) [133]. MLE selects values of the model parameters that maximize the probability of getting the observed data (i.e., parameters that maximize the log-likelihood function lnL). Given the fixed observed inter-arrival time values $\{x_1, ..., x_n\}$ and a probability density function $f(x)$ of a testing model with k parameters, where θ is the function's unknown variable, and $\theta = \{\theta_1, \theta_2, \ldots, \theta_k\}$ is a vector defined on a k-dimensional parameter space. The log-likelihood function evaluates the fitness of the distribution to the data set. The goal was to find the maximum likelihood estimator $\hat{\theta}_{MLE}$ which is closest to the true value θ. These two are defined as follows in Equations (5.3) and (5.4).

$$\begin{aligned} lnL\left(\theta|x_1, x_2, ..., x_n\right) &= \ln f\left(x_1, x_2, ..., x_n|\theta\right) \\ &= ln\left(f\left(x_1|\theta\right) \times f\left(x_2|\theta\right) \times ... \times f\left(x_n|\theta\right)\right) \\ &= \sum\nolimits_{i=1}^{n} \ln f\left(x_i|\theta\right) \end{aligned} \tag{5.3}$$

$$\hat{\theta}_{MLE} = \arg\max_{\theta\in\Theta} \hat{\ell}\left(\theta|x_1, x_2, ..., x_n\right)\ \hat{\ell} = \frac{1}{n}\ln L \tag{5.4}$$

To estimate parameters of a certain distribution model when fitting data under that distribution, a lot of tools can give the estimation based on MLE. Matlab functions [134] or Python library Scipy [135] can fulfill the job very well.

The procedure of determining parameters under a certain distribution model from normal traffic data is the training process. Following that, for any incoming connection as testing data, we can apply the same test procedure and compare the test statistic with normal ones. The whole process should help us to detect jittered connections from normal un-jittered ones.

For a normal traffic packets inter-arrival time gaps flow, General Pareto and Lognormal distribution fittings are demonstrated in Figure 5.5. Both General Pareto

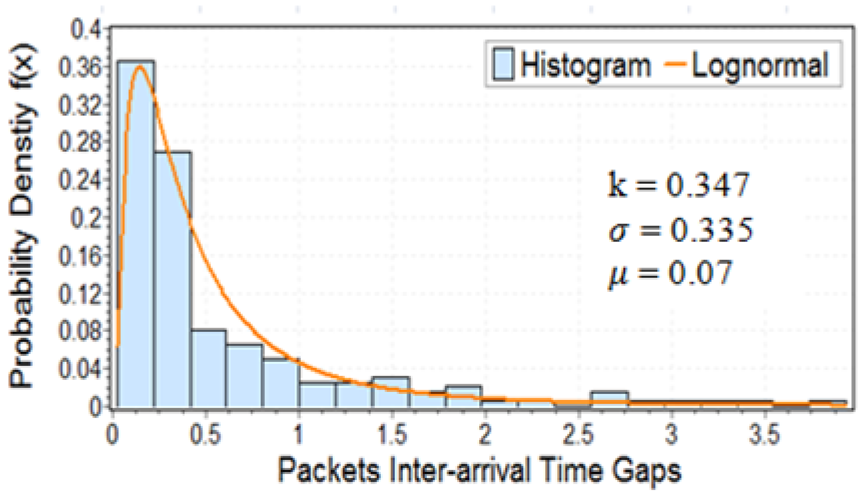
(GP) and Lognormal fit the normal traffic well. We do not need to use multiple fitting distributions to build the normal traffic model together. Based on our observation, one single distribution is enough to be used to establish the model to distinguish jittered flows from normal ones. For the rest of this chapter, we will use General Pareto Distributions to fit traffic flows.

5.2.2 Modeling Jittered Traffic

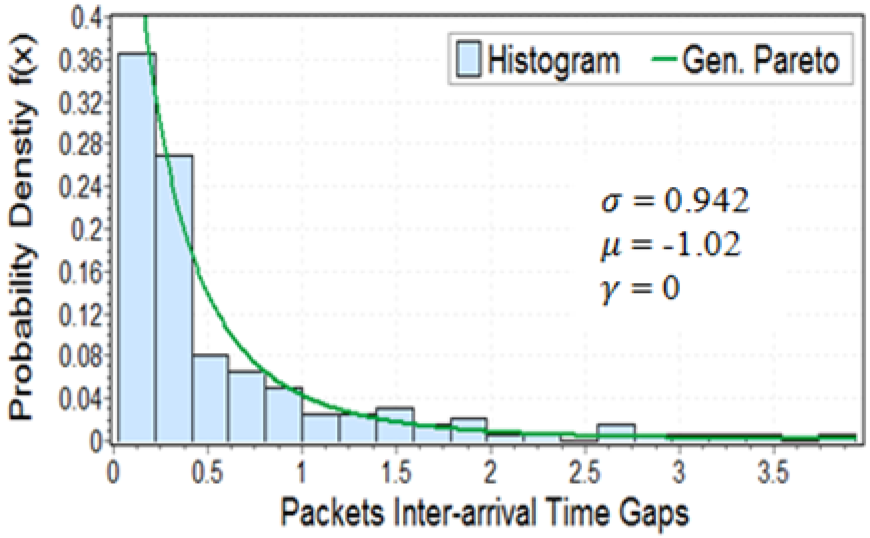
Since Pareto and Lognormal distributions model Internet traffic well, researchers also use those distribution models in their study to generate chaff packets or to jitter the original packets. The intuition behind this is to make tampered traffic close to the original one.

Figure 5.6 shows the histogram of jittered traffic flow together with the fitting. Here we randomly generate jitters based on GP distribution with parameters k =0.2 and σ =0.4 when $\mu = 0$. The histogram of jittered flow is noticeably different from the original un-jittered one shown in Figure 5.5. A comparison between the original normal traffic flow and a jittered traffic flow can be demonstrated by the histograms in the two figures.

After fitting the GP distribution to the jittered traffic flow, based on Maximum Likelihood Estimation, estimated parameters of GP distribution are calculated. We get k = -0.028 and σ = 0.87 for the jittered flow. These estimated parameters can be used to measure the difference between un-jittered and jittered flows in a numerical way. By using the General Pareto Distribution, based on our observation



(a)



(b)

Figure 5.5: Fitting Packet Gap Data with (a) General Pareto and (b) Lognormal Distributions

and the MLE test, it is clear that a traffic flow after being jittered by hackers can be measurably different from normal traffic flows. Further study suggested us that the parameters k and σ are enough to be used as features to separate un-jittered and jittered traffics. Since μ is a location parameter, for the sequence of packets inter-arrival time gaps, we can always set μ to 0 during our distribution fitting process.

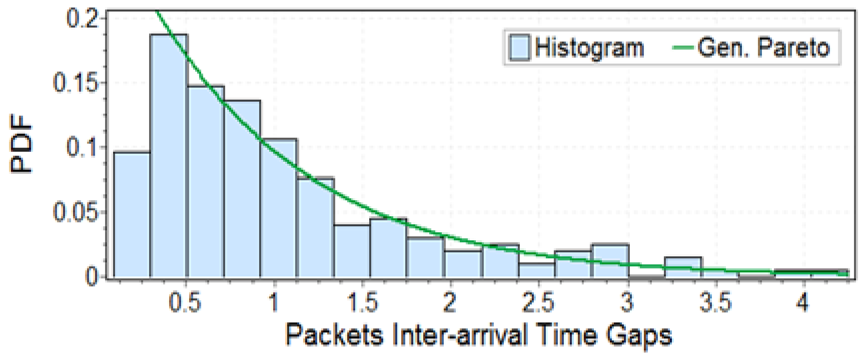


Figure 5.6: Fitting Jittered Data with General Pareto Distribution

In order to measure how well un-jittered and jittered traffics can be separated, as well as the successful detection rate of our methods, we can utilize Support Vector Machine (SVM) [136] to do the separation based on estimated parameters of General Pareto distribution which are k and σ. Our detection algorithm is about building the SVM based on un-jittered normal traffics and jittered training cases. Then for the new incoming traffic we will utilize the SVM to classify its category. An example of separation with SVM, based on General Pareto's parameter K and σ, is shown in Figure 5.7. In the figure, we can see jittered ones are perfectly separated from original ones using SVM. In this training case, we can separate the two clusters 100%. We will show that this perfect detection rate may not always be possible. If there are any misclassified ones from jittered parts, we will count those as false negative which will decrease the detection rate.

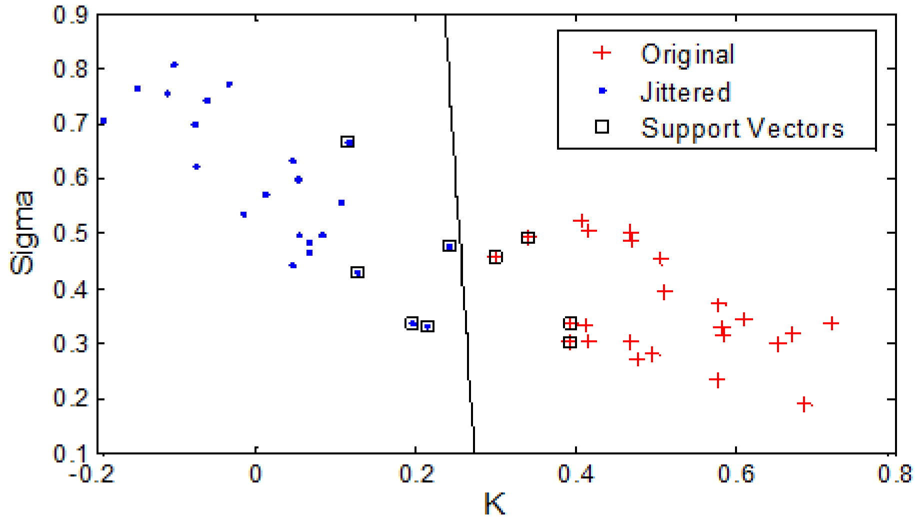


Figure 5.7: Separating Jittered Traffic Streams from Original Non-jittered Streams using Support Vector Machine

However, the problem is we cannot control how hackers would apply jitter to the traffic flow. This uncertainty issue can be divided into two unknown parts: jittering rate, distribution and parameters used to generate jitters. For a certain jittering rate, whether a packet will be jittered or not is under Bernoulli distribution [137]. There are too many ways, in other words many possible distributions, to generate jitters. Based on previous research work and our study, we choose the General Pareto distribution to demonstrate our methodology of detecting jittered abnormal traffic flows from normal traffic flows. It turns out that when other distributions were used, the result stated in this chapter did not change significantly.

5.3 Jittered Traffic Flows Detection Algorithm

Based on the methodology discussed in the previous section, we propose our jittered traffic flows detection algorithm as below:

Algorithm: Detecting Jittered Traffic Flows

Given N un-jittered traffic flows:

Step 1: *For each traffic flow, based on GP distribution, use MLE to estimate k and σ, which are two attributes to be used in the classifier.*

Step 2: *Based on un-jittered flows, under a certain distribution, generate jittered flows, which are based on a combination of various jittering rate, mean, and std.*

Step 3: *For jittered flows, based on the GP distribution, use MLE to estimate k and σ.*

Step 4: *Train the SVM by using jittered and un-jittered traffic flows with estimated k and σ.*

Step 5: *For each new incoming traffic, based on the GP distribution, use MLE to estimated k and σ, and use the trained SVM to classify the new incoming one into jittered or un-jittered.*

Step 6: *If the incoming traffic flow is classified as a jittered connection, flag it as an attack.*

To further demonstrate the effectiveness of the proposed jittered traffic detection algorithm, in the following sections, we will introduce how we set up experiments and discuss the impact among distributions, jittering rate, mean of jittering, and standard deviation of jittering.

5.3.1 Testing Jittered Traffic Flows

We set up a stepping-stone connection chain with seven intermediate hops. Every host on the chain was installed with a SSH server under the Linux system. In order to simulate long distance stepping-stone connection chain on the Internet, we picked three distant hosts located in Wisconsin, Chicago and Shanghai. All other hosts are located in our campus network, in Houston, Texas. This chain was designed to route through distant host and local host alternatively in order to collect packets locally, since most network packets collection tools, such as Wireshark, require root privilege. All experimental network traffic was captured in real time at our local hosts. We had three users, acting as three intruders, from three different local hosts, routed through the same stepping-stone chain, and finally arrived at another three different target hosts. This generated traffic was all captured at the local hosts. The experiment was repeated eight times on different days, which gives us totally twenty-four successfully captured traffic flows.

The basic idea of jittering network traffic is to store and forward the traffic through a certain host at a given port number. A lot of Man-in-the-Middle attack tools or TCP and UDP proxy software can be used to implement this function, such as Mallory [96]. Stored packets will be intentionally jittered, as well as randomly delayed, for a short amount of time. This jittering will actually change the inter-arrival time gaps between packets after those packets were being forwarded to the next host.

In our experiment, whether a packet was jittered or not was based on a Bernoulli distribution. By applying random delays generated by some probability distribution to the original network traffic, we can jitter the traffic under that distribution model.

One of the objectives of this study is to determine the effects on the detection by various parameters. Thus, we tried different combinations of mean, standard deviation, and jitter rate among different distribution models. Our experiment results indicate that the distribution model used to generate jittering does not impact our detection method.

Since an intruder can use any distribution models to jitter the original connection stream, one cannot pre-assume or control the way of how hackers manipulate the traffic. We tried several different distribution models, and based on our study, the distribution model used to jitter the traffic does not matter much. We will demonstrate our research results by using General Pareto, Lognormal, and even the simplest Uniform distributions to implement jittering of the original traffic in our experiment.

Under certain distribution models, there are different parameters used in probability density function (PDF) of a distribution. Common parameters of different distributed data are mean and standard deviation. Hence, we can transfer those parameters of a distribution used in PDF into mean and standard deviation.

When we have 100% jittering rate, with a mean set at 0.4 (seconds), we tested with a standard deviation from 0.2 to 1.4 to implement jittering based on General Pareto, Lognormal, and Uniform distributions. The detection rate of our method based on different standard deviations is shown in Figure 5.8. In the figure, the detection rates among different distribution jittered traffics are very close to each other based on the change of standard deviation. The detection rate increases as long as the standard deviation increases. We can have almost 100% detection rate when standard deviation approaches 1 second.

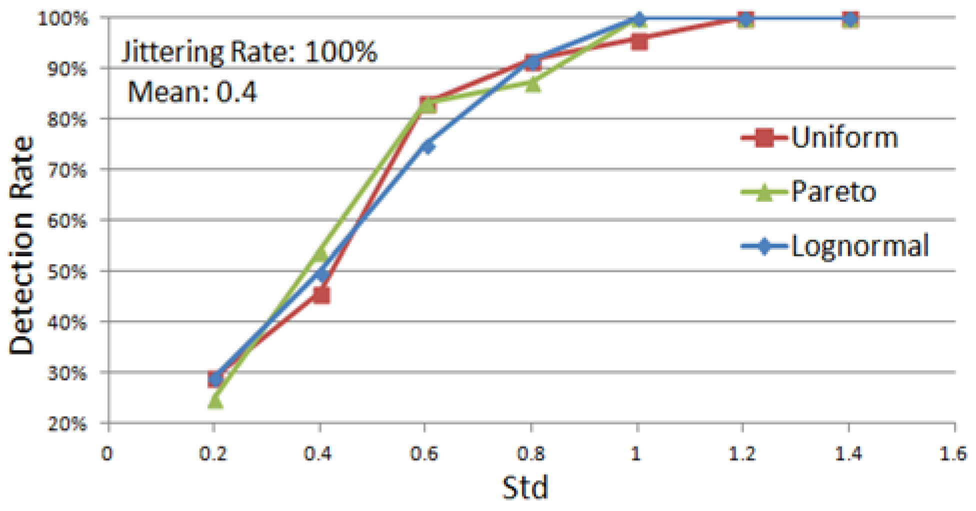


Figure 5.8: Detecting 100% Jittered Traffics with Different Standard Deviation by the Three Jittering Distributions

There are many combinations of different jittering rate, different mean, and different standard deviation. We tested our detection algorithm with different combinations among these three parameters. The results shown in Figure 5.8 below are based on a fixed mean of 0.4 and a fixed jittering rate of 100%. All results show that the detection rates based on jittering rate, mean and standard deviation do not differ much among different distribution used to generate jittering. In the following part of this chapter, we will show our results with the Uniform distribution model.

5.3.2 Impacts of Jittering Rate, Standard Deviation, and Mean on the Detection Rate

For a given jitter distribution, there are three parameters (jitter rate, mean and standard deviation of the jitter distribution) used to generate jitters. We conducted experiments to study their relationships. Based on our experimental results, we conclude that the standard deviation of the jittering distribution impacts the detection rate the most. On the other hand, the mean of the distribution doesn't impact the detection rate significantly. Different jittering rates do impact the detection rate but not as much as standard deviation does.

As shown in Figure 5.9, for each standard deviation, we compute the detection rate with mean = 0.2, 0.4, and 0.6. The detection rate for a given standard deviation doesn't change much for different mean values. All three curves shown in Figure 5.9 exhibit similar trends and very close detection rates for a fixed standard deviation value. This is also true when we vary the standard deviation from 0.2 to 1.4. The conclusion from this experiment is that the detection rate is very sensitive to the standard deviation (changes from 30% to 100%) but not as sensitive to the mean (0.2, 0.4 and 0.6) of the jittering distribution.

Our method can detect most jittered traffic streams under different jittering rates. As shown in Figure 5.10, the detection rate increases as the standard deviation increases for jittering rates of 25%, 50%, and 75%. In the figure, the standard deviation increases from 0.2 to 1.6 with a fixed mean of 0.4. Our method can detect most jittered cases with a detection rate mostly at 80% or above. The second conclusion

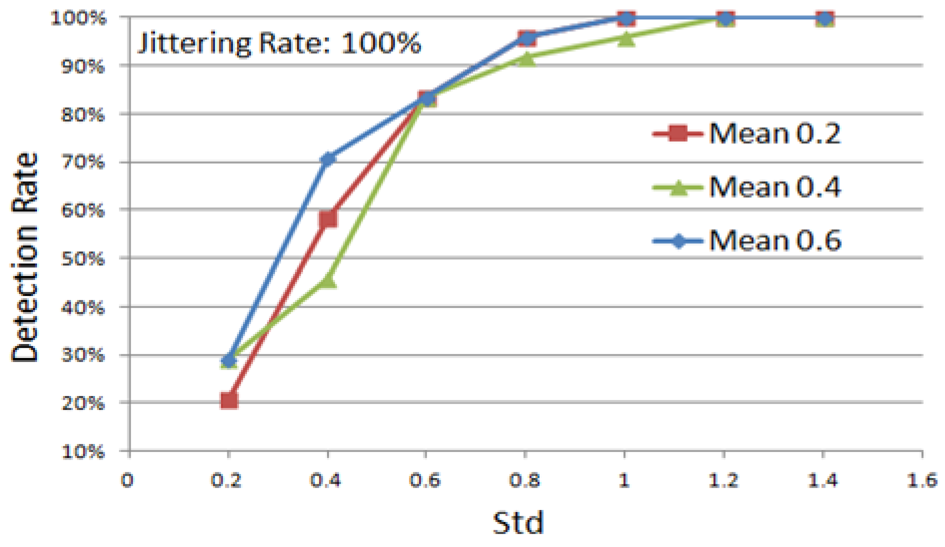


Figure 5.9: The Impact of Mean of the Jittering Distribution on the Detection Rate

from this experiment is that the detection rate is generally higher for a higher jitter rate even though the detection rates don't vary too much among the three jitter rates.

Furthermore, if we can tolerate some false positive cases, we can increase the detection rate further. The FP rate starts at 0% and goes up. Based on the trade-off between the true positive (TP) rate and the false positive rate, we can get the ROC curves shown in Figure 5.11. With a 100% jittering rate and a mean of 0.4, we can achieve a 100% detection rate with a standard deviation of 0.6, 0.8 and 1.0 if we tolerate a FP of 30%. However, as shown in the figure, when the standard deviation is 0.2, the detection rate is really low even after increasing the FP to 50%. Clearly, the detection algorithm does not work well when we have a high jittering rate, and low standard deviation. This issue will be discussed in the next section.

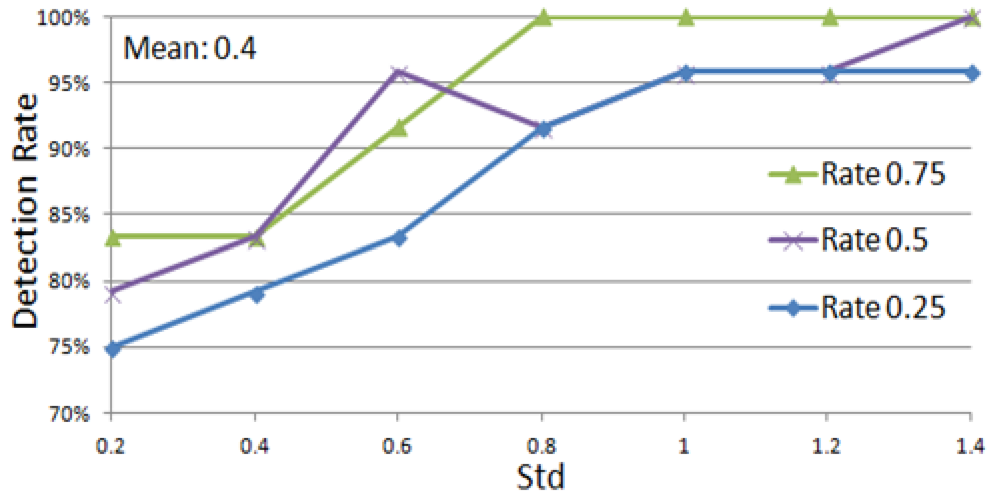


Figure 5.10: The Impact of Jittering Rate on the Jittering Detection Rate

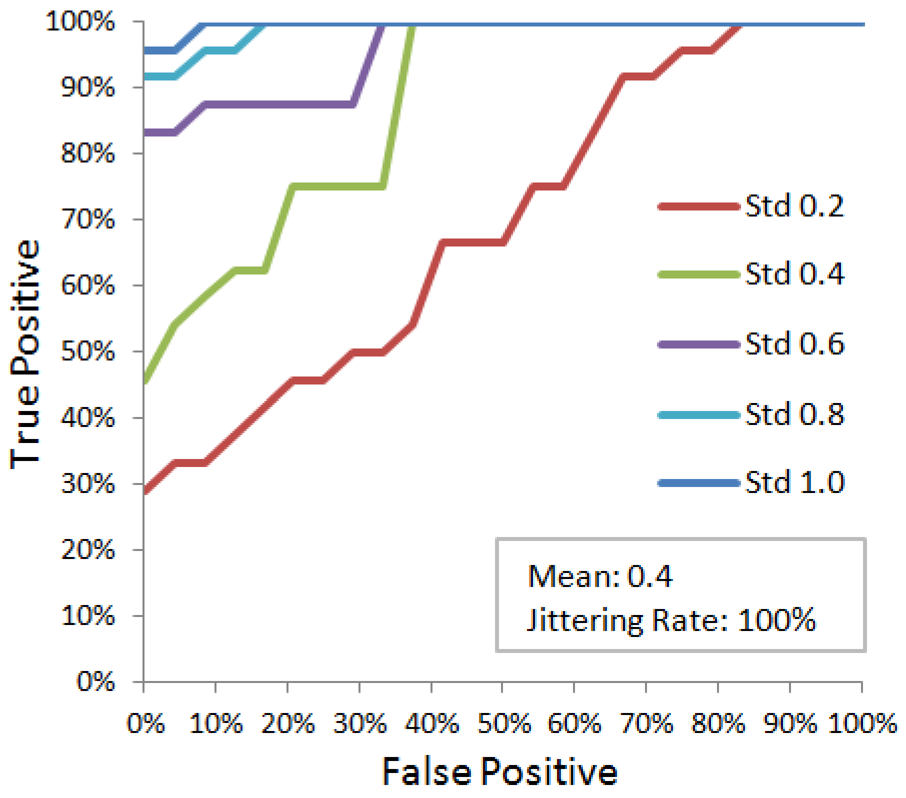


Figure 5.11: ROC Showing the Detecting Jittered Streams with Different Jittering Standard Deviation (Mean = 0.4 and Jitter Rate = 100%)

5.3.3 Deterioration of Jittering Detection Algorithm

When packets are closed to 100% being jittered, based on our observation, it is really difficult to detect the jittered traffic when the standard deviation of jitters is small. This is easy to understand. When all packets are delayed by almost the same amount (low standard deviation), the time gaps between the packets are changed only minimally. In Figure 5.12, we demonstrate the result on different means, 0.2, 0.4 and 0.6, over detection rate. The jittering rate is 100% which means all packets from the original connection are jittered.

In Figure 5.12, we can see no matter what distribution model is used to generate jitter, when we have a 100% jittering rate and a small value of standard deviation of our randomly generated sequence of jitter, our method has a low detection rate of below 30% when FP is 0%. This is because under the combination of 100% jittering and very small standard deviation, the generated random jitters vary very little among each other. When one applies almost the same amount of jitter to every packet, the inter-arrival time gaps between the two consecutive packets are almost not changed. Under this circumstance, the inter-arrival time gaps sequence of jittered one will almost remain the same with the original one.

However, this difficult case will be easily detected by most of the correlation-based detection algorithms, just because the jittered one remains almost the same with the original one. For the case of very high jittering rate, as long as the standard deviation is high, the detection rate will be close to 100%, which was shown in Figures 5.9 and 5.10.

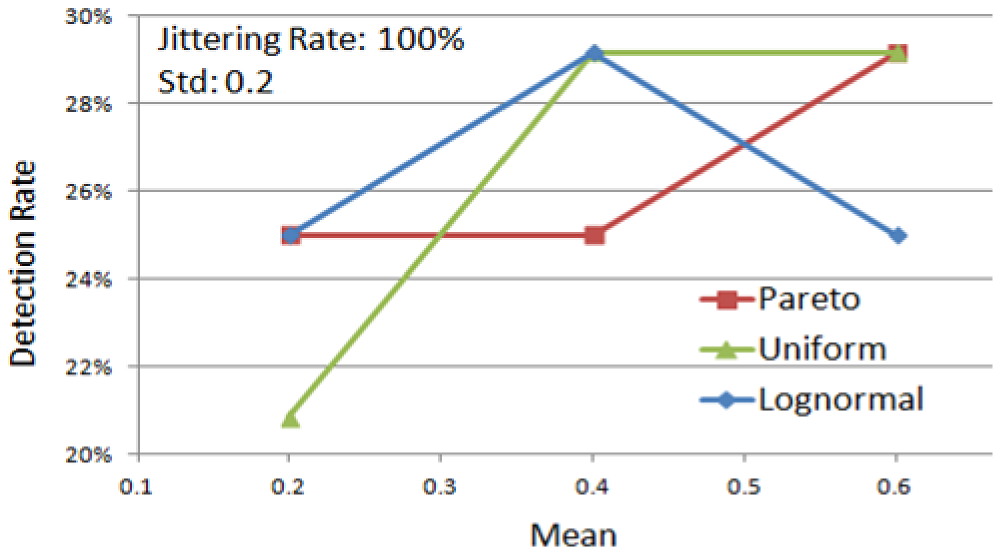


Figure 5.12: Testing Cases with Low Detection Rate (with 100% Jittering Rate and a Small Standard Deviation)

5.4 Hybrid Stepping-stone Detection Algorithm and its Validation

5.4.1 Detect Stepping Stones with or without Jittering

Our proposed approach can detect stepping-stone intrusions with jittering. If we combine this approach with correlation-based stepping-stone detection algorithms, we can come up with a much more robust detection method which can detect stepping-stone attacks either with jittering or without jittering.

As shown in Figure 5.13, we propose a hybrid stepping-stone hosts detection algorithm which combines our jittering detection module described above and a stepping-stone correlation algorithm. Figure 5.13(a) shows a typical correlation algorithm of

Table 5.1: The Optimal Slope Alignment (OSA) Algorithm

Algorithm OSA(R,T,n,m):

```
/* Given two sequences R and T with size n and m */
for (q=0; q<m; q++) cost[0][q] = 0; //initialization
for (p=1; p<n; p++)
    for (q=0; q<m; q++)
        for (k=0; k  q; k++)
            currSlope = T.get(q)-R.get(p);
            prevSlope = T.get(k)-R.get(p-1);
            slopeCorr = Math.abs(currSlope-prevSlope);
            if(cost[p][q]>cost[p-1][k]+slopeCorr)
               cost[p][q] = cost[p-1][k]+slopeCorr;
minCost= INT_MAX;
for (q=n-1;q<m;q++) //find optimal
    if(cost[n-1][q]<minCost)
    minCost= cost[n-1][q];
DisScore=minCost;
```

two streams and Figure 5.13(b) shows the hybrid algorithm. We first feed a test stream (typically an outgoing stream from a host) into our jittering detection algorithm to see if it has been jittered. If it's classified as jittered, we trigger the attack alarm. Otherwise, we will feed it into the stepping-stone correlation algorithm. By comparing with the reference streams (typically an incoming stream), if the testing connection is correlated with another connection, it will be classified as an attack. This proposed method utilizes our jittering detection approach to complement original correlation based stepping-stone detection methods to fortify its robustness.

We implement the optimal slope alignment (OSA) correlation detection algorithm [138] in our effort to validate the hybrid algorithm. The OSA detection algorithm, shown in Table 5.1, is an improvement of dynamic time wrapping-based correlation

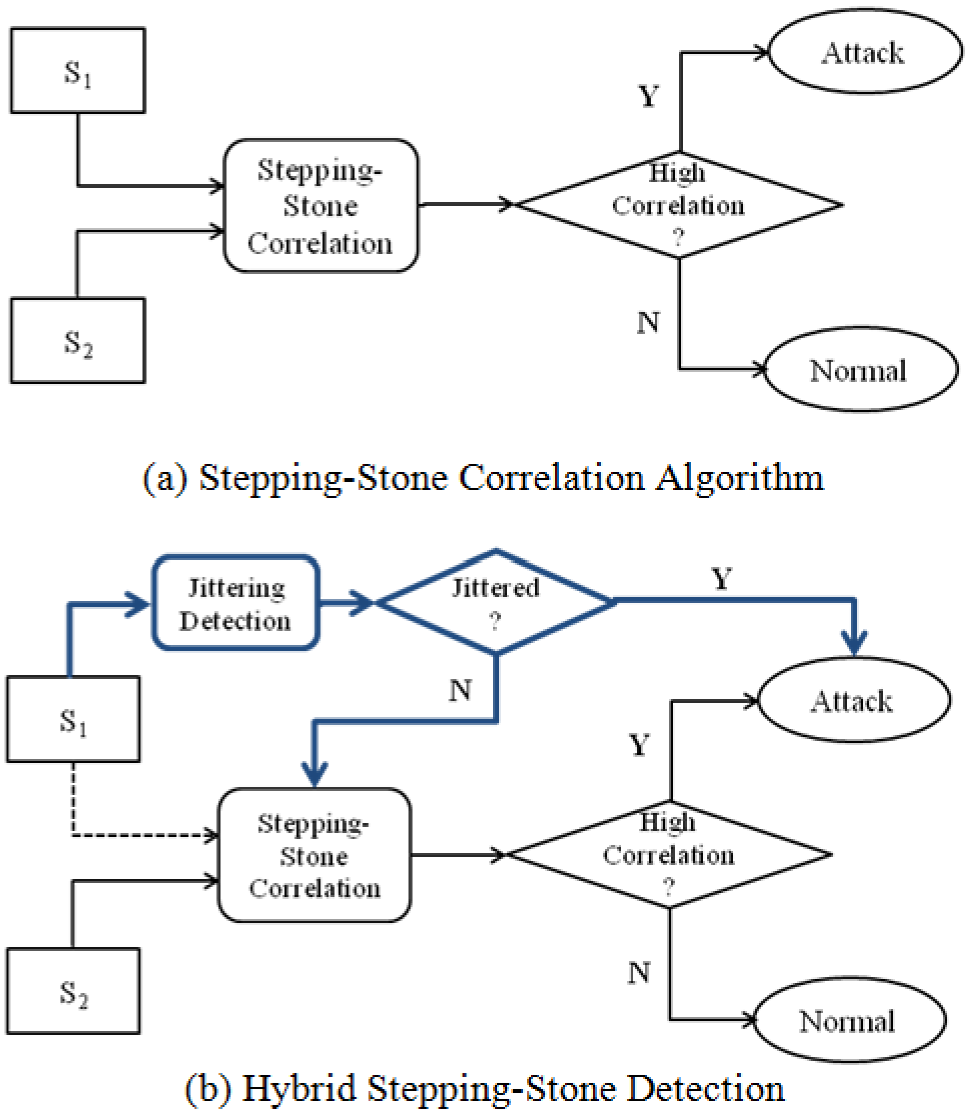

S1
S2
Stepping-Stone Correlation
High Correlation ?
Y
N
Attack
Normal
(a) Stepping-Stone Correlation Algorithm
Jittering Detection
Jittered ?
Y
N
S1
S2
Stepping-Stone Correlation
High Correlation ?
Y
N
Attack
Normal
(b) Hybrid Stepping-Stone Detection

Figure 5.13: Flow Charts of the Algorithms: (a) Stepping-stone Correlation Algorithm (b) Hybrid Stepping-stone Detection with an addition path in blue

algorithms. By given two sequences R and T, OSA tries to find the subsequence R_s of R and T_s of T such that R_s best matches T_s. The OSA algorithm compares the slope correlation to find the best match. Imagine a correct matching in Figure 5.13(a), the slopes of the line segments connecting the matching packets should be very close. OSA actually finds the best such match. If we use the OSA correlation algorithm to compare two streams of the same chain in and out of a host, we should get a low dissimilarity score (i.e., a very good match). If we correlate one stream from a connection with another stream from a different connection chain, we should get a high dissimilarity score. Calculated dissimilarity scores using OSA upon our experiment data are shown in Figure 5.14. To further illustrate the dissimilarity among different correlations by OSA, as shown in Figure 5.15, we utilize box-plot to demonstrate. One can clearly see the dissimilarity scores of unrelated chains are higher than the jittered streams with standard deviation = 0.2, 0.4, 0.6 and 0.8 correlated with their original streams. For jittered streams with standard deviation = 1.0, correlated with their original streams, the dissimilarity scores are as high as scores from the correlation between unrelated chains.

By using the minimum dissimilarity score of correlation among unrelated streams, we can get a threshold $\tau = 15.2$ to detect if two streams are from the same chain or different chains. Correlated with OSA, if two correlated streams have a dissimilarity score less than τ, they are classified as from the same stepping-stone chain. Otherwise, these two streams are classified as from the different chains. If two correlated streams are from the same stepping-stone chain, we can further label this chain as being used as a stepping-stone attacking chain.

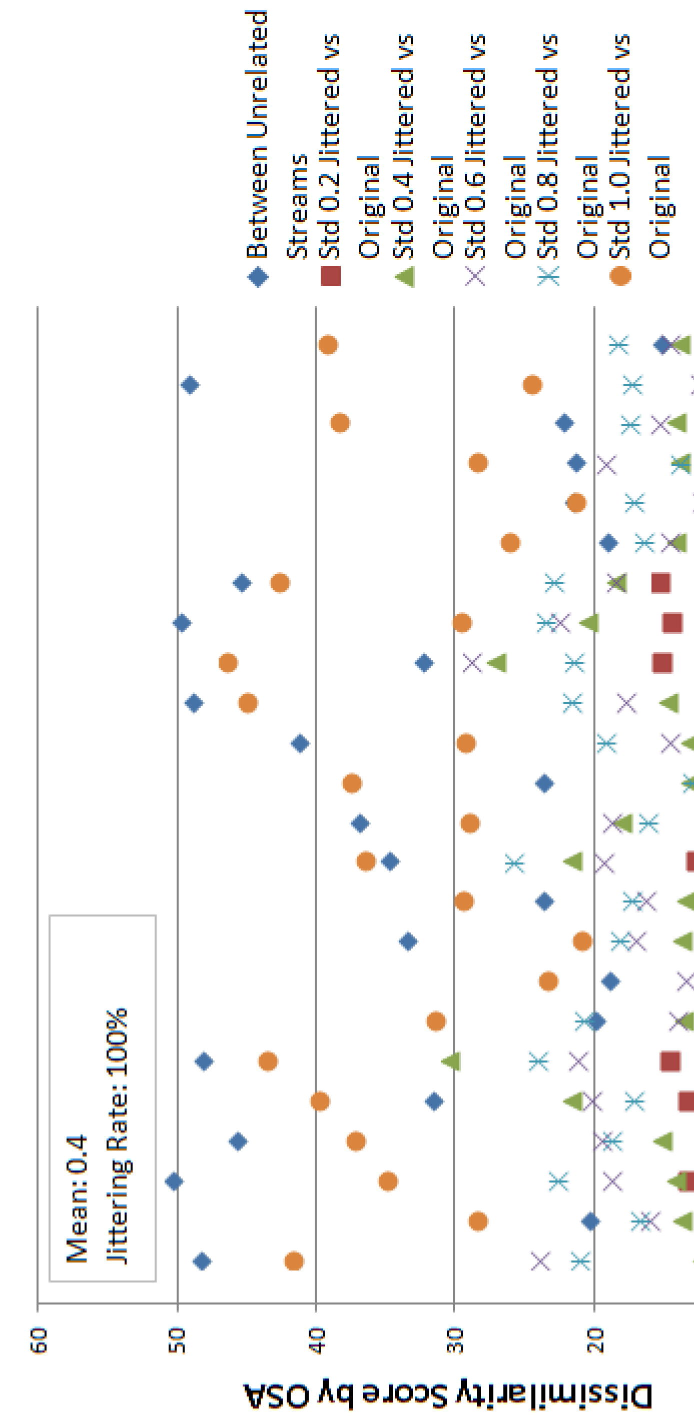


Figure 5.14: Correlated Dissimilarity Scores using OSA

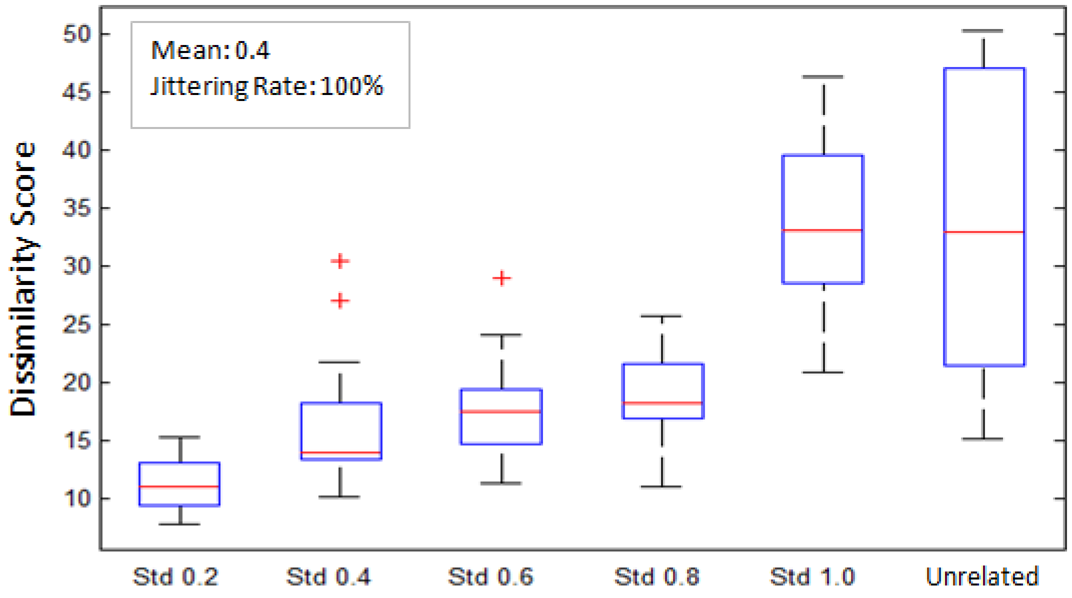


Figure 5.15: Dissimilarity Scores of the OSA Correlation Algorithms Increases as the Standard Deviation Increases

By tolerating certain FP values when producing the threshold τ, we can get the ROC curve for the OSA correlation-detection algorithm as shown in Figure 5.16. From the figure, one can see that the detection rate of the OSA algorithm is decreasing as long as the correlated stream is being jittered with higher standard deviation. The OSA stepping-stone-detection module can detect more than 90% jittered streams with std = 0.2 even with 0% FP. However, it cannot detect well jittered streams with standard deviation higher than 0.2.

The jittering detection algorithm works well when there is large jittering (higher standard deviation) but does not work well with lower jittering. When we combined the jittering detection with the correlation-based detection of stepping stones (see Figure 5.13), the result is much better. As shown in Table 5.2, for the same 24 traffic streams used in Figure 5.11 and Figure 5.17, with no false positive, the number of

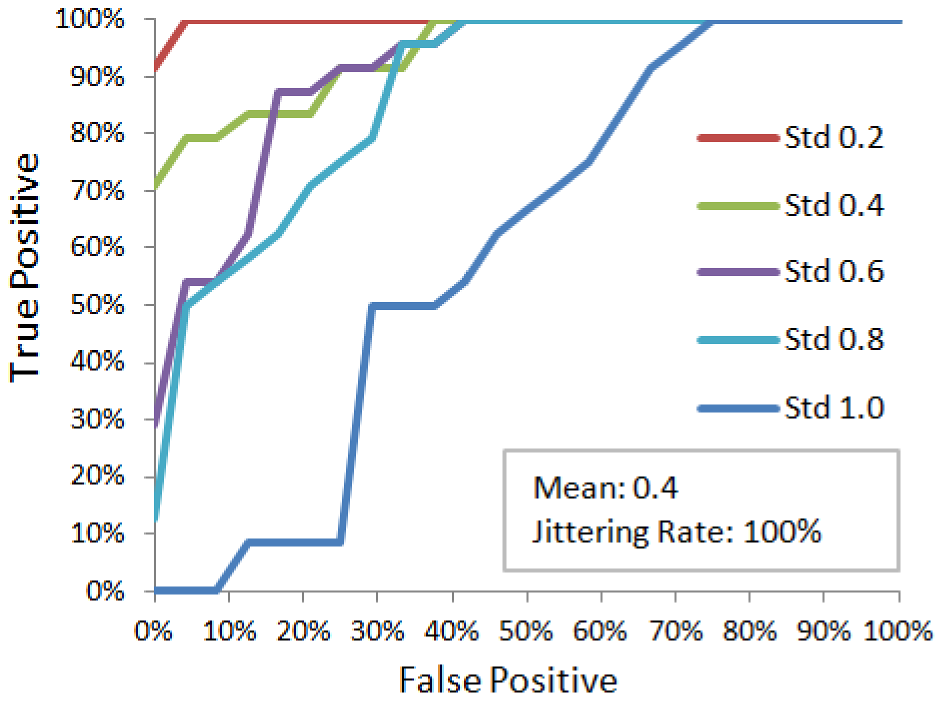


Figure 5.16: ROC Curve Showing the Detection Rate of the OSA Correlation Detecting Algorithm with Different Standard Deviations

Table 5.2: A Comparison of the Number of Successful Detection of the Three Algorithms. This table shows the successful detection (out of 24 test cases) of the algorithms under various standard deviations of jittering

Std	Correlated Algorithm	Jittering Detection	Hybrid Algorithm
0.2	22	7	22
0.4	17	11	20
0.6	7	20	20
0.8	3	22	22
1.0	0	23	23

detected streams by the OSA correlation algorithm is decreasing, as the standard deviation of the jittering increasing from 0.2 to 1.0. This coincides with our intuition that more jittering destroys the correlation-based detection. Fortunately, the jittering detection algorithm can detect more streams as the standard deviation increase from 0.2 to 1.0. In Table 5.2, the last column gives the number of cases successfully detected by the hybrid algorithm.

Using our hybrid stepping-stone host detection algorithm on the same data set, the ROC curve of the framework with a mean of 0.4 and a jittering rate of 100% is shown in Figure 5.17. Similarly, Figure 5.18 shows that the hybrid algorithm can detect almost 90% of the jittered streams with a FP around 5%, and all jittered streams with a FP around 10%. The detection rate is improved significantly compared to the OSA correlation detection algorithm alone. In Figure 5.18, the ROC curve of the hybrid algorithm with mean 0.4 and jittering rate 50% is illustrated. One can see for different standard deviations, all detection rates are higher than 90% with 0% FP. The detection rates all get up to 95% with FP less than 5%.

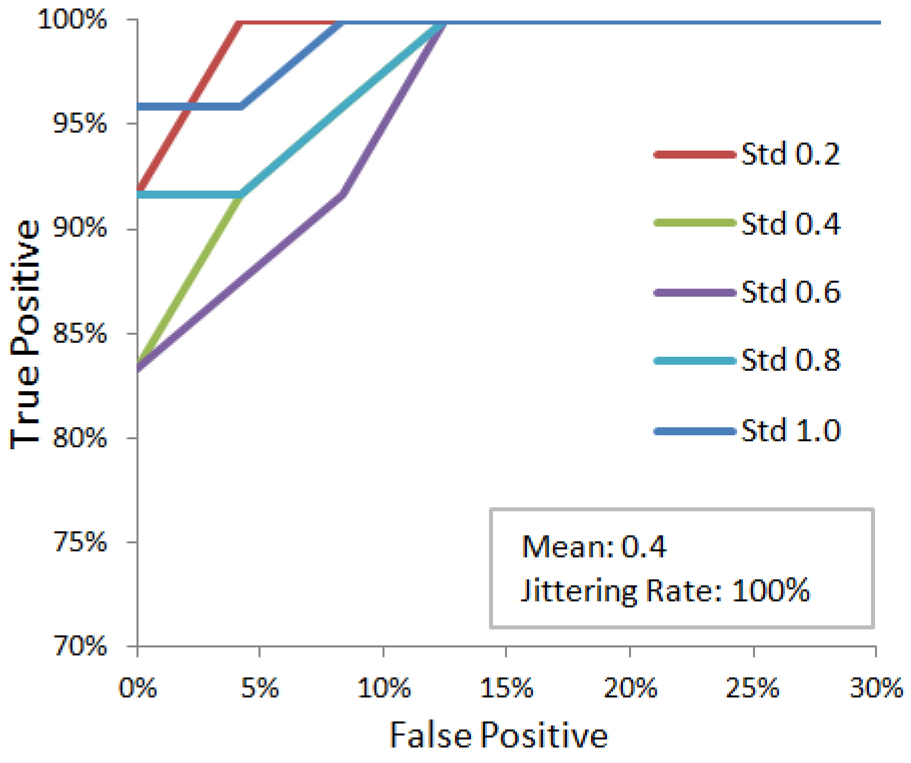


Figure 5.17: ROC Curve of Stepping-stone Detection Framework with Mean 0.4 and Jittering Rate 100%

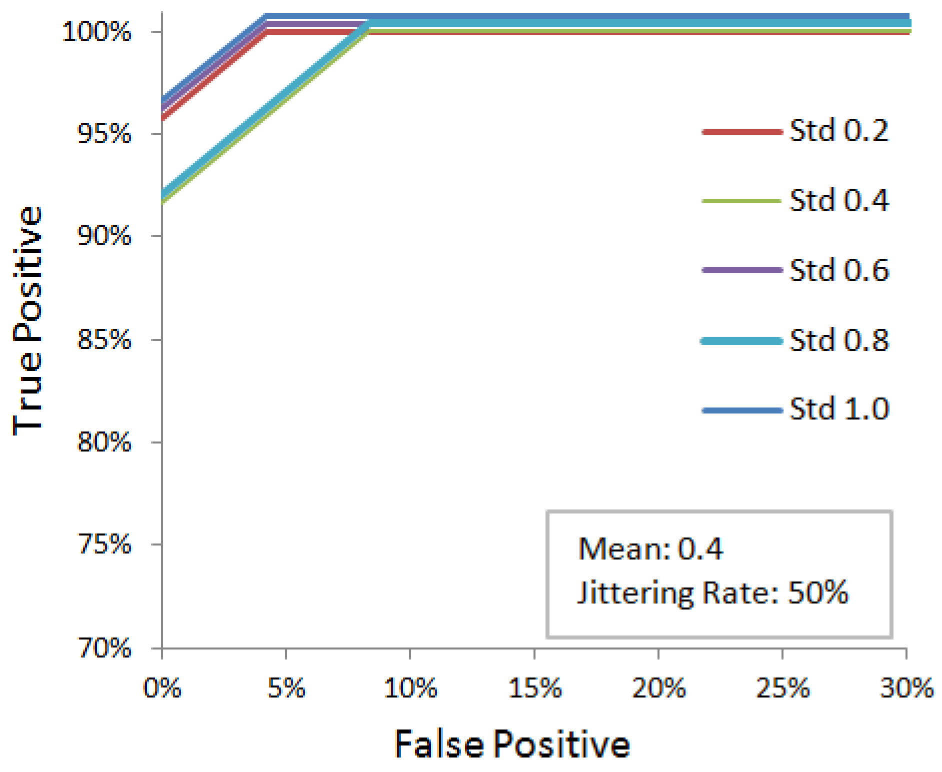


Figure 5.18: ROC Curve of Stepping-stone Detection Framework with Mean 0.4 and Jittering Rate 50%

5.5 Conclusion

In order to evade the detection from correlation-based stepping-stone detection algorithms, intruders may jitter the traffic of a stepping-stone connection chain.

Our jittering detection strategy [139] utilizes statistical distributions to fit the inter-arrival time gaps of traffic flows, followed by measuring the difference and separating jittered ones from normal ones by using SVM. Experimental results show that distributions used to generate jitters do not impact the detection rate of our method. Besides, the mean of randomly generated jitters has very little impact on our detection method. For the other two factors, standard deviation and jittering rate, the detection rate of our method will increase as the standard deviation increases. The more jittering there is the easier it is to detect.

When the traffic is jittered enough, the detection algorithm starts to fail. However, we have observed that the pattern of inter-arrival time gaps of the jittered stream would be changed too. We have found a set of features of the inter-arrival times that can be used to distinguish a jittered traffic from an un-jittered one. Our method can detect most cases based on the combination of all three factors: mean, standard deviation, and jittering rate. Experimental results show the detection rate is around 90% for cases with high standard deviation. However, the jittering detection failed to work when the standard deviation of the jittering is small.

We can distinguish intrusion connections from normal connections by either using a correlation-based algorithm (when jittering is low) or the algorithm presented here (when the jittering is high). In practice, we don't know whether the packet streams have been jittered or not. So we proposed a hybrid stepping-stone detection algorithm to employ both detection algorithms (correlation-based and jitter detections) to detect intrusions.

Thus we proposed a hybrid detection algorithm which combines the OSA correlation detection algorithms and our jittering detection algorithm. This proposed hybrid stepping-stone host detection algorithm can overcome shortages of the jittering detection algorithm and complement correlation-based stepping-stone detection algorithm to deal with jittering evasion techniques. This hybrid algorithm has been shown to be very effective in detecting stepping stones with or without jittering.

Chapter 6

Conclusions

This dissertation focused on addressing the issues of detecting intrusions through stepping-stone connection chains. The first part of our work emphasized intrusion detection at the target server. This issue had not been studied widely. Due to the limitation of monitoring at the end of the connection chain, our research tried to extract useful features from the limited information that is available. Those features had not been investigated enough or taken into considerations by other researchers previously. To detect intrusion through long connection chains at the target server, this dissertation proposed two long connection chain detection algorithms: a nearest-neighbor based algorithm and an anomaly detection-based algorithm. The second part of our work emphasized detecting stepping-stone intrusions at the intermediate host when hackers utilize jittering evasion techniques. This dissertation proposed an effective jittering detection algorithm to detect stepping-stone intrusions with jittering. Besides, a hybrid stepping-stone detection algorithm was also proposed to detect stepping-stone intrusions with or without jittering.

Our nearest neighbor-based long-chain detection algorithm centers around analyzing the delay between the time a user presses enter to finish a command and the time that the user types the next character; it uses an approximated upstream round-trip time to separate a long connection chain from short ones. It's the first time the idea of distinguishing different keystrokes was used. It's also the first time a reliable method was proposed to separate inter-command gaps from intra-command gaps in a SSH connection chain. We discussed the accuracy of separation based on a connection chain's length difference. The experimental results show that our method can correctly distinguish long chains from short chains with an accuracy above 90% when the length difference is larger than 4. Overall, 86% of long chains can be correctly identified with a false positive rate of 13%, without considering the length difference.

Our anomaly detection based long chain detection algorithm centers on utilizing a series of short connections to build the profile inter-command gap curve. For each new connection chain, after extracting the inter-command gap curve, we compare it with the profile and measure the distance. We proposed three different algorithms to quantitatively calculate the distance between the profile curve and the new extracted curve. Our experiments show that by tolerating a false positive rate of 15%, 100% of the test cases (4-hop and 6-hop) can be correctly detected with our weighted ratio distance measurement algorithm which exceeds the expectation of using the uRTTs to estimate true RTTs.

Hackers developed new techniques to evade stepping-stone detection at the intermediate hosts. Hackers can add chaff packets or jitter on original packets to evade detection. Injecting chaffing packets into a connection was discussed and studied in previous research. However, jittering evasion attacks were not studied yet. The way of how to jitter a network traffic stream is totally controlled by hackers, though it's also limited by TCP/IP protocols. This dissertation discussed possible ways of jittering based on different jittering distributions, jittering rates, standard deviation, mean, and so forth. Our proposed jittering detection algorithm [139] utilizes statistical distributions to fit the inter-arrival time gaps of traffic flows. By measuring the difference, it separates jittered ones from normal ones. We have found a set of features of the inter-arrival time gaps that can be used to distinguish a jittered traffic from an un-jittered one. Experimental results show that distributions used to generate jitters do not impact the detection rate of our method. Besides, the mean of randomly generated jitters has very little impact on our detection method. For the other two factors, standard deviation and jittering rate, the detection rate of our method will increase as the standard deviation increases. The more jittering there is, the easier it is to detect. The detection rate is around 90% for cases with a high jittering standard deviation. However, the jittering detection algorithm failed to work when the standard deviation of the jittering is low. In practice, we don't know whether the packet streams have been jittered or not. Hence, we proposed a hybrid stepping-stone detection algorithm which employs a correlation-based OSA detection algorithm and jittering detection algorithm at the same time. This hybrid stepping-stone host detection algorithm can overcome shortages of the jittering detection algorithm and complement correlation-based stepping-stone detection algorithms to deal with jittering evasion techniques. Our hybrid algorithm has been shown to be very effective in detecting stepping stones with or without jittering. Our hybrid detection algorithm can get a higher than 90% detection rate with 0% FP rate in most cases.

We have proposed algorithms to detect stepping-stone intrusions at both the intermediate hosts and the target server. But the stepping-stone intrusion detection problem is far from being completely solved. There are several limitations on our work. For our detection algorithms, threshold values used to determine long or short chains are system and network dependent, impacted by system load and network latency. Each installation of the detection algorithm should find its own threshold value. We provided methods to select these values based on the FP rate. By deploying our detection algorithms in a broader range, with bigger sample and data size, we may refine our detection algorithms with a lower FP rate and a higher detection rate.

With the development of Tor [140] like anonymous proxies, hackers have new robust choices to do remote intrusions through intermediate hosts. Those well developed, strongly encrypted, and widely distributed public proxies can be illegally exploited by hackers. Furthermore, those world-wide distributed proxies are more dynamic than traditional SSH hosts. This will increase the difficulty of implementing stepping-stone detection algorithms. To deal with intrusions from Tor like distributed proxy networks, we need better distributed stepping-stone detection algorithms to be developed and deployed. The throughput and efficiency of the detection algorithm will be highly emphasized. Besides, with the development of big data and machine learning, intelligent, self-driven, and evolving intrusion detection and prevention mechanisms should become a hot research area in the coming years.

Appendix A

Server part of the Socket program for RTT measurement:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(char *msg)
{
    perror(msg);
    exit(1);
}
int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno, clilen;
     char buffer[256];
     struct sockaddr_in serv_addr, cli_addr;
     int n;
     if (argc < 2) {
         fprintf(stderr,"ERROR,␣no␣port␣provided\n");
         exit(1);
     }
     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0)
        error("ERROR␣opening␣socket");
     bzero((char *) &serv_addr, sizeof(serv_addr));
     portno = atoi(argv[1]);
     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);
     if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
              error("ERROR␣on␣binding");
     listen(sockfd,5);
     clilen = sizeof(cli_addr);
     newsockfd = accept(sockfd,
                (struct sockaddr *) &cli_addr,
                &clilen);
     if (newsockfd < 0)
          error("ERROR on accept");
     bzero(buffer,256);
     n = read(newsockfd,buffer,255);
     if (n < 0) error("ERROR reading from socket");
     printf("Test message as %s received.\n",buffer);
     //n = write(newsockfd,"Message returned.",18);
     n = write(newsockfd,buffer,255);
     if (n < 0) error("ERROR writing to socket");
     return 0;
}
```

Client part of the Socket program for RTT measurement:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/time.h>

void error(char *msg)
{
    perror(msg);
    exit(0);
}
int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    struct timeval tv1, tv2;
    double start, finish;

    char buffer[256];
    if (argc < 3) {
       fprintf(stderr,"usage %s hostname port\n", argv[0]);
       exit(0);
    }
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR␣opening␣socket");
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR,␣no␣such␣host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
         (char *)&serv_addr.sin_addr.s_addr,
         server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR␣connecting");
    //printf("Please enter the message: ");
    bzero(buffer,256);
    //fgets(buffer,255,stdin);
    char message[256];
    memset(message, 0, sizeof(message));
    strcpy(message, "1");
    gettimeofday(&tv1, NULL);
    n = write(sockfd, message, strlen(message));
    if (n < 0)
         error("ERROR␣writing␣to␣socket");
    //bzero(buffer,256);
    n = read(sockfd,buffer,255);
    gettimeofday(&tv2, NULL);
    if (n < 0)
         error("ERROR␣reading␣from␣socket");

    if (strcmp(message, buffer) != 0) {
         printf("Send␣and␣receive␣does␣not␣match.");
    }
    start = tv1.tv_sec*1000.0 + (tv1.tv_usec/1000.0);
    finish = tv2.tv_sec*1000.0 + (tv2.tv_usec/1000.0);
    printf("RTT␣is␣%.9lf␣by␣milliseconds.\n", finish-start);
```

```
    //printf("%s\n",buffer);
    return 0;
}
```

Appendix B

Keystroke Analysis:

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <fstream>
#include <vector>
#include <sstream>
#include <iterator>
#include <algorithm>
#include <cmath>
#include <exception>

using namespace std;

double *sort_incr(double *array, int length) {
    double *sorted = new double[length];
    double temp;
    for(int i=0; i< length; i++) {
        sorted[i] = array[i];
        cout << sorted[i] << endl;
    }
    for( int i=0; i< length-1; i++) {
        for( int j=0; j< length-1; j++) {
            if(sorted[j] > sorted[j+1]) {
                temp = sorted[j];
                sorted[j] = sorted[j+1];
                sorted[j+1] = temp;
            }
        }
    }
    return sorted;
}

vector<double> sort_incr(vector<double> a) {
    vector<double> b = a;
    double temp;
    for( unsigned int i=0; i< b.size()-1; i++) {
        for( unsigned int j=0; j< b.size()-1; j++) {
            if(b[j] > b[j+1]) {
                temp = b[j];
                b[j] = b[j+1];
                b[j+1] = temp;
            }
        }
    }
    return b;
}

struct eRTT {
    double typingspeed; // Typing speed of all keystrokes
    double typingspeed_inside; // Typing speed inside
       commands
    double commdelay; // Average gap between commands
    double nRTT; // Nearest RTT
    string keystrokes; // Output file for distribution of
       keystroke gaps
    string commands; // Output file for distribution of
       command gaps
    string numbers; // Output file for NO. of the packet
       line been found as a new command
    string distribution; // Output file for the
       distribution of comm gap from 0 to 2s by range like
       [0-0.1)
    string keys_inside; // Output file for the distribution
        of keystrokes inside the commands
    string allgaps; // Added on Nov. 12, 2009 to use the
       response and request gap
};

class RTT {
public:
    void pcap_trim(string& input, char delim);
    vector<string> pcap_split(string input, char delim);
    string pcap_timestamp(string& input);
    string pcap_number(string& input);
    void keystroke_analyze(string input, int OFFSET, eRTT*
       results);
};

void RTT::pcap_trim(string& input, char delim) {
    // Trim the starting and ending blank spaces.
    string::size_type start;
    string::size_type end;
    start = input.find_first_not_of(delim);
    input.erase(0, start);
    end = input.find_last_not_of(delim);
    input.erase(end+1);
}

vector<string> RTT::pcap_split(string input, char delim) {
    // Split string based on the delim you set.
    vector<string> input_splitted;
    string temp;
    stringstream ss(input);
    while(getline(ss, temp, delim)) {
        input_splitted.push_back(temp);
    }
    return input_splitted;
}

string RTT::pcap_timestamp(string& input) {
    // Split the packets line from txt format by whitespace
       , and output the timestamp.
    char delim = '␣';
    RTT::pcap_trim(input, delim);
    string timestamp = RTT::pcap_split(input, delim)[1];
    return timestamp;
}

string RTT::pcap_number(string& input) {
    // Must be used after the pcap_timestamp, because
       pcap_timestamp trim the starting and ending blank
       spaces.
    char delim = '␣';
    string number = RTT::pcap_split(input, delim)[0];
    return number;
}

void RTT::keystroke_analyze(string input, int OFFSET, eRTT*
    results) {
    ifstream in(input.c_str());
    string line;
    int count = 0;
    int cnt = 0; // Count after OFFSET for request packet
    vector<double> request;
    vector<double> comm_gap;
    vector<double> inside_gap;
    vector<double> all_gap;
    string::size_type found_send;
    string::size_type found_echo;
    string::size_type found_ack;
    int nRTT_cnt = 0;
    double nRTT_all = 0;
    int nRTT_flag = 2;
    string timestamp;
    double gap;
    double gap_inside;
    double delay_all = 0; // Accumulate delay between all
       commands
    double time_inside = 0; // Accumulate time inside
       commands
    int cnt_gap = 0;
    int count_echo = 0;
    double t1 = 0;
    double t2 = 0;
    double t_echo1 = 0;
    double t_echoL = 0;
    double typing_all = 0;
    string number;
    vector<string> nums; // Used to store number of the
       packet line
    int gap_flag = 0;
    // Calculate the gap from the 1st echo response to the
       next request.
```

```
// If set to 0, calculate the gap from the LAST echo
   response to the next request.
double gap_threshold = 3.0; // If the calculated
   comm_gap is larger than this threshold, it will be
   discarded.
int thr_flag = 1;
// If flag set to 1, any comm_gap larger than threshold
    will be discarded.

while(getline(in, line)) {
    if(count >= OFFSET) {
        found_send = line.find("Encrypted␣request␣
           packet");
        found_echo = line.find("Encrypted␣response␣
           packet");
        found_ack = line.find(">␣ssh␣[ACK]");
        // Using t1 and t2 to store the current and
           last timestamp for nRTT
        t1 = t2;
        t2 = atof(RTT::pcap_timestamp(line).c_str());
        number = RTT::pcap_number(line);
        if((int)found_send > 0) {
            nRTT_flag = 0;
            // Insert into request
            timestamp = RTT::pcap_timestamp(line);
            request.push_back(atof(timestamp.c_str()));
            if(count_echo >= 2) {
                // Insert into new_comm
                cnt_gap ++;
                //gap = request[request.size()-1] -
                   request[request.size()-2];
                if( gap_flag == 1) {
                    gap = request[request.size()-1] -
                       t_echo1;
                } else {
                    gap = request[request.size()-1] -
                       t_echoL;
                }
                delay_all += gap;
                comm_gap.push_back(gap);
                all_gap.push_back(gap);
                nums.push_back(number);
            } else if(count_echo < 2) {
                cnt ++;
                if(cnt > 1) {
                    //gap_inside = request[request.size
                       ()-1] - request[request.size()
                       -2];
                    if( gap_flag == 1) {
                        gap_inside = request[request.
                           size()-1] - t_echo1;
                    } else {
                        gap_inside = request[request.
                           size()-1] - t_echoL;
                    }
                    time_inside += gap_inside;
                    inside_gap.push_back(gap_inside);
                    all_gap.push_back(gap_inside);
                }
            }
            count_echo = 0;
        } else if((int)found_echo > 0) {
            count_echo++;
            if(count_echo == 1) {
                t_echo1 = atof(RTT::pcap_timestamp(line
                   ).c_str());
            }
            t_echoL = atof(RTT::pcap_timestamp(line).
               c_str());
            nRTT_flag++;
        } else if((int)found_ack > 0) {
            if(nRTT_flag == 1) {
                nRTT_cnt ++;
                nRTT_all += t2-t1;
            }
            nRTT_flag += 3;
        }
    }
    count ++;
}
```

```
results->typingspeed_inside = time_inside/(cnt-1);
results->commdelay = delay_all/cnt_gap;
results->nRTT = nRTT_all/nRTT_cnt;
ofstream out_k(results->keystrokes.c_str());
double temp;
for(unsigned int i=1; i < request.size(); i++) {
    temp = request[i] - request[i-1];
    out_k << temp << "\n";
    typing_all += temp;
}
out_k.close();
ofstream out_g(results->allgaps.c_str());
for(unsigned int i=0; i < all_gap.size(); i++) {
    out_g << all_gap[i] << "\n";
}
out_g.close();
results->typingspeed = typing_all/(request.size()-1);
ofstream out_c(results->commands.c_str());

// Check the thr_flag to decide if discard any comm_gap
    larger than the gap_threshold set before
int cnt_over = 0;
if (thr_flag) {
    for(unsigned int i=0; i < comm_gap.size(); i++) {

        if (comm_gap[i] <= gap_threshold) {
            out_c << comm_gap[i] << "\n";
        } else {
            cnt_over ++;
        }
    }
} else {
    for(unsigned int i=0; i < comm_gap.size(); i++) {
        out_c << comm_gap[i] << "\n";
    }
}

out_c.close();
ofstream out_n(results->numbers.c_str());
```

```
for(unsigned int i=0; i < nums.size(); i++) {
    out_n << nums[i] << "\n";
}
out_n.close();
ofstream out_i(results->keys_inside.c_str());
for(unsigned int i=0; i < inside_gap.size(); i++) {
    out_i << inside_gap[i] << "\n";
}
out_i.close();

// Sort the commands, and find the biggest increasing
vector<double> sorted_comm;
sorted_comm = sort_incr(comm_gap);

// Calculate the distribution of comm gap from 0 to 2s
   by 0.1s as each range, such as [0-0.1), [0.1-0.2)
int distribute[20];
for(int i =0; i<20; i++) {
    distribute[i] = 0;
}

int index = 0;
for(unsigned int i=0; i < sorted_comm.size(); i++) {
    if(sorted_comm[i] < 2.0) {
        index = (int)floor(sorted_comm[i]*10);
        distribute[index] ++;
    }
}
ofstream out_d(results->distribution.c_str());
    for(unsigned int i=0; i < sizeof(distribute)/sizeof
       (int); i++) {
        out_d << distribute[i] << "\n";
    }
out_d.close();

vector<double> comm_incr;
for(unsigned int i=0; i< sorted_comm.size()-1; i++) {
    comm_incr.push_back(sorted_comm[i+1] - sorted_comm[
       i]);
```

```
        //cout << comm_incr[i] << endl;
    }

    double max_gap = 0;
    // Find the maximun gap
    int checkpoint = comm_incr.size()/2;
    for( unsigned int i=0; i<comm_incr.size(); i++) {
        max_gap = (comm_incr[i] < max_gap) ? max_gap :
           comm_incr[i];
    }

    cout << "Average␣typing␣speed␣is␣" << results->
       typingspeed << endl;
    cout << "Average␣command␣delay␣is␣" << results->
       commdelay << endl;
    cout << "Average␣speed␣inside␣commands␣is␣" << results
       ->typingspeed_inside << endl;
    cout << "Nearest␣RTT␣is␣" << results->nRTT << endl;

    if (thr_flag) {
        cout << cnt_over << "␣command␣gaps␣larger␣than␣the␣
           threshold␣" << gap_threshold << "␣are␣discarded.
           " << endl;
    }
}
```

Bibliography

[1] Herbert Lin. *Cryptography's Role in Securing the Information Society*. National Academies Press, 1996.

[2] The Security Practitioner. An introduction to information security. `http://security.practitioner.com/introduction/infosec_2.htm/`. Online accessed 10-March-2014.

[3] Mariana Hentea. A perspective on achieving information security awareness. *Informing Science: International Journal of an Emerging Transdiscipline*, 2:169–178, 2005.

[4] Robert R. Moeller. Information Security: Basic to Specialized Topics. *Information Security Journal: A Global Perspective*, 1:33–37, 1993.

[5] Olaf Tettero. Intrinsic information security - embedding security issues in the design process of telematics systems. *Solid State Ionics*, 2000.

[6] Margareth Stoll. E-learning promotes information security. In *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education*, pages 309–314. Springer, 2008.

[7] Dennis Longley William Caelli and Michael Shain. Information Security Handbook. *Computer Law and Security Report*, 8, 1992.

[8] Veronika Durcekova, Ladislav Schwartz, and Nahid Shahmehri. Sophisticated denial of service attacks aimed at application layer. In *ELEKTRO, 2012*, pages 55–60. IEEE, 2012.

[9] Vrizlynn L Thing, Morris Sloman, and Naranker Dulay. A survey of bots used for distributed denial of service attacks. In *New Approaches for Security, Privacy and Trust in Complex Environments*, pages 229–240. Springer, 2007.

[10] Zhang Shu and Partha Dasgupta. Denying denial-of-service attacks: A router based solution. In *International Conference on Internet Computing*, pages 301–307, 2003.

[11] Felix Lau, Stuart H. Rubin, Michael H. Smith, and Ljiljana Trajkovic. Distributed denial of service attacks. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 3, pages 2275–2280, 2000.

[12] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Detecting distributed denial of service attacks using source IP address monitoring. In *International Conference on Computer Science and Software Engineering*, pages 771–782, 2002.

[13] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Proactively detecting distributed denial of service attacks using source IP address monitoring. In *Networking*, pages 771–782, 2004.

[14] Andrew Simmonds, Peter Sandilands, and Louis Van Ekert. An ontology for network security attacks. In *Applied Computing*, pages 317–323. Springer, 2004.

[15] Network security. Network security. `http://en.wikipedia.org/wiki/Network_security`. Online accessed 10-March-2014.

[16] H. S. Javitz and A. Valdes. The nides statistical component: Description and justification. In *Technical Report, Computer Science Laboratory, SRI International*, 1993.

[17] Peter G. Neumann and Phillip A. Porras. Experience with emerald to date. In *1st USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 73–80, April 1999.

[18] Stefan Axelsson. *Intrusion Detection Systems: A Survey and Taxonomy.* Department of Computer Engineering, Chalmers University, March 2000.

[19] Richard P. Lippmann. Evaluating intrusion detection systems: Darpa 1998 off-line intrusion detection evaluation. In *Proceedings of DARPA Information Survivability Conference and Exposition*, pages 12–26, 2000.

[20] Hazem M. El-Bakery and Nikos Mastorakis. A real-time intrusion detection algorithm for network security. *WSEAS Transactions on Communications*, 7(12):1222–1228, 2008.

[21] Vladimir I. Gorodetski, Igor V. Kotenko, and Oleg Karsaev. Multi-agent technologies for computer network security: attack simulation, intrusion detection and intrusion detection learning. *Computer Systems: Science and Engineering*, 18:191–200, 2003.

[22] Prachi Jain, Pramod Kumar Singh, and Ajith Abraham. Intrusion detection and self healing model for network security. In *Next Generation Web Services Practices (NWeSP), 2011 7th International Conference on*, pages 320–325. IEEE, 2011.

[23] Mei-Ling Shyu, Shu-Ching Chen, Kanoksri Sarinnapakorn, and LiWu Chang. A novel anomaly detection scheme based on principal component classifier. 2003.

[24] Bill Hancock. Automated intrusion detection systems and network security. *Network Security*, 1:14–15, 1998.

[25] Moon Sun Shin and Kyeong Ja Jeong. An alert data mining framework for network-based intrusion detection system. pages 38–53, 2005.

[26] Hong Han, Xian-Liang Lu, and Li-Yong Ren. Using data mining to discover signatures in network-based intrusion detection. 2002.

[27] Dong Seong Kim and Jong Sou Park. Network-based intrusion detection with support vector machines. pages 747–756, 2003.

[28] Moon Sun Shin, Eun Hee Kim, and Keun Ho Ryu. *False Alarm Classification Model for Network-Based Intrusion Detection System.* 2004.

[29] Danny Hesse, Jana Dittmann, and Andreas Lang. Network Based Intrusion Detection to Detect Steganographic Communication Channels - On the Example of Images. In *Conference on Software Engineering and Advanced Applications*, pages 453–456, 2004.

[30] S. Staniford-Chen and L. Heberlein. Holding intruders accountable on the internet. In *Proceeding of IEEE Symposium on Security and Privacy*, pages 39–49, May 1995.

[31] Y. Zhang and V. Paxson. Detecting stepping stones. In *Proceedings of the 9th Conference on USENIX Security Symposium*, volume 9, pages 171–184, 2000.

[32] Baris Coskun and Nasir D. Memon. Online Sketching of Network Flows for Real-Time Stepping-Stone Detection. In *Annual Computer Security Applications Conference*, pages 473–483, 2009.

[33] Ping Li, Wanlei Zhou, and Yini Wang. Getting the real-time precise round-trip time for stepping stone detection. In *International Conference on Network and System Security*, pages 377–382, 2010.

[34] Han-Wei Hsiao and Wei-Cheng Fan. Detecting stepstone with network traffic mining approach. In *International Conference on Innovative Computing, Information and Control*, pages 1176–1179, 2009.

[35] Anming Xie, Cong Tang, Nike Gui, Zhuhua Cai, Jianbin Hu, and Zhong Chen. An adjacency matrixes-based model for network security analysis. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–5, 2010.

[36] Mohd Nizam Omar, Lelyzar Siregar, and Rahmat Budiarto. Hybrid stepping stone detection method. In *International Conference on Distributed Framework and Applications, DFmA*, pages 134–138, 2008.

[37] Ping Li, Wanlei Zhou, and Yanli Yu. A Quick-Response Real-Time Stepping Stone Detection Scheme. In *High Performance Computing and Communications*, pages 677–682, 2010.

[38] Abhinay Kampasi, Yin Zhang, Giovanni Di Crescenzo, Abhrajit Ghosh, and Rajesh Talpade. *Improving Stepping Stone Detection Algorithms using Anomaly Detection Techniques*. Computer Science Department, University of Texas at Austin, 2007.

[39] S. Ansari, S. G. Rajeev, and H. S. Chandrashekar. Packet sniffing: a brief introduction. *IEEE Potentials*, 21:17–19, 2002.

[40] Joshua Broadway, Benjamin Turnbull, and Jill Slay. Improving the Analysis of Lawfully Intercepted Network Packet Data Captured for Forensic Analysis. In *Availability, Reliability and Security*, pages 1361–1368, 2008.

[41] Xiaohong Yuan, Percy Vega, Jinsheng Xu, Huiming Yu, and Yaohang Li. Using packet sniffer simulator in the class: experience and evaluation. In *ACM Southeast Regional Conference*, pages 116–121, 2007.

[42] Jim S. Tiller and Bryan D. Fish. Packet Sniffers and Network Monitors. *Information Systems Security*, 9:1–8, 2000.

[43] Semyon Litvinov and Abdullah Alhamamah. Using a packet sniffer to analyze the efficiency and power of encryption techniques used to protect data over a computer network. `http://www.micsymposium.org/mics_2001/litvinov_guster.pdf`. Online accessed August-2014.

[44] Internet Engineering Task Force. Request for comments (rfc). `https://www.ietf.org/rfc.html`. Online accessed 10-March-2014.

[45] Tim Carstens. Programming with pcap. `http://www.tcpdump.org/pcap.html`. Online accessed 10-March-2014.

[46] LBNL's Network Research Group. Libpcap api. `http://ee.lbl.gov/`. Online accessed 10-March-2014.

[47] WinPcap Team. Introduction to winpcap. `http://www.winpcap.org/docs/default.htm`. Online accessed 10-March-2014.

[48] Wikipedia. Tcp dump. `http://en.wikipedia.org/wiki/Tcpdump`. Online accessed 10-March-2014.

[49] The Tcpdump team. Tcp dump. `http://www.tcpdump.org/manpages/tcpdump.1.html`. Online accessed 10-March-2014.

[50] WinPcap Team. Tcpdump for windows using winpcap. `http://www.winpcap.org/windump/default.htm`. Online accessed 10-March-2014.

[51] Steven Mccanne and Van Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *USENIX Technical Conference*, pages 259–270, 1993.

[52] Wireshark Foundation. Wireshark network protocol analyzer. `http://wiki.wireshark.org/`. Online accessed 10-March-2014.

[53] Vasil Hnatyshin and Andrea F. Lobo. Undergraduate data communications and networking projects using opnet and wireshark software. In *Technical Symposium on Computer Science Education*, pages 241–245, 2008.

[54] G. Munz and Georg Carle. Distributed Network Analysis Using TOPAS and Wireshark. In *IEEE Network Operations and Management Symposium Workshops*, pages 161–164, 2008.

[55] Usha Banerjee, Ashutosh Vashishtha, and Mukul Saxena. Evaluation of the capabilities of wireshark as a tool for intrusion detection. *International Journal of Computer Applications*, 6(7):1–5, 2010.

[56] Wireshark Foundation. What's up with the name change? `http://www.wireshark.org/faq.html#q1.2`. Online accessed 10-March-2014.

[57] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In *Recent Advances in Intrusion Detection*, pages 203–222, 2004.

[58] Animesh Patcha and Jung min Park. Network anomaly detection with incomplete audit data. *Computer Networks*, 51:3935–3955, 2007.

[59] Juan M. Estevez-Tapiador, Pedro Garc ia Teodoro, and Jesus E. D iaz Verdejo. N3: A geometrical approach for network intrusion detection at the application layer. In *Computational Science and Its Applications, ICCSA 2004*, pages 841–850, 2004.

[60] Roberto Perdisci, Davide Ariu, Prahlad Fogla, Giorgio Giacinto, and Wenke Lee. McPAD: A multiple classifier system for accurate payload-based anomaly detection. *Computer Networks*, 53:864–881, 2009.

[61] Neminath Hubballi, Santosh Biswas, and Sukumar Nandi. Layered Higher Order N-grams for Hardening Payload Based Anomaly Intrusion Detection. In *Availability, Reliability and Security*, pages 321–326, 2010.

[62] Sun il Kim and Nnamdi Nwanze. Noise-Resistant Payload Anomaly Detection for Network Intrusion Detection Systems. In *Power Conversion Conference*, pages 517–523, 2008.

[63] DH Summerville, N Nwanze, and VA Skormin. Anomalous packet identification for network intrusion detection. In *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*, pages 60–67. IEEE, 2004.

[64] Ill young Weon, Doo Heon Song, and Chang hoon Lee. Effective intrusion detection model through the combination of a signature-based intrusion detection system and a machine learning-based intrusion detection system. *Journal of Information Science and Engineering*, 22:1447–1464, 2006.

[65] Mueen Uddin and Azizah Abdul Rahman. Dynamic multi layer signature based intrusion detection system using mobile agents. *Computing Research Repository*, abs/1010.5, 2010.

[66] Christopher Kruegel and Thomas Toth. Using decision trees to improve signature-based intrusion detection. In *Recent Advances in Intrusion Detection*, pages 173–191. Springer, 2003.

[67] Derek Pao, Nga Lam, and Ray C. C. Cheung. A memory-based NFA regular expression match engine for signature-based intrusion detection. *Computer Communications*, 36:1255–1267, 2013.

[68] Farooq Anjum, Dhanant Subhadrabandhu, and Saswati Sarkar. Signature based intrusion detection for wireless ad-hoc networks: a comparative study of various routing protocols. In *Vehicular Technology, IEEE Conference*, volume 3, pages 2152–2156, 2003.

[69] Meharouech Sourour, Bouhoula Adel, and Abbes Tarek. Network security alerts management architecture for signature-based intrusions detection systems within a NAT environment. *Journal of Network and Systems Management*, 19:1–24, 2011.

[70] Tamer F. Badran, Hany H. Ahmad, and Mohamad Abdelgawad. A reconfigurable multi-byte regular-expression matching architecture for signature-based intrusion detection. In *International Conference on Information and Communication Technologies: From Theory to Applications*, pages 1–4, 2008.

[71] Zhuowei Li, Amitabha Das, and Jianying Zhou. USAID: Unifying signature-based and anomaly-based intrusion detection. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 702–712, 2005.

[72] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *EuroSys Conference*, pages 15–27, 2006.

[73] Bernhard Amann Vern Paxson and Justin Azoff. The bro network security monitor. `http://www.bro.org/`. Online accessed 10-March-2014.

[74] Daniel B. Cid and Jia-Bing Cheng. Ossec: Open source host-based intrusion detection system. `http://www.ossec.net/`. Online accessed 10-March-2014.

[75] Juniper Networks. Juniper srx series. `http://chimera.labs.oreilly.com/books/1234000001633`. Online accessed 10-March-2014.

[76] IBM. Ibm security network intrusion prevention system. `http://www-03.ibm.com/software/products/en/network-ips/`. Online accessed 10-March-2014.

[77] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *USENIX Systems Administration Conference*, pages 229–238, 1999.

[78] Martin Roesch and Sourcefire. Snort: An open source network intrusion prevention and detection system. `http://www.snort.org/`. Online accessed 10-March-2014.

[79] X. Wang and D. Reeves. Sleepy watermark tracing: An active network-based intrusion response framework. In *Proc. of the 16th International Information Security Conference*, pages 369–384, 2001.

[80] G. Zhao and J. Yang. Correlating tcp/ip interactive sessions with correlation coefficient to detect stepping-stone intrusion. In *International Conference on Advanced Information Networking and Applications*, pages 546–551, 2009.

[81] T. He and L. Tong. A signal processing perspective to stepping-stone detection. In *CISS*, pages 687–692, 2006.

[82] T. He and L. Tong. Detecting encrypted interactive stepping-stone connections. In *IEEE Transactions on Signal Processing*, volume 55, pages 1612–1623, May 2007.

[83] T. He and L. Tong. Detecting encrypted interactive stepping-stone connections. In *Proc. 2006 IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 33–38, May 2006.

[84] T. He and P. Venkitasubramaniam. Packet scheduling against stepping-stone attacks with chaff. In *Military Communications Conference*, pages 1–7, 2006.

[85] M.D. Wan and S. Huang. Finding the longest similar subsequence of thumbprints for intrusion detection. In *20th International Conference on Advanced Information Networking and Applications*, pages 255–262, 2006.

[86] K. H. Yung. Detecting long connection chains of interactive terminal sessions. In *Recent Advances in Intrusion Detection, Lecture Notes in Computer Science*, pages 1–16, 2002.

[87] J. Yang and S. Huang. A real-time algorithm to detect long connection chains of interactive terminal sessions. In *Proceedings of the 3rd International Conference on Information Security, ACM*, pages 198–203, 2004.

[88] J. Yang and S. Huang. Matching tcp packets and its application to the detection of long connection chains on the internet. In *19th International Conference on Advanced Information Networking and Applications*, pages 1005–1010, 2005.

[89] J. Yang and S. Huang. A clustering-partitioning algorithm to find tcp packet round-trip time for intrusion detection. In *20th International Conference on Advanced Information Networking and Applications*, pages 231–236, 2006.

[90] A. G. Donoho and S. Staniford. Multiscale stepping-stone detection: Detecting pairs of jittered interactive streams by exploiting maximum tolerable delay. In *5th International Symposium on Recent Advances in Intrusion Detection, LNCS 2516*, pages 17–35. Springer, 2002.

[91] Y.-W. Kuo and S. Huang. Stepping-stone detection algorithm based on order preserving mapping. In *International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1–8, 2007.

[92] Y.-W. Kuo and S. Huang. An algorithm to detect stepping-stones in the presence of chaff packets. In *14th IEEE International Conference on Parallel and Distributed Systems*, pages 485–492, 2008.

[93] B. Lee J. Yang and S. Huang. Monitoring network traffic to detect stepping-stone intrusion. In *22nd International Conference on Advanced Information Networking and Applications - Workshops*, pages 56–61, 2008.

[94] A. Blum and D. Song. Detection of interactive stepping-stones: Algorithms and confidence bounds. In *Proceedings of International Symposium on Recent Advance in Intrusion Detection (RAID)*, pages 20–35, September 2004.

[95] L. Zhang and A. G. Persaud. Detection of stepping stone attack under delay and chaff perturbations. In *25th IEEE International Conference on Performance, Computing, and Communications Conference (IPCCC)*, pages 246–256, 2006.

[96] Intrepidus Group. Mallory: Transparent tcp and udp proxy. `https://intrepidusgroup.com/insight/mallory/`. Online accessed 10-March-2014.

[97] Mitmproxy Team. Interactive and ssl-capable mitm proxy. `http://mitmproxy.org/doc/index.html`. Online accessed 10-March-2014.

[98] Portswigger Web Security. Burp intercepting proxy server. `http://portswigger.net/burp/proxy.html`. Online accessed 10-March-2014.

[99] Matasano and s7ephen. Matasano port forwarding interceptor. `http://www.woodmann.com/collaborative/tools/index.php/Matasano_Port_Forwarding_Interceptor`. Online accessed 10-March-2014.

[100] Massimiliano Montoro. Cain and abel. `http://www.oxid.it/cain.html`. Online accessed 10-March-2014.

[101] Philippe Biondi. Scapy: A packet manipulation tool. `http://www.secdev.org/projects/scapy/`. Online accessed 10-March-2014.

[102] Justin Searle and Matt Carpenter. Middler: Man in the middle tool. `https://code.google.com/p/middler/`. Online accessed 10-March-2014.

[103] Geeknet Inc. Ettercap: Multipurpose sniffer. `http://ettercap.sourceforge.net/index.php`. Online accessed 10-March-2014.

[104] Paros and Yukusan. Paros: Http/https proxy. `http://sourceforge.net/projects/paros/`. Online accessed 10-March-2014.

[105] Rogan Dawes. Owasp webscarab. `https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project#tab=Main`. Online accessed 10-March-2014.

[106] Fresh Ports. Netsed: Real time packets tampering. `http://www.freshports.org/net/netsed/`. Online accessed 10-March-2014.

[107] Squid Software Foundation. Squid: Caching proxy for the web. `http://www.squid-cache.org/`. Online accessed 10-March-2014.

[108] Qiong Li and David L. Mills. On the long-range dependence of packet round-trip delays in internet. In *Proceedings of International Conference on Communications (ICC98)*, pages 1185–1192, June 1998.

[109] The Tcpdump team. http://www.tcpdump.org/.

[110] Thomas Williams and Colin Kelly. Gnuplot: a portable command-line driven graphing utility. `http://http://www.gnuplot.info/`. Online accessed 10-March-2014.

[111] Wei Ding and Stephen Huang. Detecting stepping-stone intruders with long connection chains. In *the 5th Intl. Conference on Information Assurance and Security*, pages 665–669, 2009.

[112] Wei Ding and Stephen Huang. Detecting Stepping-Stone Intruders with Long Connection Chains. *Journal of Information Assurance and Security*, 5:500–509, 2010.

[113] Wei Ding and Stephen Huang. Detecting intruders using a long connection chain to connect to a host. In *the 25th IEEE Intl. Conference on Advanced Information Networking and Applications*, pages 121–128, 2011.

[114] Internet Crime Complaint Center. 2013 ic3 internet crime annual report. `http://www.ic3.gov/media/annualreport/2013_IC3Report.pdf`. Online accessed 10-March-2014.

[115] Xavier Anguera, Robert Macrae, and Nuria Oliver. Partial sequence matching using an Unbounded Dynamic Time Warping algorithm. In *International Conference on Acoustics, Speech, and Signal Processing*, pages 3582–3585, 2010.

[116] Hagen Kaprykowsky and Xavier Rodet. Globally optimal short-time dynamic time warping, application to score to audio alignment. In *International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 1–5, 2006.

[117] A. M. Youssef, T. K. Abdel-Galil, E. F. El-Saadany, and M. M. A. Salama. Disturbance classification utilizing dynamic time warping classifier. volume 19, pages 272–278, 2004.

[118] Andrés Marzal and Vicente Palazón. Dynamic time warping of cyclic strings for shape matching. In *Pattern Recognition and Image Analysis*, pages 644–652. Springer, 2005.

[119] Andrea Corradini. Dynamic time warping for off-line recognition of a small gesture vocabulary. In *International Workshop on Recognition, Analysis, and Tracking of Faces and Gestures in Real-Time Systems*, pages 82–89, 2001.

[120] Eamonn Keogh and Chotirat Ann Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and information systems*, 7(3):358–386, 2005.

[121] Cory S. Myers and Lawrence R. Rabiner. A level building dynamic time warping algorithm for connected word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29:284–297, 1981.

[122] Ying-Wei Kuo, Shou-Hsuan Stephen Huang, Wei Ding, Rebecca Kern, and Jianhua Yang. Using dynamic programming techniques to detect multi-hop stepping-stone pairs in a connection chain. In *Advanced Information Networking and Applications*, pages 198–205, 2010.

[123] Shou-Hsuan Stephen Huang and Ying-Wei Kuo. Detecting chaff perturbation on stepping-stone connection. In *International Conference on Parallel and Distributed Systems*, pages 660–667, 2011.

[124] A. Jian. *Cyber Crime: Issues and Threats*. Gyan Publishing House, 2005.

[125] Wei Ding and Stephen Huang. Detecting stepping-stones under the influence of packet jittering. In *the 9th International Conference on Information Assurance and Security*, pages 31–36, 2013.

[126] Paul Barford and David Plonka. Characteristics of network traffic flow anomalies. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 69–73. ACM, 2001.

[127] Becchi Michela. From poisson processes to self-similarity: a survey of network traffic models. `http://www1.cse.wustl.edu/~jain/cse567-06/ftp/traffic_models1/`. Online accessed August-2014.

[128] A. Adas. Traffic models in broadband networks. *IEEE Communications Magazine*, 35:82–89, 1997.

[129] Allen B. Downey. Lognormal and Pareto distributions in the Internet. *Computer Communications*, 28:790–801, 2005.

[130] I. Antoniou, V. V. Ivanov, and P. V. Zrelov. On the log-normal distribution of network traffic. *Physica D-nonlinear Phenomena*, 167:72–85, 2002.

[131] Amp Bhattacharjee and Sukumar Nandi. Statistical analysis of network traffic inter-arrival. In *Advanced Communication Technology (ICACT), 2010 The 12th International Conference on*, volume 2, pages 1052–1057. IEEE, 2010.

[132] Deniz Ersoz, Mazin S. Yousif, and Chita R. Das. Characterizing Network Traffic in a Cluster-based, Multi-tier Data Center. In *International Conference on Distributed Computing Systems*, pages 59–59, 2007.

[133] In Jae Myung. Tutorial on maximum likelihood estimation. volume 47, pages 90–100. Elsevier, 2003.

[134] MathWorks. Gpfit: Generalized pareto parameter estimates. `http://www.mathworks.com/help/stats/gpfit.html`. Online accessed 10-March-2014.

[135] SciPy Project Team. Parameter estimates by python library scipy. `http://docs.scipy.org/doc/scipy/reference/index.html`. Online accessed 10-March-2014.

[136] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–167, 1998.

[137] Wolfram MathWorld. Bernoulli distribution. http://mathworld.wolfram.com/BernoulliDistribution.html. Online accessed 10-March-2014.

[138] Ying-Wei Kuo, Shou-Hsuan Stephen Huang, and Christopher Hill. Detect multi-hop stepping-stone pairs with clock skew. In *Information Assurance and Security*, pages 74–79, 2010.

[139] Wei Ding and Stephen Huang. Detecting stepping-stones under the influence of packet jittering. In *the 9th Intl. Conference on Information Assurance and Security (IAS)*, pages 31–36, 2013.

[140] The Tor Project Group. Tor: anonymity online. https://www.torproject.org/. Online accessed 10-March-2014.