electronics

MDPI

Article

Explore Utilizing Network Traffic Distribution to Detect Stepping-Stone Intrusion

Jianhua Yang * and Lixin Wang

TSYS School of Computer Science, Columbus State University, Columbus, GA 31907, USA; wang_lixin@columbusstate.edu

* Correspondence: yang_jianhua@columbusstate.edu; Tel.: +1-706-507-8180

Abstract: In the past three decades, stepping-stone intrusion has become a professional and primary way used by intruders to launch their attacks since they can be protected behind a long TCP connection chain. Many different algorithms have been proposed to detect stepping-stone intrusion since 1995. But most algorithms cannot resist intruders' session manipulation. In this paper, we propose a novel approach using the distribution of round-trip time (RTT) of network traffic to detect stepping-stone intrusion. This approach can resist intruders' chaff-perturbation since the round-trip time of network packets can fairly be affected by chaffed packets. The ratio between the standard deviation of the RTTs between Send and Echo packets and the standard deviation of the RTTs between Send and Ack packets can be used to predict if a stepping-stone intrusion exists. The closer to 0 the ratio, the more suspicious a stepping-stone intrusion.

Keywords: stepping-stone intrusion; downstream detection; round-trip time; standard deviation of RTT; network traffic distribution

1. Introduction

Most professional intruders utilize stepping stones [1] to launch their attacks. One significant benefit that intruders can obtain by doing so is that they can protect themselves behind a long connection chain. As we know, a computer network communication using TCP protocol can expose the source IP address to the destination. However, a network communication connection chain containing multiple connections with each utilizing TCP protocol cannot expose the source IP of the first host of the connection chain to the last destination host. This indicates that, at the end of a TCP connection chain, it is infeasible to know where the network packets come from originally. Therefore, by utilizing a long interactive TCP connection chain, intruders can be well protected.

Capturing stepping-stone intruders is non-trivial due to some inherent design defects of TCP protocol. However, detecting such kind of intrusion is possible. Since the 1990s, many approaches have been proposed and developed to detect stepping-stone intrusion (SSI). The SSI detection techniques can be divided into two different categories: host-based and network-based detection. The host-based SSI detection technique is to check if there exists a relayed incoming and outgoing connection pair. The network-based SSI detection technique determines an intrusion using the length of a connection chain. The longer a connection is employed, the higher the probability there exists a stepping-stone intrusion. Estimating the length of a connection chain becomes key for network-based SSI detection.

Some well-known host-based SSI detection approaches include Packet Content Thumbprint (C-Thumbprint), Packet Time Thumbprint (T-Thumbprint), packet counts, Random-Walk, TCP Session Watermark, and cross-over packets. C-Thumbprint was proposed to detect stepping-stone intrusion by S.S. Chen and L. T. Heberlein [2] in 1995. It is the first approach developed for SSI detection. Its basic idea is to compare the packet content of the incoming connection of a host with its outgoing connection to see if there exists a

check for updates

Citation: Yang, J.; Wang, L. Explore Utilizing Network Traffic Distribution to Detect Stepping-Stone Intrusion. *Electronics* **2024**, *13*, 3258. https://doi.org/10.3390/electronics13163258

Academic Editor: Hung-Yu Chien

Received: 28 June 2024
Revised: 1 August 2024
Accepted: 14 August 2024
Published: 16 August 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

relayed pair. For efficiency reasons, instead of comparing the contents of packets directly, C-Thumbprint uses a hash function to compute a fixed-size thumbprint from the packets captured in a session. That is why this approach is normally called content thumbprint. The biggest disadvantage of C-Thumbprint is that it cannot apply to an encrypted TCP session. Since 2000, more and more hackers used encrypted tools, such as OpenSSH, to make a long connection chain to launch their attacks.

In 2000, some computing scientists, such as Y. Zhang, V. Paxson, K. Yoda, and H. Etoh, proposed packet-content-independent approaches to detect stepping-stone intrusion and made significant progress. Y. Zhang et al. [1] developed an algorithm to use the time period for which a session has packets captured, called ON-time, and one for which a session has no packets captured, called OFF-time, to detect SSI. After monitoring a TCP session for a certain period of time, we would obtain two ON-OFF time period data sequences. One is from the incoming connection of a host, and another one is from its outgoing connection. The ON-OFF time sequence is called time thumbprint, denoted as T-Thumbprint. Simply comparing the two T-Thumbprints can help us determine if a host is used as a stepping stone or not. T-Thumbprint is independent of packet contents. Therefore, this method can apply to encrypted TCP sessions. In the same year, K. Yoda and H. Etoh proposed to employ session deviation to detect stepping-stone intrusion [3]. Session deviation can be computed based on the header information of the packets captured from a TCP session other than packet contents. This approach can also apply to an encrypted TCP session.

Between 2002 and 2007, the approaches of comparing the amount of packets from the incoming and outgoing connections of a host were proposed to detect stepping-stone intrusion. They are called packet count methods. Two typical ones in this category are the algorithm proposed by A. Blum et al. in 2004 [4], and another one developed by T. He and L. Tong in 2007 [5]. X. Wang proposed watermark approaches to detect stepping-stone intrusion from 2001 to 2004 [6–8]. The basic idea of the watermark approaches is to inject a small piece of a watermark into the incoming connection of a host and check its outgoing connections to see if the injected watermark can be detected or not.

If a host is used as a stepping stone, the amount of packets monitored from one incoming connection of the host is supposed to be close to the amount of packets from its relayed outgoing connection. In an ideal scenario, the two numbers should be equal. In a non-ideal case, the difference between the two numbers should be bounded. This is the basic idea employed in Random-Walk approach to detect stepping-stone intrusion. Using cross-over packets to detect stepping-stone intrusion was proposed in 2016 by S. H. Huang et al. [9]. A cross-over packet is defined as an Echo packet that meets the second Send packet before the Echo packet of the first Send packet comes back to the sender. The amounts of the cross-over packet from the two relayed sessions are close. In other words, we can monitor the amount of cross-over packets from the incoming connection of a host, as well as its outgoing connection, to detect stepping-stone intrusion.

All of the above host-based SSI detection approaches suffer from two serious issues. One is that they may bring high false-positive errors since a stepping-stone host may not be used for intrusion. Another one is that they could not resist intruders' evasion, such as chaff-perturbation and/or time-jittering manipulation. Network-based SSI detection approaches have been proposed since 2002 to mitigate the above two issues coming with the host-based SSI detection approaches. In the following section, we will briefly summarize some typical network-based SSI detection algorithms.

The representatives of the network-based SSI detection technique involve Yung's approach, Step-function, the cross-over packet approach, and the clustering–partitioning algorithm. The significant benefit of employing the length of a connection chain to detect SSI is that it can reduce false-positive errors. The reason behind this is that some legal applications may use one or two hosts as a stepping stone, but there are few applications using three or more than three hosts as stepping stones. If we can estimate the length of a connection chain, we can, most likely, determine if a host is used for intrusion. The longer a connection chain, the higher the probability the chain is used for stepping-stone intrusion.

To the best of our knowledge, K. H. Yung [10] is the first researcher proposing an algorithm to detect stepping-stone intrusion by estimating the downstream length of a connection chain. If a host is used as a stepping stone, every Send packet along the connection chain would be first acknowledged by its adjacent host and then echoed by the end host of the connection chain. The time difference between a Send packet and its acknowledgment packet would reflect the length between the sender host and its downstream adjacent host along the connection chain. This can be used to measure the length of one connection. We denote it as Ts-a. The time difference between a Send packet and its echoed packet from the end of the connection chain can represent the length of the whole connection chain including multiple connections. We denote this as Ts-e. The ratio of Ts-a and Ts-e can reflect the length of the downstream part of the whole connection chain. The closer the ratio is to zero, the longer the connection chain, and the higher the probability the chain is used as a stepping-stone intrusion. However, K. H. Yung's method failed to match Send with Echo as well as the acknowledgment packet. This makes the method come with high false-negative errors.

Step-function [11] and clustering–partitioning [12] approaches were proposed by J. Yang et al. in 2004 and 2007 [11,12], respectively. Both methods can estimate the length of a connection chain precisely and bring down the false-negative detection error rate. The step-function approach can only apply to a local area network. The clustering–partitioning algorithm can apply to the Internet to detect stepping-stone intrusion, but it needs to collect the network packets from the beginning to the end of establishing a connection chain. A cross-over packet is used to detect stepping-stone intrusion [13]. This is based on an observation that the more the amount of cross-over packets, the longer the downstream connection chain. Unfortunately, cross-over packets can be easily manipulated by intruders' chaff-perturbation and time-jittering evasion.

In this paper, we explore a novel approach to detect stepping-stone intrusion via the distribution of computer network traffic. The advantages of this approach are that (1) it can bring down false-positive detection errors since it belongs to a network-based detection method; (2) it does not need to collect packets from the beginning to the end of establishing a connection chain; (3) it can resist intruders' chaff-perturbation manipulation; and (4) it receives a high detection rate.

As we mentioned before, a Send packet from a sender host can be acknowledged (we call it an Ack packet) by its adjacent host and be echoed (we call it an Echo packet) by the end host of a connection chain. If we can match Send packets with its Ack packets, as well as Echo packets correctly, we would obtain a bunch of round-trip times (RTTs) between Send and Ack (we denote this as RTTs-a), as well as Send and Echo (denoted as RTTs-e). The distribution of RTTs-a and RTTs-e can help us estimate the length of a downstream connection chain. This is the basic idea of using the distribution of RTTs of computer network traffic to detect stepping-stone intrusion.

We arranged the rest of the paper as the following. Section 2 describes the preliminaries needed in this paper. Section 3 presents a significant result on network traffic distribution. Section 4 introduces the stepping-stone detection algorithm using network traffic distribution. Section 5 shows our experimental justification and its analysis. We conclude the whole paper and discuss future work in Section 6.

2. Preliminaries

2.1. Modeling Network Traffic

The most popular way that most attackers launch their stepping-stone intrusion attack is to establish a long connection chain using a tool, such as OpenSSH. In the process of making a connection chain, users need to send a request to the destination host and receive a response packet of the request. Each request is acknowledged by its directly connected adjacent neighbor host and then echoed only by the end host of a connection chain. For example, if a user makes a connection chain from Host A to Host B and then connects to

Host C, which is the end of the chain A→B→C, any request packet sent from Host A is acknowledged by Host B first and then echoed by Host C.

The packets sent from a host with an SSH server (port 22) as its destination are called Send packets. The packets used to acknowledge Send packets are called Ack packets. The response packets used to echo Send packets are called Echo packets. A Send packet must be sent from a user's port, but its destination port must be 22, the source IP is the user's host IP, and its TCP flag contains "P". An Echo packet's source port must be 22, its destination IP must be the user host IP address, and its TCP flag must contain "P". An Ack packet is a packet with source port 22, the user's host IP address as its destination IP, and TCP flag "A" only. In this paper, we use Send, Echo, and Ack packets to model the network communication traffic in an SSH connection chain.

2.2. Packet Match and RTT

Each Send packet incurs a response packet, which is called a corresponding Echo packet. They are called a matched packet pair. A Send packet can also match with its corresponding Ack packet. They are also called a matched packet pair. One Send packet can only be acknowledged by one Ack packet but may trigger multiple Echo packets. The case of one Send and one Echo matched pair is trivial to understand. However, if one Send packet triggers multiple Echo packets, it is not trivial to match them. We will use the MMD data mining packet-matching algorithm [14] developed in 2021 to match Sends and Echoes.

Each matched pair can be used to calculate a round-trip-time (RTT). The RTT from a matched pair between a Send and its Ack can reflect the length of a connection between a sender and its downstream adjacent host along the connection chain. The RTT from a matched packet pair between a Send and its Echo can present the length of the downstream connection chain from a sender to the end host of the chain.

2.3. RTT Distribution

Network traffic can be modeled as Send, Echo, and Ack packets. RTTs can be obtained from matched Send and Echo packet pairs and represent the length of a connection chain. Different matched Send and Echo pairs may result in different RTTs. This is because the network delay introduced by different routers along the same connection chain may vary. The RTTs from the same connection chain may fluctuate but are bounded. The variation of RTTs follows the Poisson distribution [12].

3. Significant Results on Network Traffic Distribution

The RTT from a matched Send and Echo packet pair is composed of the delay introduced by each connection in a connection chain. Obviously, the more connections a connection chain contains, the larger the RTT value. In each connection, the network delay comes from the following delays from each router along the connection plus propagation delay: queue delay, processing delay, and transmission delay. Among the four delays from a TCP connection, transmission delay dominates the whole connection chain delay. It is trivial to know that a longer connection chain would bring a larger RTT. From our experimental tests, we found that the longer the connection chain, the more inconsistent the values of RTTs. Since the variation in RTTs from a connection chain follows the Poisson distribution, we use the standard deviation of RTTs to represent its inconsistency. Our experimental results show that the longer the connection chain, the higher the standard deviation of the RTTs from the connection chain. In the following, we introduce the experimental setup and the results.

In order to justify the relation between the standard deviation of the RTTs of network traffic and the length of a connection chain, we designed the following network connection in the JEMES Lab of the TSYS School of Computer Science, Columbus State University, GA. We selected five computer hosts, named CCT30, CCT29, CCT28, CCT27, and CCT26. All five hosts were connected locally to be part of a LAN. They all ran Linux systems and had SSH servers installed. We established an SSH connection from CCT30 to CCT26 via CCT29,

CCT28, and CCT27. We operated CCT26 from CCT30. This SSH connection chain has four connections: CCT30→CCT29, CCT29→CCT28, CCT28→CCT27, and CCT27→CCT26. While operating CCT 26 from CCT30, we collected the outgoing network traffic, including Send and Echo packets from CCT30, CCT29, CCT28, and CCT27 using the following tcpdump command. We assumed to collect the outgoing network traffic at the host CCT30 from the connection CCT30→CCT29.

sudo tcpdump -nn -tt 'tcp port 22 && (host IP-CCT30 && host IP-CCT29) and (tcp[flags] & tcp-psh == tcp-psh)' > CCT30_TestX.txt, where X represents the number of the test.

The above command can collect Send and Echo packets from the outgoing connection of the host CCT30 and store the collected packets in the file CCT30_Test1.txt if this is the first test. A Send packet takes port 22 as the destination port and uses the IP address of CCT30 as its source IP address and CCT29 as its destination address. An Echo packet takes port 22 as its source port and uses CCT30 as its destination host and CCT29 as its source host. Both Send and Echo packets contain Push flags as Flags [P.]. We repeated the test 10 times. We obtained ten collected packet files, CCT30_TestX.txt, where X = 1, 2, 3, . . ., 10. Similarly we obtained packet files from CCT29, CCT28, and CCT27 as CCT29_TextX.txt, CCT28_TestX.txt, and CCT27_TestX.txt, respectively. The following shows the format of one collected packet.

1643041500.815242 IP 168.27.2.106.40946 > 168.27.2.103.22: Flags [P.], seq 2409120754: 2409120790, ack 3129783136, win 501, options [nop,nop,TS val 1613436092 ecr 2848772024], length 36.

In this collected sample packet, the first field is the timestamp to collect this packet, the second field tells us this is an IP packet, and the following fields tell us the source IP address (168.27.2.106), the destination IP address (168.27.2.103), the source port (40496), the destination port (22), and much other packet header information. Obviously, based on the Send and Echo definition discussed in Section 2, this is a Send packet. The packet flag field contains the flag information showing that the packet carries data and also acts as an acknowledgment. The following fields show its sequence and acknowledgment number. The last field displays the length of the packet. We apply the method discussed in Section 5 to process the collected packets in the files CCTY_TestX.txt, where Y = 30, 29, 28, and 27, X = 1 to 10. We obtained the standard deviation of RTTs from the hosts CCT30, CCT29, CCT28, and CCT27. The results can be found in Table 1.

Table 1. The standard deviation of 10 Tests.

Test \ Host	CCT30	CCT29	CCT28	CCT27
Test 1	338.77	256.48	66.43	31.64
Test 2	231.90	116.50	73.22	57.01
Test 3	121.37	117.73	75.49	45.93
Test 4	234.92	179.58	170.96	60.75
Test 5	259.11	242.48	124.07	82.21
Test 6	234.34	187.01	150.66	70.80
Test 7	245.73	203.86	157.59	70.32
Test 8	279.06	206.16	118.82	97.22
Test 9	265.52	210.68	128.41	88.89
Test 10	147.69	131.82	91.97	97.20

In Table 1, each column shows the RTT standard deviation for different hosts, and each row represents the number of tests. From the column of CCT30, we obtain the

RTT standard deviation of the outgoing connection chain of CCT30 including four connections: CCT30→CCT29, CCT29→CCT28, CCT28→CCT27, and CCT27→CCT26. The column CCT29 represents the RTT standard deviation of the outgoing connection chain of CCT29. This connection chain contains three connections, including CCT29→CCT28, CCT28→CCT27, and CCT27→CCT26. Similarly, the CCT28 column and CCT27 column represent the RTT standard deviation for the outgoing connection chain of CCT28 and CCT27, respectively. The length of the CCT28 outgoing connection chain is 2, but for CCT27, the length is 1.

From Table 1, it is trivial to observe that, along with the decrease in the length of the connection chain, the standard deviation of the RTTs obtained in different hosts also becomes smaller. If we examine the row of Test 1, we see that in CCT30, where the length of the connection chain is 4, the standard deviation of the RTTs is 338.77, but for CCT29, CCT28, and CCT27, the standard deviations of the RTTs are 256.48, 66.43, and 31.64, respectively, with corresponding 3, 2, and 1 connections. A similar conclusion can be drawn from each test from Test 2 to Test 10. Only one exception happened in CCT28 from Test 10 as shown in the table.

The experimental result shows that a longer connection chain can introduce a larger standard deviation of the RTTs of the network traffic. A longer connection chain has more routers and hosts involved. Each router or host can introduce network delays, including queue delay, transmission delay, and processing delay. The three delays are indeterministic. They introduce inconsistency of RTTs. The more routers and hosts in a connection chain, the greater the inconsistency of the RTTs of the network traffic. Therefore, the standard deviation of the RTTs of network traffic can be used to reflect the length of a connection chain. It is hard to infer the number of connections in a connection chain. It is impossible to tell if a connection chain is long or short just depending on the standard deviation of the RTTs of network traffic. A measuring unit is needed to determine if a connection chain is long or short. Here is a good metaphor for this: if we have two ropes, one's length is 180, and the other one's is 10; it is hard to tell which rope is longer because this depends on the length measuring unit. This will be discussed in Section 4.

4. Detection Algorithm

4.1. Modeling the Problem

As we have discussed, most attackers tend to launch their attacks on a victim via a long connection chain. In this way, attackers can be well protected. The goal of this research is to estimate the length of a connection chain using the standard deviation of RTTs. The security problem can be modeled as the following.

We assume Bob, an attacker, launches attacks on Alice remotely via an OpenSSH connection chain spanning hosts Host1, Host2, ..., and HostN. Our detection program can run at any one of Host1 to HostN, such as HostK ($1 \leq K \leq N$), to detect if attacks exist. HostK, where a detection program resides, is called a sensor. We collect the network packets from HostK and use them to determine if there exists a long connection chain. Figure 1 shows such a security model. In this model, Bob acts as an attacker, Alice is the victim, and the hosts Host1, ..., HostN are assumed as the compromised hosts: stepping stones.

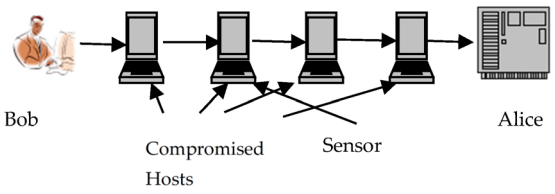

Figure 1. A security model with attacker, victim, and sensor.

4.2. TheAlgorithm

In the security model shown in Figure 1, the algorithm detects if the chain from the sensor to Alice is a long connection chain. In order to reach this goal, network traffic is collected at the outgoing connection of the sensor host. We primarily focus on collecting Send, Echo, and Ack packets while ignoring others. Unlike other algorithms proposed before, this algorithm does not need to collect packets from the beginning of establishing a connection chain to its completion. It only needs to collect network traffic for a certain period of time. The detection algorithm is shown as the following.

Step 1: Collect packets, including Send, Echo, and Ack, from the outgoing connection of a sensor host for a certain period of time.

Step 2: Match Send with Echo packets using the MMD data mining algorithm [14] and compute the RTTs from the matched Send and Echo pairs. Denote them as RTT-E set.

Step 3: Match Send with Ack packets using the same packet-matching algorithm as in Step 2. Compute the RTTs from the matched Send and Ack pairs. Denote them as RTT-A set.

Step 4: Filter out the outliers of RTT-E and RTT-A sets using the "Z-score of two" outlier elimination method [15].

Step 5: Compute the standard deviation of RTT-E and RTT-A sets. And denote them as StdE and StdA, respectively.

Step 6: Calculate the ratio α between StdA and StdE. The ratio α is between 0 and 1.

Step 7: Make a decision. The closer to 0 the ratio α, the higher the probability a long connection chain is used, and the more suspicious the user acts as a stepping-stone attacker.

In the above stepping-stone detection algorithm, we employ the fact that the longer a connection chain, the greater the inconsistency of the RTTs regardless of whether the RTTs are between Send and Echo or the RTTs are between Send and Ack. In this algorithm, the RTT-A is used as a measure bar to decide the length of the whole connection between a sensor and the destination host (Alice in Figure 1). If the measure bar is too small, a short connection chain may be measured as a longer one. The detection algorithm may introduce false-positive errors. Correspondingly, a too-large measure bar may make this algorithm introduce false-negative errors because a long connection chain may be measured as a shorter one, thus missing an intrusion.

4.3. Efficiency Analysis of the Detection Algorithm

We will analyze the efficiency of the detection algorithm using Big O notation. In the above algorithm, we assume n Send, n Echo, and n Ack packets are captured. Please note that, here, it is normally impossible to have the same number of Send, Echo, and Ack packets captured. For simplicity of the efficiency analysis, we assume they are the same. In Step 1, the algorithm has O(n) efficiency. In Step 2, its efficiency depends on the packet-matching algorithm in [14]. It is O(wn), where w is the size of the sliding window with its typical value of 7. So, n is a large number, but w is a small one. In Step 3, its efficiency is O(wn). In Step 4, its efficiency is O(wn + wn). In Step 5, its efficiency is O(n). In Step 6, its efficiency is O(1). The six steps in the detection algorithm are sequential ones. Therefore, the total efficiency of the algorithm is O(n) + O(wn) + O(wn) + O(2wn) + O(wn) + O(1) $\approx$ O(wn). If w is a small number, its efficiency is O(n)—linear. The state-of-the-art stepping-stone detection algorithm [12] has O(n^2) quadratic efficiency, which is worse than the proposed one.

5. Experimental Results and Analysis

An experiment was designed to assess the performance of the detection algorithm. We made a long connection chain using OpenSSH spanning eight hosts, including an attacker host, compromised hosts, and a victim host. Network traffic was collected from each compromised host. We followed the algorithm to compute α for each host. Then, we examined if α becomes larger as the sensor is closer to the victim. This is because a larger

α represents a shorter connection chain. The closer a sensor is to a victim, the shorter the connection chain between the sensor and the victim.

5.1. Experimental Setup

In this experiment, we made a connection from one computer CCT30 in the CCT lab to seven AWS servers, AWS1, AWS2, AWS3, AWS4, AWS5, AWS6, and AWS7, and connected back to CCT31 in the same lab. We built an eight-connection chain containing nine hosts with CCT30 as an attacker and CCT31 as a victim. Each host ran Linux OS with SSH client and server installed. The tcpdump tool was also available in each host. Table 2 shows the AWS server's public and private IP addresses and their corresponding geographic locations.

Table 2. AWS servers' IP and geographic location.

Server Name	OS	Public IP Address	Private IP Address	Geographic Location
AWS 1	Ubuntu	34.239.127.118	172.31.86.245	Ashburn, VA, USA
AWS 2	Ubuntu	52.59.96.142	172.31.29.198	Frankfurt, Germany
AWS 3	Ubuntu	18.133.230.26	172.31.39.118	London, UK
AWS 4	Ubuntu	52.60.79.102	173.31.12.163	Vancouver, BC, Canada
AWS 5	Ubuntu	52.194.232.92	172.31.0.70	Tokyo, Japan
AWS 6	Ubuntu	34.248.116.10	172.31.32.141	Dublin, Ireland
AWS 7	Ubuntu	52.67.153.198	172.31.13.98	San Paulo, Brazil

5.2. Data Collection

In order to assess the detecting and adaptive performance of the proposed detection algorithm, we simulated three different attackers using different scripts to launch attacks via the host CCT30 to the victim CCT31. The scripts used by the three attackers are shown in the following.

The attacker 1 script was as follows:
pwd
whoami
sudo su
ls
cd/etc
ls −a
scp −p shadow attacker username@attacker IP:/home/seed/Documents exit

The attacker 2 script was as follows:
whoami
pwd
cd/home/seed/Documents
ls
nano text_file.txt
//paste a large text and save it
ls
cat hello.txt
exit

The attacker 3 script was as follows:
whoami

```
pwd
cd/home/seed/Documents
ls
nano hello.txt
//enter a few sentences and save the text file
ls
cat hello.txt
exit
```

Attacker 1 first checks the name of its current folder; then checks who they are; changes to admin privilege; checks the details of the current folder; changes its folder to etc, which contains some sensitive files; lists the details in the etc folder; and finally copies a file to the etc folder. What Attacker 2 does includes (1) checking who they are; (2) checking where they are; (3) changing its current folder to home/seed/Documents; (4) listing the files and directories in the current folder; (5) creating a file, text_file.txt, which is a large file; (6) checking if the file is created; (7) displaying the content of hello.txt; and (8) exiting. Attacker 3 does similar things as Attacker 2 does except for creating hello.txt, which is a small file.

Each attacker repeated their corresponding script ten times to make network traffic all the way down from CCT30 to CCT31 via the AWS servers: AWS1, AWS2, . . ., AWS7. We ran the following tcpdump command on each host except CCT31 to collect Send, Echo, and Ack packets from its outgoing connection.

sudo tcpdump -nn -tt "(dst port 22 && src xxx.xxx.xxx.xxx) || (dst xxx.xxx.xxx.xx && src port 22)" >HostName_Out_AttackX_TestY.txt

We use the server AWS1 as an example to demonstrate how to sniff network packets from a host's outgoing connection. As we discussed above, the outgoing connection of AWS1 is the connection from AWS1 to AWS2. Therefore, in the above tcpdump command, in order to collect Send and Echo packets, we apply the following rule: the source IP is the IP address of AWS1 with destination port 22, or the destination IP is the IP address of AWS2 with source port 22. The filter "*dst port 22 && src xxx.xxx.xxx.xxx*" is to capture the Send packets from AWS1 to AWS2, but the filter "*dst xxx.xxx.xxx.xx && src port 22*" is to capture the Echo and Ack packets from AWS2. In the packet stored file name, the field "*AttackX*" represents Attacker1, Attacker2, or Attacker3, and the field "*TestY*" represents the number of tests; here, *Y* is an integer between 1 and 10 included because each attacking script code was repeated 10 times. For example, if Attacker1 runs its script at AWS1 a third time to launch attacks on the victim CCT31 via the compromised servers, the captured packets are stored in the file "*AWS1_Out_Attack1_Test3.txt*". In each host, after completing all three attacks by repeating each attack 10 times, we obtained 30 files, which contain all of the sniffed Send, Echo, and Ack packets. In each file, every sniffed packet is stored in one row. The following shows the format of one captured packet.

1643041500.815242 IP 168.27.2.106.40946 > 168.27.2.103.22: Flags [P.], seq 2409120754: 2409120790, ack 3129783136, win 501, options [nop,nop,TS val 1613436092 ecr 2848772024], length 36.

From the above captured packet, we can obtain its timestamp (field 1), source IP and source port number (field 2), destination IP and destination port number (field 3), TCP flags (field 4), sequence number (field 5), acknowledgment number (field 6), and some other trivial fields. The last field is the length of the packet, which is 36 bytes in this example. From the captured packet information, we know this is a Send packet sent from IP 168.27.2.106 and port 40946 to IP 168.27.2.103 with port 22. Obviously, from the port number, IP address, and TCP flags, it is trivial to determine the type of a captured packet: Send, Echo, or Ack.

5.3. Data Processing

In order to compute the RTTs of the captured network traffic and calculate their standard deviations, we need to go through several steps to process the collected data in each host as the following.

Step 1: Obtain Send, Echo, and Ack packets.
Step 2: Match Send packets with Echo packets to obtain RTT-E.
Step 3: Match Send packets with Ack packets to obtain RTT-A.
Step 4: Filter out the outliers of RTT-E and RTT-A.
Step 5: Calculate the stand deviation StdE from RTT-E, as well as StdA from RTT-A.
Step 6: Compute the ratio α between StdA and StdE.

In Step 1, a Python code was made to obtain Send, Echo, and Ack packets from a collected packet data file and store them in Send.txt, Echo.txt, and Ack.txt, respectively. In Step 2, we matched Send packets with Echo packets and computed the RTTs for each matched Send and Echo pair using the data mining approach proposed in 2021 [14]. The matched RTTs are stored in RTT-E.txt, and similarly, from Send and Ack packets, we could obtain RTT-A as shown in Step 3. In Step 4, a Python code was made to filter out the outliers in RTT-E.txt, as well as in RTT-A.txt. The algorithm used to filter out the outliers is the "Z-score of two" outlier elimination method as we mentioned in Section 4. In Step 5, we calculated the standard deviation of the matched Send and Echo packets. The longer a connection chain, the more inconsistent the RTTs between the matched Send and Echo pair, and the higher the standard deviation of the RTTs. In order to know how high the StdE is, we calculated StdA. We used StdA as a measure bar to gauge StdE. This is what was carried out in Step 6.

5.4. Data Analysis

The three tables, Tables 3–5, show the StdE in 10 tests for Attacker 1, Attacker 2, and Attacker 3, respectively.

Table 3. StdE in 10 tests for Attacker 1.

Test \ Server	CCT30	AWS1	AWS2	AWS3	AWS5	AWS6	AWS7
Test1	343.10	347.35	330.75	263.03	217.28	170.88	139.05
Test2	350.54	327.97	304.37	220.81	166.87	144.10	203.28
Test3	432.87	424.06	435.74	307.32	201.23	138.91	77.50
Test4	1020.15	1016.38	924.57	219.66	190.55	312.86	192.42
Test5	314.23	310.99	623.03	610.85	182.78	135.35	528.85
Test6	349.54	339.91	316.20	610.85	210.93	199.64	76.38
Test7	301.76	288.33	248.48	220.65	235.03	118.19	66.66
Test8	446.64	440.74	428.90	376.91	187.81	200.57	112.19
Test9	655.37	321.59	286.71	244.71	179.23	139.43	107.58
Test10	518.01	558.01	430.40	400.74	321.31	292.32	133.12

Table 4. StdE in 10 tests for Attacker 2.

Test \ Server	CCT30	AWS1	AWS2	AWS3	AWS5	AWS6	AWS7
Test1	801.01	768.4	716.72	642.47	1028.85	733.14	428.51
Test2	577.39	524.70	392.24	374.14	293.32	216.13	173.52
Test3	538.87	519.72	414.22	384.52	293.32	263.96	205.64
Test4	723.35	717.79	707.63	663.26	526.86	244.17	190.76
Test5	989.68	979.14	991.69	870.53	737.54	731.51	196.03
Test6	433.98	396.02	417.13	296.07	167.63	101.10	77.13
Test7	397.01	479.22	342.78	280.51	240.16	219.49	156.00
Test8	392.23	425.46	397.98	337.12	279.43	222.30	179.57
Test9	535.65	510.50	492.53	413.09	301.50	202.80	208.70
Test10	735.40	709.55	640.84	397.09	367.97	343.42	320.16

Table 5. StdE in 10 tests for Attacker 3.

Test \ Server	CCT30	AWS1	AWS2	AWS3	AWS5	AWS6	AWS7
Test1	435.86	439.11	392.46	388.18	284.37	203.63	184.58
Test2	494.5	475.81	428.53	378.65	285.90	189.35	178.54
Test3	482.21	657.31	624.66	594.97	582.40	434.27	387.53
Test4	405.27	420.61	379.63	279.88	193.55	156.07	154.28
Test5	690.59	677.73	644.84	546.37	495.87	450.94	190.36
Test6	656.07	654.95	608.18	294.60	247.91	201.55	167.38
Test7	494.09	483.32	480.33	442.96	373.43	307.27	245.37
Test8	427.77	401.15	385.42	343.66	282.82	163.10	126.54
Test9	752.04	731.87	665.25	583.87	462.62	288.17	199.72
Test10	702.49	663.35	669.11	593.35	325.67	339.92	148.93

Tables 6–8 show the StdA in 10 tests for Attacker 1, Attacker 2, and Attacker 3, respectively.

Table 6. StdA in 10 tests for Attacker 1.

Test \ Server	CCT30	AWS1	AWS2	AWS3	AWS5	AWS6	AWS7
Test1	51.2	65.1	71.7	93.4	105.3	172.2	165.9
Test2	66	63.8	91.8	101	94.5	95	201.2
Test3	58	72	110.23	104.8	103.5	106.5	68.3
Test4	169.7	186.1	209.9	177.7	226.7	232.1	189.3

Table 6. *Cont.*

Server / Test	CCT30	AWS1	AWS2	AWS3	AWS5	AWS6	AWS7
Test5	55.4	66.7	157.1	98.1	107.2	93.7	87.2
Test6	57.2	71.1	92.2	91.7	115.5	125.2	71.3
Test7	83.1	87.9	103.7	97.8	146.1	85.7	62.8
Test8	52.7	67.8	92.6	92.8	81.6	63.9	97.9
Test9	88.7	58.5	69.2	90.6	81.6	115	102.3
Test10	80	101.5	88.3	112.2	86	99.5	92.6

Table 7. StdA in 10 tests for Attacker 2.

Server / Test	CCT30	AWS1	AWS2	AWS3	AWS5	AWS6	AWS7
Test1	127.84	142.3	162.7	152.3	160.5	178.9	249.8
Test2	63.6	88	84.2	104.8	135.4	153.9	174
Test3	82.8	95.5	89.2	99.5	87	97.5	167.7
Test4	118.3	125.6	148.2	183.2	182.8	120.3	171.7
Test5	163.5	176	207.1	218.8	207.9	252.7	187.1
Test6	67.9	59.5	80.8	75.8	57.6	49.7	94.1
Test7	95.2	92.9	73.1	72.1	84.2	113.5	145.8
Test8	75	73.9	73.7	81.7	83.7	117.1	177.5
Test9	70.7	87.3	89.3	84.4	91.1	114.9	205.1
Test10	91.9	106.2	121.8	89.5	142.1	204	312.2

Table 8. StdA in 10 tests for Attacker 3.

Server / Test	CCT30	AWS1	AWS2	AWS3	AWS5	AWS6	AWS7
Test1	58	63.8	79.6	89.4	79.1	119.7	151.4
Test2	51.5	74.6	83.9	93.2	92	96.2	164.9
Test3	103	125.9	136.8	152.8	224.4	231.3	347.8
Test4	51.1	56.1	60.5	49.7	48.9	51.9	86
Test5	89.1	94.6	90.3	82.7	74.8	153.4	98.5
Test6	77.4	84.5	91.9	59.5	64.4	107.3	160.8
Test7	75.5	91.7	101.6	117.9	126.1	173.7	225.2
Test8	45.5	57	73.4	66.8	53.6	81.2	101.8
Test9	89.7	98.9	122.8	113.2	98.9	90.2	151.4
Test10	79.8	90.8	86.68	123.6	124.5	167.6	142.6

Tables 9–11 show the ratio α between StdA and StdE in 10 tests for Attacker 1, Attacker 2, and Attacker 3, respectively.

Table 9. Ratio between StdA and StdE in 10 tests for Attacker 1.

Test \ Server	CCT30	AWS1	AWS2	AWS3	AWS5	AWS6	AWS7
Test1	0.15	0.19	0.22	0.36	0.48	1.01	1.19
Test2	0.19	0.19	0.30	0.46	0.57	0.66	0.99
Test3	0.13	0.17	0.25	0.34	0.51	0.77	0.88
Test4	0.17	0.18	0.23	0.81	1.19	0.74	0.98
Test5	0.18	0.21	0.49	0.47	0.59	0.69	0.68
Test6	0.16	0.21	0.29	0.29	0.55	0.63	0.93
Test7	0.28	0.30	0.42	0.44	0.62	0.73	0.94
Test8	0.12	0.15	0.22	0.25	0.43	0.32	0.87
Test9	0.14	0.18	0.24	0.37	0.46	0.82	0.95
Test10	0.15	0.18	0.21	0.28	0.27	0.34	0.70

Table 10. Ratio between StdA and StdE in 10 tests for Attacker 2.

Test \ Server	CCT30	AWS1	AWS2	AWS3	AWS5	AWS6	AWS7
Test1	0.16	0.19	0.23	0.24	0.26	0.34	0.58
Test2	0.11	0.17	0.21	0.28	0.46	0.71	1.00
Test3	0.15	0.18	0.22	0.26	0.30	0.37	0.82
Test4	0.16	0.17	0.21	0.28	0.35	0.49	0.90
Test5	0.17	0.18	0.21	0.25	0.28	0.35	0.95
Test6	0.16	0.15	0.19	0.26	0.34	0.49	1.22
Test7	0.16	0.19	0.21	0.26	0.35	0.52	0.93
Test8	0.15	0.17	0.19	0.24	0.30	0.53	0.99
Test9	0.13	0.17	0.18	0.20	0.30	0.57	0.98
Test10	0.12	0.15	0.19	0.23	0.39	0.59	0.98

Table 11. Ratio between StdA and StdE in 10 tests for Attacker 3.

Test \ Server	CCT30	AWS1	AWS2	AWS3	AWS5	AWS6	AWS7
Test1	0.13	0.16	0.20	0.23	0.28	0.59	0.82
Test2	0.10	0.16	0.20	0.25	0.32	0.51	0.92
Test3	0.15	0.19	0.22	0.26	0.39	0.53	0.90
Test4	0.11	0.13	0.16	0.18	0.25	0.33	0.56
Test5	0.13	0.14	0.14	0.15	0.15	0.34	0.52

Table 11. *Cont.*

Test \ Server	CCT30	AWS1	AWS2	AWS3	AWS5	AWS6	AWS7
Test6	0.12	0.13	0.15	0.20	0.26	0.53	0.96
Test7	0.15	0.19	0.21	0.27	0.34	0.57	0.92
Test8	0.11	0.14	0.19	0.19	0.19	0.50	0.80
Test9	0.12	0.14	0.18	0.19	0.21	0.31	0.76
Test10	0.11	0.14	0.13	0.21	0.38	0.49	0.96

The ratio α between RTT-E and RTT-A is supposed to reflect the length of the connection chain from the host where we collect network traffic to the host that is located at the end of the connection chain. Let us examine the ratios we obtained from three attackers, each with 10 tries. The connection chain we made was from CCT30 to CCT31 via AWS servers AWS1, AWS2, . . ., and AWS7. The chain has eight connections. If we collect network traffic at CCT30, the value of α should be close to 0. Similarly, if we collect data at AWS1, AWS2, . . . or AWS7, the value of α should be increased from close to 0 to far away from 0 but still bounded within 1. By checking the ratios in Table 9 for Attacker 1, we found three exceptions with its ratio exceeding 1 (refer to the ratios of Test1-AWS6, Test1-AWS7, and Test4-AWS5). We also found that the ratio decreased when the chain became shorter, which is opposite to our algorithm (refer to the ratios of Test5-AWS3, Test8-AWS6, and Test10-AWS5). In Tables 10 and 11, we did not find any exceptions.

Even though some ratios in Table 9 are not consistent with the number of corresponding connections, we are confident about one point we observed from the ratios in the three tables. The point is, in general, in each row of Tables 9–11, the value of α is increased from CCT30 to AWS7. This indicates that as the length decreases in a connection chain, the corresponding α increases. If we count two neighbor hosts as one pair, such as in Table 9, the ratio pair (0.15→0.19) maps to two neighbor hosts (CCT30→AWS1), and in Table 8, we obtain 60 ratio pairs. The ones in Test5-AWS3, Test8-AWS6, and Test10-AWS5 represent abnormal pairs. From the three tables, Tables 8–10, we altogether obtain 180 ratio pairs with 3 abnormal pairs. It is trivial to know that among the tests from three Attackers, the abnormal rate is 3/180 = 1.67%.

From the above experimental result, we definitely know that the ratio between StdA and StdE represents the length of a connection chain from the sensor to the end host. The closer to 0 the ratio, the longer the connection chain. This ratio is not affected by intruders' manipulation because the RTTs of network traffic are from the MMD packet-matching algorithm [14], which resists intruders' chaff-perturbation and time-jittering evasion.

5.5. Comparison with the State-of-the-Art Detection Algorithm

To the best of our knowledge, the method of the state-of-the-art detection algorithm is to mine network traffic to detect stepping-stone intrusion [12]. The strength of the method [12] is that it can accurately estimate the number of connections in a chain. The weaknesses of the method include (1) it having to capture the session packets from the beginning to the end of establishing a connection chain; (2) it being limited to resisting intruders' session manipulation; and (3) poor detection efficiency ($O(n^2)$), where n is the number of packets captured.

For this proposed approach, its strength includes the fact that (1) it is not necessary to monitor a TCP session from the beginning to the end. Therefore, the approach can reduce the number of packets captured and thus improve its detection efficiency. (2) It can resist intruders' chaff-perturbation to a high capacity. Its obvious weakness is that it cannot accurately estimate the number of connections in a chain. It can only tell if a connection chain is longer or shorter based on the ratio between StdA and StdE.

6. Conclusions and Future Work

In this paper, we explore using the distribution of RTTs of network traffic to detect stepping-stone intrusion. The rationale behind this is that the longer a connection chain, the more inconsistent the RTTs of the network traffic. In other words, the distribution of RTTs from a connection chain can help us to decide if a connection chain is longer or shorter. A longer connection chain indicates a higher possibility of a stepping-stone intrusion attack. The experimental results from three attackers, over 30 tries, show that the ratio between StdA and StdE can be used to detect stepping-stone intrusion. If the ratio is closer to 0, it is highly suspicious that a stepping-stone intrusion attack exists. The first strength of our proposed algorithm is that it can resist intruders' session manipulation, such as chaff-perturbation. The rationale is that the packet-matching algorithm used can filter out the chaffed packets. The second strength of the proposed algorithm to detect stepping-stone intrusion is that it works in O(n)-linear efficiency, which is better than the state-of-the-art detection algorithm. The third one is that, unlike the previous algorithm, it is not necessary to capture the traffic from the beginning to the end of establishing a connection chain. The weakness of the proposed algorithm is very obvious: it can only tell if a connection chain is longer or shorter, not exactly the number of connections in the chain, which may bring false-positive errors.

In the future, our effort will focus on studying the relation between the number of connections in a chain and the ratio between StdE and StdA of network traffic.

Author Contributions: J.Y.: conceptualization, methodology, writing, project administration, data curation, and funding acquisition; L.W.: software, validation, formal analysis, investigation, supervision, and funding acquisition. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Security Agency NCAE-C research grant H98230-20-1-0293 with Columbus State University, Columbus, GA, USA.

Data Availability Statement: Derived data supporting the findings of this study are available from the corresponding author on request. All data underlying the results are available as part of the article, and no additional source data are required. Data supporting this study are included within the article**.** The network traffic packets captured for this project can be found from the shared Drive: https://drive.google.com/drive/folders/1KvTRBPthVGEf5sk1HJiftSI0uThYi1Dp.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Zhang, Y.; Paxson, V. Detecting Stepping-Stones. In Proceedings of the 9th USENIX Security Symposium, Denver, CO, USA, 14–17 August 2000; pp. 67–81.
2. Staniford-Chen, S.; Heberlein, L.T. Holding Intruders Accountable on the Internet. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, USA, 8–10 May 1995; pp. 39–49.
3. Yoda, K.; Etoh, H. Finding Connection Chain for Tracing Intruders. In Proceedings of the 6th European Symposium on Research in Computer Security, Toulouse, France, 4–6 October 2000; Lecture Notes in Computer Science; Volume 1985, pp. 31–42.
4. Blum, A.; Song, D.; Venkataraman, S. Detection of Interactive Stepping-Stones: Algorithms and Confidence Bounds. In Proceedings of the International Symposium on Recent Advance in Intrusion Detection, Sophia Antipolis, France, 15–17 September 2004; pp. 20–35.
5. He, T.; Tong, L. Detecting encrypted stepping-stone connections. *IEEE Trans. Signal Process.* **2007**, *55*, 1612–1623. [CrossRef]
6. Wang, X.; Reeves, D.S.; Wu, S.F.; Yu, J. Sleepy Watermark Tracing: An Active Network-Based Intrusion Response Framework. In Proceedings of the 16th International Conference on Information Security, Paris, France, 11–13 June 2001; pp. 369–384.
7. Wang, X.; Reeves, D.S. Robust Correlation of Encrypted Attack Traffic Through Stepping Stones by Manipulation of Inter-packet Delays. In Proceedings of the 10th ACM Conference on Computer and Communications Security, Washington, DC, USA, 27–30 October 2003; pp. 20–29.
8. Wang, X. The Loop Fallacy and Serialization in Tracing Intrusion Connections through Stepping Stones. In Proceedings of the 2004 ACM Symposium on Applied Computing, Nicosia, Cyprus, 14–17 March 2004; pp. 404–411.
9. Huang, S.S.-H.; Zhang, H.; Phay, M. Detecting Stepping-stone intruders by identifying crossover packets in SSH connections. In Proceedings of the 30th IEEE International Conference on Advanced Information Networking and Applications, Fukuoka, Japan, 23–25 March 2016; IEEE proceedings and Digital Library; pp. 1043–1050.

10. Yung, K.H. Detecting Long Connecting Chains of Interactive Terminal Sessions. In Proceedings of the International Symposium on Recent Advance in Intrusion Detection (RAID), Zurich, Switzerland, 16–18 October 2002; pp. 1–16.
11. Yang, J.; Huang, S.-H.S. A Real-Time Algorithm to Detect Long Connection Chains of Interactive Terminal Sessions. In Proceedings of the 3rd ACM International Conference on Information Security (Infosecu'04), Shanghai, China, 14–16 November 2004; pp. 198–203.
12. Yang, J.; Huang, S. Mining TCP/IP Packets to Detect Stepping-Stone Intrusion. *Comput. Secur.* **2007**, *26*, 479–484. [CrossRef]
13. Yang, J.; Wang, L.; Lee, A.; Wan, P.-J. Stepping-Stone Intrusion Detection via Estimating Numbers of Upstream and Downstream Connections using Packet Crossover. *J. Wirel. Mob. Netw. Ubiquitous Comput. Dependable Appl.* **2022**, *13*, 24–39. [CrossRef]
14. Yang, J.; Wang, L. Applying MMD Data Mining to Match Network Traffic for Stepping-Stone Intrusion Detection. *Sensors* **2021**, *21*, 7464. [CrossRef] [PubMed]
15. Five Outlier Detection Techniques. Available online: https://towardsdatascience.com/5-outlier-detection-methods-that-every-data-enthusiast-must-know-f917bf439210 (accessed on 18 August 2023).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.