Analyzing Distribution of Packet Round-Trip Times using Fast Fourier Transformation

Lixin Wang[1*], Jianhua Yang[2] and Maochang Qin[3]

[1*] Professor, Computer Science, Columbus State University, GA, USA.
wang_lixin@ColumbusState.edu, Orcid: https://orcid.org/ 0000-0003-4965-5510

[2] TSYS School of Computer Science, Columbus State University, GA, USA.
yang_jianhua@columbusstate.edu, Orcid: https://orcid.org/ 0000-0003-2745-8524

[3] TSYS School of Computer Science, Columbus State University, GA, USA.
qin_maochang@ColumbusState.edu, Orcid: https://orcid.org/0009-0000-9742-2178

Received: April 14, 2023; Accepted: June 08, 2023; Published: June 30, 2023

Abstract

Hackers usually send attacking commands through compromised hosts, called stepping-stones, for the purpose of decreasing the chance of being discovered. An effective approach for stepping-stone intrusion detection (SSID) is to estimate the length of a connection chain. This type of detection method is referred to as the network-based SSID (NSSID). All the existing NSSID approaches use the distribution of packet round-trip times (RTTs) to estimate the length of a connection chain. In this paper, we explore a novel approach – Fast Fourier Transformation (FFT) to analyze the distribution of packet RTTs. We first capture network packets from different stepping-stones in a connection chain, identify and match the Send and Echo packets in each stepping-stone. Packet RTTs can be obtained from matched pairs of packets. We then apply the FFT interpolation method to obtain a RTT time function and finally conduct FFT transformation to the RTT function in each stepping-stone host. Finally, we conduct a complete FFT analysis for the distribution of packet RTTs and present the FFT analysis results in this paper.

Keywords: Stepping-Stone Intrusion, Intrusion Detection, Interpolation, Round-Trip Time, Fast Fourier Transformation.

1 Introduction

Most attackers, especially professionals, normally intrude a computing system indirectly, other than directly. The primary reason is that direct intrusion would expose their IPs to most detection systems. The attacking is easy to be detected, prevented, and even captured. In order to avoid to be detected and captured, indirectly invasion is widely used most professional attackers. With indirect attacking, hackers send their attacking commands via several compromised computing hosts called stepping-stones (Zhang, Y., 2000). Such an attack is referred to as a stepping-stone intrusion (SSI). An algorithm proposed to detect SSI is called stepping-stone intrusion detection (SSID).

Since 1995, there were tons of approaches developed for SSID. These methods proposed to detect SSI can be classified into two categories. One is to check the incoming and outgoing network traffic of the same host to determine if there are any relayed connections. This type of approach is called host-

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), volume: 14, number: 2 (June), pp. 137-149 DOI: *10.58346/JOWUA.2023.I2.011*

*Corresponding author: Professor, Computer Science, Columbus State University, GA, USA.

based stepping-stone intrusion detection (HSSID). The other one is to estimate the length of a connection chain from a host to the corresponding victim host. Here the host where we run our SSID program is called a sensor. If the length is three or more, most likely, the session is launched by an attacker. This type of detection approach is called network-based SSID (NSSID).

The first HSSID approach was proposed by S. Chen and L. T. Herblein (Staniford-Chen, S., 1995). In their research, a thumbprint method was proposed to identify a TCP connection. A relayed connection pair can be distinguished by comparing the thumbprints of an incoming and outgoing connection, respectively. A thumbprint in (Staniford-Chen, S., 1995) was defined as the summary of the contents of the packets captured in a connection for certain time period, for example, a period of 4 minutes. Relayed connection pair should have very similar thumbprints. Unfortunately, this method cannot be applicable to encrypted TCP sessions since the contents of the packets captured from such sessions are not readable. Another type of time thumbprint used to address the encryption issue was proposed by Y. Zhang and V. Paxson (Zhang, Y., 2000) in 2000. This time thumbprint is called time-based thumbprint (TTP), and the former one proposed in (Staniford-Chen, S., 1995) is called content-based thumbprint (CTP). A TTP in (Zhang, Y., 2000) is looking for the "ON" and "OFF" time gap in a sequence of packets captured from a TCP session. The "ON" time represents the time gap in the second unit there are packets active in the session. Similarly, the "OFF" time represents a time gap there are no packets active (Johnson, C., 2021). If a TCP session is observed for an enough time period, we should be able to get a sequence containing "ON" and "OFF" gaps. This sequence is normally unique for a session. It is defined as a TTP. Two closely related TTPs coming from the incoming and outgoing connection of a host respectively can identify a stepping-stone. Similar ideas as TTP were proposed from 2004 to 2021 including the findings and the algorithms proposed in (Yoda, K., 2000) by K. Yoda and H. Etoh, (Blum, A., 2004) by A. Blum, etc., (Wang, X., 2001) (Wang, X., 2003) by X. Wang, etc., (He, T., 2007) by T. He and L. Tong, and (Yang, J., 2015) (Yang, J., 2016) (Yang, J., 2021) by J. Yang, etc. All the host-based SSID methods suffer from not only high false-positive errors, but also, more or less, inability of resisting in intruders' evasion using hacking techniques such as time-jittering or chaff perturbation.

In order to reduce the high false-positive errors produced in the HSSID methods, another type of SSID approach was introduced to detect SSI by estimating a connection chain length and referred to as the network-based SSID (NSSID). False-positive detection errors can be reduced by using NSSID methods, but they may introduce false-negative errors if the victim host happens to be very close to the sensor host. In 2002, K. H. Yung (Yung, K.H., 2002) proposed the first NSSID method to estimate approximately the length of a downstream connection chain (from the victim host to the sensor). He used the ratio between the RTT of Echo packet received at the sensor host from the victim host and the RTT of ACK packet received at the sensor from its next adjacent host in the same connection chain. With Yung's method, it is quite rough in estimating the downstream connection chain length, and thus it introduced high false-negative errors as discussed in (Yang, J., 2004). In order to estimate more accurately the downstream connection chain length, J. Yang and S. Huang (Yang, J., 2004) proposed a step-function method in 2005 by improving Yung's approach proposed in (Yung, K.H., 2002). However, the method proposed in (Yang, J., 2004) only works well within a local area network. An improved detection approach for SSID developed by J. Yang and S. Huang (Yang, J., 2007) in 2007 works effectively in the context of Internet. This method estimates the downstream connection chain length via mining network traffic. In most recently, a method using k-means to mine network traffic to estimate the connection chain length was proposed by L. Wang, etc. (Yang, J., 2021). All these existing NSSID methods use the distribution of packet round-trip times (RTTs) to estimate the length of a connection chain. More or less, NSSID approaches are still affected by intruders' session manipulation.

It is vital and significant to explore a new approach to address the above issues when intruders use hacking `techniques to evade detection, as well as to bring down significantly the false-negative and/or false-positive errors. In this paper, we conduct multidisciplinary research, and propose a novel idea of employing FFT signal processing technique to analyze the distribution of packet RTTs that is used to estimate a connection chain length for all the existing NSSID approaches. Using FFT signal processing technique to analyze the distribution of packet RTTs is a quite promising way to efficiently and precisely detect SSI as well as resist in intruders' evasion by using various hacking tools. We summarize our novel idea to analyze the distribution of packet RTTs as followings:

1. Collect the network traffic in a sensor from a TCP interactive session;
2. Identify the Send and Echo packets from the collected traffic and match them;
3. From the matched pair of Send and Echo packets, we can obtain the RTT for every Send packet;
4. Obtain a RTT function with the timestamp of a Send packet as its x axis, and the value of each RTT as its y axis by using an interpolation method;
5. Get the FFT value based on the RTT function.

As our follow-up research, we will explore an appropriate filter to analyze the FFT values. We then apply the filtered FFT values to precisely estimate a connection chain length, and thus we are confident that we will be able to develop new NSSID approaches that will be resistant to intruders' session manipulation with both false positive and negative errors significantly reduced, compared to all known SSID approaches.

The rest of this paper is organized as followings. In section 2, we present some preliminaries needed for this research; in section 3, we describe the algorithm of RTT interpolation and it's FFT; in section 4, we show some experimental results; and in section 5, we conclude the whole paper, and discuss the future research directions in this area.

2 Preliminaries

Before discussing FFT analysis for TCP/IP packets round-trip time (RTT) distribution, we first introduce the concepts of connection chain, Send and Echo packets, RTT and its distribution.

Connection Chain

Most hackers send attacking commands to a target system through a chain of stepping-stone hosts. An attacker can connect to a compromised host using a remote connection tool, such as OpenSSH, rlogin, or other similar tools. In this paper, we assume the tool OpenSSH is used to establish a connection chain by intruders. Similarly, the attacker can connect to the second compromised host, the third one, …, until to the targeted victim host.

As we mentioned before, the compromised hosts are called stepping-stones. A connection chain made by an intruder is composed of the intruder's host, stepping-stones, and the victim host. The amount of stepping-stone hosts can neither too large, nor too small. The more stepping-stone hosts used in a TCP connection chain, the safer for an intruder to launch their attacks. However, more stepping-stones used may cause a long network delay, as thus incur an inefficient communication. The program to detect an SSI must reside in one of the stepping-stone hosts compromised by intruders. This stepping-stone host is called a detection sensor, or a sensor. The connection from the victim host to the sensor is called downstream connection, and the one from the sensor back to the attacker's host is called upstream connection.

Send and Echo Packets

The tool OpenSSH can be used to make a TCP connection from a client-side port to the port 22 which is a SSH server-side port. When a connection is created, the user on the client side can operate the server via the connection chain. A request can send from the client to the server, and the server can process the received request to send response back to the client side. The request is called a Send packet, and the response to a request is referred to as the corresponding Echo packet. A large request, or response can be sent in multiple Send/Echo packets.

Technically, in a detection program, a Send or Echo packet can be identified using the following algorithm. A Send packet is identified as a packet with the TCP port "22" as the destination port, the client side IP address as the source IP address, the server side IP address as the destination IP address, and the flag having "Push" bit set up. An Echo packet is identified as a packet with port "22" as the source port, the client side IP address as the destination IP address, the server side IP address as the source IP address, and the flag having "Push" bit set up.

Round-Trip Time

In a TCP connection, a Send packet may incur more than Echo packets sent from the server side. In some cases, the following scenarios may happen: 1) several Send packets may be echoed by a single Echo one; 2) two or more Send packets may be echoed by more than one Echo packets. Generally speaking, if the program OpenSSH is employed in an interactive TCP connection session, the matching between Send and Echo packets is not one on one. If we can match an Echo packet with its corresponding Send packet, the time gap between the two matched packets is called a round-trip time, short name RTT.

The RTT between a matched pair of an Echo packet and its matched Send one in a connection chain can roughly reflect the connection chain length. Obviously, different matched pairs from the same connection chain may generate different RTT values. So the RTTs from different Send packets fluctuates along time. It is a function of timestamps of different Send packets. It was found that the values of RTTs from the same connection chain follows Poisson distribution (Yang, J., 2007).

3 RTT Interpolation

From each collected Send packet, if we can get its corresponding Echo packet, the RTT for the Send packet can be obtained. As we discussed before, the RTTs obtained from collected Send packets is a discrete function of time T. For the purpose of FFT analysis, we need to get the analog function of RTTs on T. Once we obtain RTT, and T which represents the timestamp of each Send packet, we can find a smooth function RTT = *f*(T) which pass through all points (RTT, T) of each test.

It is well-known that the equation of a linear function passing through two points (x_1, y_1) and (x_2, y_2) can be constructed via Lagrange interpolation formula as follows,

$$f(x) = \frac{x-x_1}{x_2-x_1}(y_2 - y_1) + y_1. \qquad (1)$$

Similarly, a smooth function passing through $\boldsymbol{n}$ points $(x_1, y_1), (x_2, y_2) \cdots, (x_n, y_n)$ can be constructed via the following Lagrange interpolation formula,

$$f(x) = \frac{(x-x_2)(x-x_3)\cdots(x-x_n)}{(x_1-x_2)(x_1-x_3)\cdots(x_1-x_n)} y_1 + \frac{(x-x_1)(x-x_3)\cdots(x-x_n)}{(x_2-x_1)(x_2-x_3)\cdots(x_2-x_n)} y_2$$

$$+\cdots+\frac{(x-x_1)(x-x_2)\cdots(x-x_{n-1})}{(x_n-x_1)(x_n-x_2)\cdots(x_n-x_{n-1})}y_n$$
$$=\sum_{i=1}^{n}\frac{\prod_{k\neq i}(x-x_k)}{\prod_{k\neq i}(x_i-x_k)}y_i \qquad (2)$$

For arbitrarily given $n$different points, what we obtain via the Lagrange interpolation formula is a polynomial function with the degree $n-1$. For any given point x, we can use the Lagrange interpolation formula to construct a polynomial function $y=f(x)$ and calculate the predicted value y_p for the given independent variable x. However, it is difficult (sometimes even impossible) for us to find all the coefficients of polynomial function $y=f(x)$, especially when $n>10$.

The following is the Lagrange Interpolation algorithm:

Algorithm: Lagrange Interpolation

1. Start
2. Read number of data n
3. Read data x_i and y_i $for\ i=1$ to n
4. Read independent value x_p
5. Initialize $y_p=0$
6. For $i=1$ to n
7. Set $p=1$
8. For $j=1$ to n
9. If $i\neq j$
10. Calculate $p=p*(x_p-x_j)/(x_i-x_j)$
11. End if
12. Next j
13. Calculate $y_p=y_p+p*y_i$
14. Next i
15. Display y_p as interpolate value
16. Stop

To find a Lagrange interpolation function, there are three different approaches that are described below:

1) Use all the data points of a data set to find a Lagrange interpolation function $y=f(x)$. Here and in what follows, we use the dataset obtained from AWS1 (an Amazon AWS server, please refer to Section IV.A for the detailed explanation) to show simulation result.

In Table 1, it shows if the given independent variable t belonging to the dataset T, that is $t\in T_s$, the predicted value y_p is same as observed RTT values. However, if t'does not belong to the dataset T, the corresponding predict value y_p is largely different from the observed RTTs values. This has been shown in Table 2. We obtained similar results for other AWS servers used.

Table 1: Predict Value for any given $t\in T_s$ Data Set

RTTs	t	Predicty$_p$
316.430	0.0	316.43
315.299	0.0159951	315.299
314.902	0.0327875	314.902
316.585	0.265648	316.585
…	…	…
314.799	8.5303443	314.799
317.885	8.6271807	317.885

Table 2: Predict Value for any given $t' \notin T_s$ Data Set

RTTs	t'	Predict y_p
316.430	0.0	316.43
315.299	0.0005	-1.4434e+34
314.902	0.001	-2.6694e+34
316.585	0.0015	-3.6974e+34
…	0.0585	-1.0096e+35
314.799	0.059	-1.0168e+35
317.885	0.0595	-1.0236e+35

2) With this method, we randomly selected a subset data of *m* points among all the *n* points in the dataset to find a Lagrange interpolation function $y = f(x)$. Tables 3 and 4 show the interpolation results with 11 points selected out of 101 points.

From the results shown in Table 3, it is trivial to see if the given independent variable t belong to the selected subset, the predicted value y_p is same as observed RTT values. However, if t'does not belong to the selected subset, the predicted value y_p is very different from the observed RTT values. Similar results can be obtained for other datasets.

Table 3: Predict Value for any given $t \in T_{subs} \subset T_s$ Subset

RTTs	t	Predict
315.192	0.035047654	315.192
315.018	0.03331984	315.018
315.976	0.077575357	315.976
314.971	0.042367935	314.971
…	…	…
317.347	0.043111637	317.347
315.226	0.009168.298	315.226

Table 4: Predict Value for any given $t' \notin T_{subs} \subset T_s$ Subset Data

RTTs	t'	Predict
316.43	0.0	10373.716
315.299	0.0159951	9555.476
314.902	0.0327875	8752.346
316.585	0.0265648	2079.002
…	…	…
315.009	0.04792148	271.26
314.7	4887.954	239.881

From case 1) and 2), we found that the system accumulates errors when increasing the degree of Lagrange interpolation function. The polynomial function obtained via the Lagrange interpolation formula is not a good fit for the dataset in our research project. We look at the third case as the following.

3) With this approach, we will find a good fit function RTT = *f*(t) based on all the *n* points of the dataset. Suppose there exists a polynomial function

$$f(t) = a_0 + a_1 t + a_2 t^2 + \cdots a_m t^m \qquad (3)$$

that fits the dataset.

Substitute all the points of the dataset into the above polynomial function (3) and write the result in matrix form, we have

$$\begin{pmatrix} 1 & t_1 & t_1^2 & \cdots & t_1^m \\ 1 & t_2 & t_2^2 & \cdots & t_2^m \\ 1 & t_3 & t_3^2 & \cdots & t_3^m \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ 1 & t_n & t_n^2 & \cdots & t_n^m \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} = \begin{pmatrix} RTTs_1 \\ RTTs_2 \\ RTTs_3 \\ \vdots \\ RTTs_n \end{pmatrix} \Leftrightarrow \text{TA=RTT} \quad (4)$$

The fit function (3) can be determined once we get the coefficients in matrix A. It is trivial to see that

$$\text{A}=(T'T)^{-1}T'\text{RTT}$$

satisfies the above equation (4).

With this method, instead of using all the data collected from AWS1 server, we apply different number of elements from the dataset, such as 3, 4, 5, …, to see if we can get the best fit.

First, we try m = 1, and randomly pick up two elements from the dataset to compute a_0, a_1. We then obtain a RTT function $(t) = a_0 + a_1 t$. Similarly, we try different value for *m*, and get different RTT functions. For each function obtained, we use the predicted value to determine which RTT function is the best fit. For the dataset collected from AWS1 server, we found the best-fit RTT function when *m* = 9. The results from Table 5 and Table 6 show that the predicted value obtained from the best-fit function $a_0 + a_1 t + a_2 t^2 + \cdots + a_9 t^9 = F(t)$ are closer to the observation than other predicts.

Table 5: Predict Value for any given $t \in T_s$ Subset Data

RTTs	*t*	Predict
316.43	0	315.815
315.299	0.0159951	315.829
314.902	0.0327875	315.841
316.585	0.265648	315.788
317.099	0.4176257	315.672
314.888	0.4320359	315.662
314.818	0.451224	315.648
314.71	0.4672484	315.637

Table 6: Predict Value for any given $t' \notin T_s$ Subset Data

RTTs	*t*′	Predict
	0	315.815
	0.0005	315.815
	0.001	315.816
	0.0015	315.816
	0.002	315.817
	0.0025	315.817
	0.003	315.818
	0.0035	315.818

4 Experimental Results and Analysis

Network Setup

We set up a network connection chain from Columbus State University (CSU) to Amazon AWS servers. We ran SSH from one computer host in Cybersecurity Lab, CSU, named CCT30 with Ubuntu Linux installed. We connected to the first AWS server (AWS1) from CCT30. AWS1 is located in Northern Virginia with IP of "34.239.127.118". We then connected to the second AWS server (AWS2) from AWS1. AWS 2 is located in Frankfurt, Germany with IP of "52.59.96.142". The connection chain was

further extended to the third AWS server (AWS3) from AWS2. AWS3 is located in London, UK with IP of "18.133.230.26". Finally, we connected to the last AWS server (AWS4) located in Tokyo, Japan with IP of "54.199.163.186" from AWS 3.

Data Collection

Any network traffic generated at CCT30 will go through the connection chain set up in 4.1 and arrive at AWS4. We run TcpDump to collect network traffic at CCT30, AWS1, AWS2, and AWS3 respectively.

To collect the data at each host, we ran a TcpDump command to filter out non-needed packets. The following command was ran for every host except the victim itself, as it collects the downstream packets.

sudo tcpdump –nn –tt '(tcp dst port 22 && src IPlocal) ||

Tcp src port22 && dst IPlocal) &&

ip[2:2]>52>HostID_AttackX_TestX.txt,

where IPlocal refers to a given host's IP address; ip[2:2]>52: remove all 0 length packets used for establishing a connection and all ACK and SYN signal packets; ID_AttackX_TestX.txt: refers to a text file that can store the results from TcpDump.

The following shows two packets captured and stored in the text file with the first one sent to the SSH sever, and the second one echoed from the server.

1644853535.518797 IP 168.27.2.107.43870 > 34.239.127.118.22: Flags [P.], seq 4291344119:4291344155, ack 2768127312, win 501, options [nop,nop,TS val 606453850 ecr 2151956101], length 36

1644853535.646703 IP 168.27.2.107.43870 > 34.239.127.118.22: Flags [P.], seq 36:72, ack 1, win 501, options [nop,nop,TS val 606453978 ecr 2151986183], length 36

For our experiments, we designed three attacking scenarios as shown in the following. For every "attack scenario", we performed ten tests on a connection, and recorded the data at the four separating hosts. We ran the three "attacking scenarios", and obtained a total of 120 TcpDump text files which store the packets captured. We use these files as our dataset to conduct FFT analysis. The ultimate goal is to see if we can apply results from the FFT analysis to do stepping-stone upstream detection.

First Attacking Scenario – Attacker 1

On SSH Session
pwd
whoami
sudo su
ls
cd/etc
ls –a
cp/etc/shadow/tmp/shadowCopy
chmod 755/tmp/shadowCopy

On another Computer
Scp –p buntu@59.199.163.186:/tmp/shadowCopy
./Documents
On sSH Session

Rm/tmp/shadow Copy
Exit
Second Attacking Scenario – Attacker 2
On SSH Session
whoami
pwd
cd/home/ubuntu
ls
nano text_file.txt
//paste a large text file and save it
cat hello.txt
exit

Third Attacking Scenario – Attacker 3
On SSH Session
whoami
pwd
cd/home/ubuntu
ls
nano text_file.txt
//enter a few sentences and save the text file
cat hello.txt
exit

Data Processing

For each tcpdump ".txt" file, we made a Python program Send-Echo-Splitter.py to process it and generate two separate files. One is called Send packet file which records all of the timestamps for each Send packet. Another is called Echo packet file recording all the timestamps for each Echo packet.

We use the data mining clustering approach published (Yang, J., 2021) in 2021 to match an Echo packet with its corresponding Send packet from the above two Send and Echo packets files. The matched pair could be utilized for computing the RTT for each corresponding Send packet. The following shows some matched results. In each row, the first column represents the RTT values computed, the second column tells us the index of the matched packet pair (Send Index, Echo Index), and the third column records the timestamp for each Send packet.

331552.000000, (51,51), {1645207699408398}
332175.000000, (52,52), {1645207699528256}
332029.000000, (53,53), {1645207699720625}
331973.000000, (54,56), {1645207700367961}
331762.000000, (55,57), {1645207700495874}
331568.000000, (56,58), {1645207700647937}
332507.000000, (57,59), {1645207700831916}
332528.000000, (58,60), {1645207701063912}

The following criteria are also used to filter out the incorrect "extra" packet matches.

1. A particular send packet can only match with one echo packet
2. A particular echo packet can only match with one send packet

3. In the case of multiple uses of one send packet, the first match found, where the RTT is within a particular variance (10% for our tests) of the current running RTT, is kept. All other matches using that send packet discarded. If no match found is within that variance, then the first found match kept. This is considered to be an outlier match. (And almost always have a much larger or smaller value than the average)
4. Whenever an echo packet found in a match, and that echo packet has already been found to be in a correct match, the initial match is discarded. This occurs even if it would discard all matches for a particular send value.

Experimental Results

Collected Data

We collected all the data packets from CCT30, AWS1, AWS2, and AWS3 respectively. We will not show all the packets captured here, but the following are two sample packets captured at CCT30.

Packet 1

1644853535.863699 IP 168.27.2.107.43870 > 34.239.127.118.22: Flags [P.], seq 72:108, ack 37, win 501, options [nop,nop,TS val 606454195 ecr 2151986499], length 36

Packet 2

1644853535.979056 IP 34.239.127.118.22 > 168.27.2.107.43870: Flags [P.], seq 37:73, ack 108, win 471, options [nop,nop,TS val 2151986627 ecr 606454195], length 36

Packet 1 is a request packet (Send) sent from CCT30 to AWS1, and Packet 2 is the response packet (Echo) sent from AWS1 to CCT30. Packet 2 is originally sent from AWS4.

Processed Data

Packet 1 and Packet 2 are a packet pair that matches with each other. From a matched packet pair, we can compute its corresponding RTT which can reflect the connection chain length from CCT30 to AWS4. We can run an algorithm for packet-matching (Yang, J., 2021) to match all the packets captured from CCT30, AWS1, AWS2, AWS3, respectively. The following Table 7 shows part of the matching results at CCT30 and AWS1. In this table, the column RTT with unit Millisecond represents the round-trip time for match packet pairs. The column T with unit Second represents the timestamp at which each Send was captured.

Table 7: Part of the Matched Data from Data Sets CCT30 and AWS1

CCT30Attack1test2		**AWS1Attack1test2**	
RTT(ms)	T(s)	RTT(ms)	T(s)
333396.0	1644852311589120.0	316430.0	1644852348576518.0
332313.0	1644852311749088.0	315299.0	1644852348736469.0
331806.0	1644852311917076.0	314902.0	1644852348904393.0
333567.0	1644852314245633.0	316585.0	1644852351232998.0
334749.0	1644852315765399.0	317099.0	1644852352752775.0
331869.0	1644852315909481.0	314888.0	1644852352896877.0
333298.0	1644852316101481.0	314818.0	1644852353088758.0
331649.0	1644852316261634.0	314710.0	1644852353249002.0
332051.0	1644852316381259.0	315009.0	1644852353368666.0

Interpolation RTT Time Function Results

Figure 1 shows the interpolation RTT=f(t) function for CCT30 based on the data in Table 7, where the Y axis contains the RTT values in millisecond, and the X axis contains time in second. The red line represents the observations of data set obtained from CCT30. The green line represents the predict value obtained from the interpolated function RTT=*f*(t).

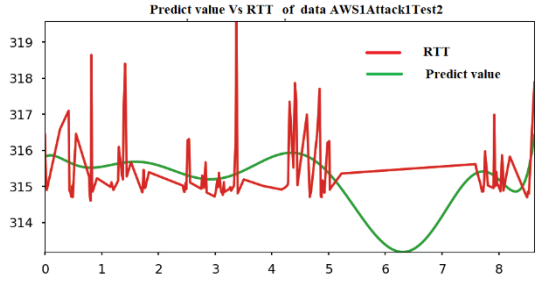

Figure 1: Interpolated RTT Function at CCT30

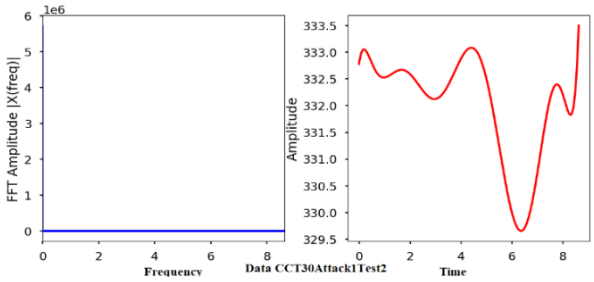

Figure 2: FFT Results from the RTT Function at CCT30

FFT Results from the Interpolated Time Function

Figure 2 shows the FFT results from the RTT function at CCT30. The right one shows the RTT simulated function. Figure 3 shows the filtered FFT results at CCT30. The left figure shows the FFT results before filtering. The right one shows the FFT results after filtering. The reason we applied a filter to the FFT results is that we can examine the FFT value in details from CCT 30, AWS1, AWS2, and AWS3. We found the unfiltered FFT values remains the same for CCT30, AWS1, AWS2, and AWS3. We design the filter as the following: for any given cut-off frequency, here, we use cut_off = 5Hz, and assume that the FFT amplitudes are set to zero when the absolute frequencies is less than the cut-off value. The blue curve in the right of Figure 3 shows the relationship between the filtered FFT amplitude (where absolute frequencies larger than the cut-off) and frequency of data set from CCT30.

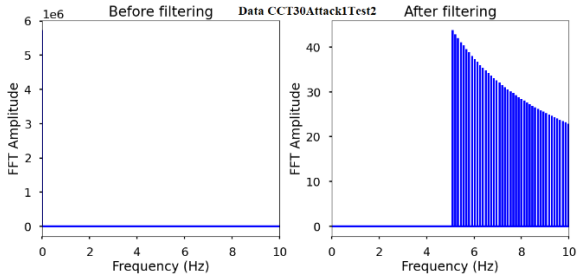

Figure 3: FFT Filtered Results at CCT30

5 Conclusion and Future Work

All the known network-based SSID methods used the distribution of packet RTTs to estimate a connection chain length. This paper proposed the FFT transformation approach to analyze the distribution of packet RTTs. A critical RTT time function was obtained by using FFT interpolation based on matched pairs of Echo and Send packets. We then applied the FFT analysis to the resulting RTT time function.

As for future research direction, we will explore an appropriate filter to analyze the FFT values. We then apply the filtered FFT values to precisely compute a connection chain length, and thus we are confident that we will be able to develop new NSSID approaches that will be resistant to attackers' evasion by using hacking tools such as time-jittering and/or chaff perturbation, with both false positive and negative errors significantly reduced, compared to all the known NSSID approaches.

References

[1] Blum, A., Song, D., & Venkataraman, S. (2004). Detection of interactive stepping stones: Algorithms and confidence bounds. In *Recent Advances in Intrusion Detection: 7th International Symposium, RAID, Sophia Antipolis, France. Proceedings 7*, 258-277. Springer Berlin Heidelberg.

[2] He, T., & Tong, L. (2007). Detecting encrypted stepping-stone connections. *IEEE Transactions on Signal Processing*, *55*(5), 1612-1623.

[3] Johnson, C., Khadka, B., Ruiz, E., Halladay, J., Doleck, T., & Basnet, R.B. (2021). Application of deep learning on the characterization of tor traffic using time-based features. *Journal of Internet Services and Information Security (JISIS)*, *11*(1), 44-63.

[4] Staniford-Chen, S., & Heberlein, L.T. (1995). Holding intruders accountable on the internet. In *Proceedings IEEE Symposium on Security and Privacy*, 39-49.

[5] Wang, X., & Reeves, D.S. (2003). Robust correlation of encrypted attack traffic through stepping stones by manipulation of interpacket delays. *In Proceedings of the 10th ACM conference on Computer and communications security*, 20-29.

[6] Wang, X., Reeves, D.S., Wu, S.F., & Yuill, J. (2001). Sleepy watermark tracing: An active network-based intrusion response framework. *In Trusted Information: The New Decade Challenge 16*, 369-384. Springer US.

[7] Yang, J. (2016). Resistance to chaff attack through tcp/ip packet cross-matching and rtt-based random walk. *In IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, 784-789.

[8] Yang, J., & Huang, S.H.S. (2004). A real-time algorithm to detect long connection chains of interactive terminal sessions. *In Proceedings of the 3rd international conference on Information security*, 198-203.

[9] Yang, J., & Huang, S.H.S. (2007). Mining TCP/IP packets to detect stepping-stone intrusion. *computers & security*, *26*(7-8), 479-484.

[10] Yang, J., & Wang, L. (2021). Applying MMD Data Mining to Match Network Traffic for Stepping-Stone Intrusion Detection. *Sensors*, *21*(22), 1-18.

[11] Yang, J., & Zhang, Y. (2015). RTT-based random walk approach to detect stepping-stone intrusion. *In IEEE 29th International Conference on Advanced Information Networking and Applications*, 558-563.

[12] Yang, J., Wang, L., Shakya, S., & Workman, M. (2021). Identify encrypted packets to detect stepping-stone intrusion. In *International Conference on Advanced Information Networking and Applications*, 536-547. Cham: Springer International Publishing.

[13] Yoda, K., & Etoh, H. (2000). Finding a connection chain for tracing intruders. In *Computer Security-ESORICS: 6th European Symposium on Research in Computer Security, Toulouse, France. Proceedings 6*, 191-205. Springer Berlin Heidelberg.

[14] Yung, K.H. (2002). Detecting long connection chains of interactive terminal sessions. In *Recent Advances in Intrusion Detection: 5th International Symposium, RAID 2002 Zurich, Switzerland, Proceedings 5*, 1-16. Springer Berlin Heidelberg.

[15] Zhang, Y., & Paxson, V. (2000). Detecting stepping stones. In *USENIX Security Symposium*, *171*, 1-11.

Authors Biography

Dr. Lixin Wang

Dr. Lixin Wang is currently a Full Professor of Computer Science at Columbus State University, Columbus, Georgia. He holds a Ph.D. degree in Computer Science from Illinois Institute of Technology, Chicago IL. His research interests include Network Security, Intrusion Detection Systems, Wireless Networks, Algorithm Design and Analysis.

Dr. Jianhua Yang

Dr. Jianhua Yang is currently working at TSYS School of Computer Science, Columbus State University (CSU), Columbus, GA, USA as a Full Professor. Before joining CSU, he was an Assistant Professor at Bennett College from 2006 to 2008, University of Maryland Eastern Shore from 2008 to 2009, and Associate Professor at Beijing Institute of Petro-Chemical Technology, Beijing, China from 1990 to 2000.

Mr. Maochang Qin

Maochang Qin is currently a graduate student at the TSYS School of Computer Science in Columbus State University, Columbus, GA, USA. He is a part-time faculty in the Mathematics Department at Columbus State University. He has worked on several research and commercial projects in the fields of cybersecurity, machine learning, bioinformatics, symmetry and conservations, etc.