Modelling Network Traffic and Exploiting Encrypted Packets to Detect Stepping-stone Intrusions*

Jianhua Yang†, Lixin Wang, and Suhev Shakya
TSYS School of Computer Science, Columbus State University, Columbus, GA 31904, USA
{yang_jianhua, wang_lixin, shakya_suhev}@columbusstate.edu

Received: December 27, 2021; Accepted: February 5, 2022; Published: February 28, 2022

Abstract

In order to avoid being detected, most professional intruders have exploited stepping-stones to make a long connection chain to launch their attacks indirectly, other than directly, since 1990s. The longer a connection chain, the harder to capture the intruders and detect their intrusions. Most existing approaches suffer from intruders' session manipulation, such as chaff perturbation. In this paper, we propose a novel algorithm by modelling network traffic and exploiting encrypted packets to detect stepping-stone intrusions. The experimental results show that the proposed algorithm cannot only detect stepping-stone intrusions, but also resist intruders' single-side chaff perturbation up to 70% in the context of a local area network, as well as 80% in the context of the Internet. The algorithm presents much stronger performance in resisting intruders' both-side chaff perturbation. Our study shows if the incoming and outgoing connections of a sensor host are both manipulated, the algorithm can resist intruders' chaff rate up to 140%, and even more, regardless of a local area network or the Internet environment.

Keywords: Stepping-stone Intrusion, modelling network traffic, encrypted packet, Intrusion Detection

1 Introduction

1.1 Overview of Stepping-stone Intrusion and its Detection

Stepping-stones [2] have been widely used by intruders to launch their attacks since early 1990s. The primary reason of exploiting stepping-stones to launch attacks is to protect intruders themselves from capture. Since most intruders tend to establish a long connection chain by spanning multiple stepping-stones to conduct their attacks, it is extremely challenging to capture such type of intruders. Detecting such kind of attacks is, sometimes, difficult. The attacks launched by exploiting stepping-stones are called stepping-stone intrusion (SSI). The way to detect stepping-stone intrusions is called stepping-stones intrusion detection (SSID).

The basic idea of SSID is to determine if a computer host is used as a stepping-stone. Such a stepping-stone is normally called a sensor. A sensor is actually a host where we can capture network traffic and run a detection program. Since 1990s, there have been many different algorithms developed to detect stepping-stone intrusions. These algorithms can be classified as two categories: host-based stepping-stone intrusion detection (HSSID), and connection-based stepping-stone intrusion detection (CSSID).

*This is an extended journal version of the short conference paper [1] published on the proceedings of the 35th International Conference on Advanced Information Networking and Applications, Toronto, Canada, May 2021.

†Corresponding author: TSYS School of Computer Science, Columbus State University, 4225 University Avenue, Columbus, GA, USA 31907, Tel: 001-706-507-8180, Web: `http://csc.columbusstate.edu/yang`

The idea of HSSID is to collect the network packets coming to a host which are called incoming traffic, and the packets leaving from the host which are called outgoing traffic, then compare the two traffics to see if there exists a relayed connection pair. Unlike the HSSID, CSSID is to estimate the length of a connection chain from a sensor to the victim host which is the end host of the connection chain. If the length is more than three connections which indicate more than three stepping-stones are used to access the victim host, it is highly suspicious that the sensor is used as a stepping-stone.

1.2 Related Work and Its Limitations

Some typical approaches developed for SSID belonging to HSSID include packet content-thumbprint [3], packet time-thumbprint [2], connection deviation [4], packet number difference [5], connection watermark [6], [7], [8], and detect bounded memory chaff [9]. The packet content-thumbprint to detect stepping-stone intrusion was proposed by S. Staniford-Chen, and L. T. Heberlein [3] in 1995. They designed a thumbprint approach by hashing the contents of the packets captured in the connection of a host to describe the behavior of the network traffic. By comparing the thumbprint of an incoming connection with the thumbprint of an outgoing connection of a host, it is trivial to decide if the host is used as a stepping-stone. Packet content-thumbprint approach is easy to be circumvented by an encrypted connection. Packet time thumbprint can overcome this issue to detect SSI via an encrypted session. For each TCP/IP packet captured, we can record its timestamp. If we can sniff a certain amount of packets, we could obtain a sequence in which each element indicates the timestamp of a packet captured. It is rational to convert the timestamp sequence to a time gap sequence by computing the timestamp difference between each packet and its subsequent one. So in this way, we get a timestamp gap sequence which is called time-thumbprint. Comparing the time-thumbprint from the incoming connections with the ones from the outgoing connections can help us determine if the host is used as a stepping-stone. The higher the similarity of two time-thumbprints, the higher the probability that a host is used as a stepping-stone. Yoda and Etoh [4] proposed a connection deviation-based approach to detect stepping-stone intrusion. This method is based on the observation that the deviation for two un-relayed connections is large enough to be distinguished from the deviation of those connections within the same connection chain.

A. Blum, et al. [5] proposed a packet number difference-based (PND-based) approach that detects stepping-stones by checking the difference of the number of Send packet between two connections. This method is based on the idea that if two connections are relayed, the difference between the numbers of packets from the two connections, respectively, is bounded. This method can resist intruders' evasions, such as time jittering and chaff perturbation. D. L. Donoho, et al. [10] showed for the first time that there are a theoretical limit on the ability of attackers to disguise their traffics using evasions during a long interactive session. The major problem with the PND-based approach is that the upper bound on the number of packets required to monitor is large, while the lower bound on the amount of chaff an attacker needs to evade the detection is small. This fact makes Blum's method very weak in resisting to intruders' chaff evasion. X. Wang, et al. [6], [7], [8] conceived an approach using watermarks to decide relayed connections. Injecting a watermark to a TCP/IP session may result in lots of computations, thus making the approach inefficient. Another issue is that the injected watermarks may be manipulated by intruders. T. He and L. Tong proposed an algorithm DBDC (DETECT-BOUNDED-MEMORY-CHAFF) [9] for SSID with bounded memory or bounded delay perturbation. It is stated that DBDC can deal with chaff evasion and tolerate a number of chaff packets proportional to the size of the attacking traffic. Their study shows that an intruder needs to insert at least $n/(1+\lambda\Delta)$ chaff packets in n packets to evade DBDC detection if the packets delay is bounded by Δ. This tells us the chaff-rate of DBDC is $1/(1+\lambda\Delta)$, where λ is a parameter of a Poisson distribution which indicates the expected number of occurrences during a given time interval. It is obvious that a smaller λ and Δ can make DBDC tolerate more chaff, but could also make DBDC suffer from a high false alarm probability for a wide range of normal traffic.

Some typical approaches belonging to CSSID include packet round-trip time (RTT) approach proposed by K.H. Yung in 2002 [11], step-function approach [12], and clustering-partitioning algorithm [13] proposed by J. Yang, et al. in 2004 and 2007 respectively. In Yung's method, two RTTs are computed: one is the RTT from a sensor to the victim host which is called RTTe; another one is the RTT between the sensor and its adjacent host (next immediate neighbor in the connection chain) which is denoted as RTTa. The ratio between RTTa and RTTe can be used to estimate the length of a downstream connection chain. The step-function approach [12] takes a different way from Yung's method to estimate the length of a connection chain. It collects and matches all the Send and Echo packets to compute the RTTs for all Send packets. The RTTs could form different Steps since the RTTs belonging to different stages of a connection chain would go to different clusters with each of them representing one step. Simply counting the number of steps can tell us how many hosts are used as stepping-stones. Clustering-partitioning data mining approach [13] is a method to detect SSI by estimating the length of a connection chain without matching TCP/IP packets. Matching TCP/IP packets, in some cases, are infeasible. This approach makes use of the distribution of RTTs to find the RTTs using data mining approach. On the contrary, the computed RTTs can also provide us the packet-matching results.

In 2010, Y. Zhang, etc. proposed a context-based TCP/IP packets matching approach to detect stepping-stone intrusion [14]. It is claimed that the approach can resist intruders' manipulation. A dynamic programming technique was proposed to detect multi-hop stepping-stone pairs by Y. Kuo, etc. in 2010 [15]. However, the dynamic approach cannot resist intruders' chaff perturbation attack. Y. Sheng, etc. developed a data mining approach to mine network traffic to detect stepping-stone intrusion efficiently [16] in 2012. The issue that if the approach can resist intruders' chaff manipulation was not discussed in the paper. In 2015, J. Yang and Y. Zhang proposed a random-walk approach to detect stepping-stone intrusion [17] which claims the algorithm can resist intruders' chaff perturbation attack to around 20%. In 2016, J. Yang proposed an approach using packet cross-matching and random walk to detect stepping-stone intrusion [18]. This method can resist intruders' chaff attack up to 40%. L. Wang, etc. designed an algorithm using k-Means clustering to mine network traffic to detect stepping-stone intrusion [19] in 2021. This approach does not perform well in terms of resisting intruders' session manipulation.

1.3 Summary of the Main Contribution and the Paper Layout

The common issue of the above approaches to detect stepping-stone intrusion is that their capabilities to resist intruders' manipulation are more or less limited. A preliminary result via identifying encrypted packets to detect stepping-stone intrusion was published in a conference proceeding [20]. The main contribution of the published preliminary research is to explore the possibility of using encrypted packets to detect stepping-stone. In this paper, in order to improve the resistance performance to intruders' chaff manipulation, we propose a novel approach by making use of the features of encrypted packets. The main contribution of this paper is to justify the capability of the proposed algorithm to resist intruders' chaff manipulation. After analysing encrypted packets captured in an OpenSSH session, we observed that the contents of a packet at application layer are encrypted. The fields of a packet header in other layers are not encrypted. The most important observation we obtained is that the length of a single encrypted character in transport layer keeps the same no matter what the character is. Some header fields remain the same between two OpenSSH connections if the two connections are relayed which means the host is used as a stepping-stone. The length of an encrypted packet can be used to detect stepping-stone since it remains unchanged regardless of the encryption key and its content. But here, we assume both sides use the same encryption algorithm. If we combine both the header fields and the length of an encrypted packet, we can not only detect stepping-stone intrusion, but also resist intruders' manipulation. Our experimental results in both a local area network and the Internet show this approach can improve resistance performance to intruders' manipulation to a new high level.

This is paper is organized as the following. Section 2 introduces the preliminaries used in other sections. Section 3 presents how to understand an encrypted packet. In Section 4, we propose a way to model network traffic. In Section 5, we design an algorithm to detect stepping-stone intrusion by employing encrypted packets, as well as a parser to parse network traffic. In Section 6, we present experimental results and its analysis. The whole paper is concluded and the future work is discussed in Section 7.

2 Preliminaries

2.1 Downstream and Upstream Connections

As we have mentioned, in order to protect themselves from capture, most professional intruders launch their attacks via a long interactive session that is also called a long connection chain. A long connection chain is defined as a TCP session that spans at least three computer hosts. The reason using three computer hosts here is that some legal applications may use one or two hosts as stepping-stones. But we rarely see a legal application using three or more computer hosts as stepping-stones. An interactive TCP session can be made by using many different tools, such as Telnet, rlogin, OpenSSH, and so on. OpenSSH is one popular tool used to access a remote host. It is also widely used by intruders to establish a long connection chain and launch their attacks. In Figure 1, it shows a long connection chain from Host 0 to Host N. An attacker can establish a connection chain starting from Host 0, then connecting to Host 1, Host 2,..., Host i-1, Host i, Host i+1,..., until Host N. Here, Host 0 is controlled by an attacker; Host 1 to Host N-1 are the stepping-stones used by the attacker; Host N is assumed a victim host. If Host i is used as a sensor, the chain from Host 0 to Host i is defined as an upstream connection, and the one from Host i to Host N is called a downstream connection.

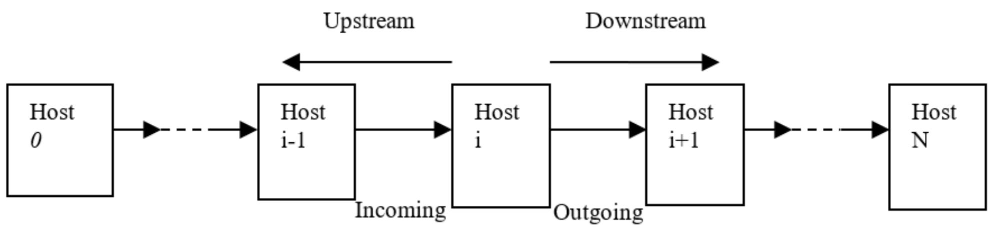


Figure 1: A long connection chain

2.2 Send and Echo Packets

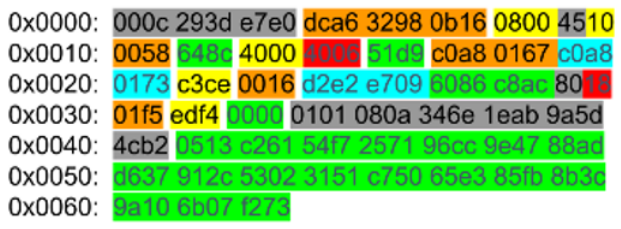
As long as a long connection chain is established, just as shown in Figure 1, an intruder can access Host N from Host 0 via the chain. Whatever a keystroke typed at Host 0 by the intruder, the packets will be forwarded to Host 1 from Host 0, then to Host 2, Host 3, until to Host N. Each keystroke is encapsulated into a request packet. Each request packet can go all the way down to Host N from Host 0 along the connection chain. The request packet is defined as a Send packet. When a Send packet arrives at Host N, it will be received, de-capsulated, and processed. A response packet corresponding to each Send packet can be sent back to Host 0 from Host N along the same connection chain, but in a reverse direction. This response is called an Echo packet. One Send packet can normally incur one Echo packet at Host N, but sometimes may trigger multiple Echo packets. For example, if an intruder types a UNIX command ”ls” at Host 0, two Send packets representing ”l” and ”s” respectively, will be forwarded to Host N along the connection chain. Two Echo packets corresponding to ”l” and ”s”, respectively, will be incurred at Host N, and come back to Host 0. However, as long as an ”Enter” is entered at Host 0, the third Send packet is forwarded to Host N, and this will incur multiple Echo packets at Host N depending on the execution results of the command ”ls” at Host N.

2.3 An Encrypted Packet and its Header Fields

SSL is a layer in between the application and the transport layers. A packet in the application layer including its payload and header fields can be encrypted in SSL layer, then pass to the transport layer. At the transport layer and the layers below, headers can be added to an encrypted packet, but the header fields in the transport layer, network layer, data link layer are not encrypted. A packet captured in different layers can present different headers. Both the existing packet sniffing tools, such as Wireshark and TCPdump, and self-developed packet capturing programs using WinPcap (MS Windows related Operating System (OS)) or Libpcap (Unix/Linux related OS) can sniff packets in the transport layer and the layers below. We do not consider any packet header fields in the application layer for our detection algorithm since they are encrypted in SSL. The header fields in transport, network, and datalink layers, respectively, are in our consideration to design an algorithm to detect a stepping-stone intrusion. In the transport layer, some header fields can be used to determine a relayed session pair (will be defined later) since they remain unchanged from an incoming connection to an outgoing connection of the same host. These fields include destination port, source port, and TCP flags. Sequence number, acknowledgement number, offset, window size, checksum, urgent pointer and options cannot be used to detect a stepping-stone intrusion since they are packet/session oriented. In the network layer, none of the header fields of a packet can be considered since they remain either the same or packet/session oriented except the source IP address, destination IP address, and Total Length fields. Similarly, in the data link layer, even though there are many header fields, such as destination MAC address, source MAC address, protocol type, or CRC code, none of them can be considered in our algorithm design.

3 Understanding Encrypted Packets

In order to design an algorithm and make a program to detect stepping-stone intrusion by inspecting each packet captured, it is necessary to understand each field of the header in a packet and know how to read the header information from an encrypted packet. In this section, we use a captured packet as an example to illustrate what each binary number in a header exactly means. Before the discussion, it is worth to mention that, when capturing packets using TCPdump, a network interface must be specified. Otherwise, the destination and source MAC addresses would not be obtained because TCPdump captures packets in ”Cooked” mode which has a different type of frame header from the standard. In this experiment, we captured a packet from a computer host with MAC address 00:0c:29:3d:e7:e0 and IP address 192.168.1.115. The packet was sent from a computer host with MAC address dc:a6:32:98:0b:16 and IP address 192.168.1.103 via an ssh connection. In Figure 2, it shows the captured packet in a binary format. The first six bytes ’000c293de7e0’ from 0x0000 to 0x0005 represent the destination MAC address of the host used to capture the packet. The second six bytes ’dca632980b16’ represent the source MAC address of the host from which the packets were sent out. The next two bytes ”0800” represent Ethernet type: Ethernet Type II. The first number of the second last byte ”4” indicates the IP version, and the second number ”5” represents the network IP header length=5*32 bits=160 bits = 20 bytes. The last byte ”10” (in binary bits: 00010000) represent the Type of Service (ToS) with its first three bits ”000” known as precedence bits, the next 4 bits (1000) indicate the ToS of ”Minimize Delay”, and the last bit is left unused. The first two bytes of the second row starting from 0x0010 ”0058” (Hex number)



```
0x0000: 000c 293d e7e0 dca6 3298 0b16 0800 4510
0x0010: 0058 648c 4000 4006 51d9 c0a8 0167 c0a8
0x0020: 0173 c3ce 0016 d2e2 e709 6086 c8ac 8018
0x0030: 01f5 edf4 0000 0101 080a 346e 1eab 9a5d
0x0040: 4cb2 0513 c261 54f7 2571 96cc 9e47 88ad
0x0050: d637 912c 5302 3151 c750 65e3 85fb 8b3c
0x0060: 9a10 6b07 f273
```

Figure 2: A captured packet in a binary number format

stand for the total length of the IP packet, which is 88 bytes in decimal number system. So the bytes from 0x000E (byte '45') to 0x0065 (byte '73') form the whole IP packet include the IP header and the payload. Its first 20 bytes from 0x000E '45' to 0x0021 '73' denote the IP header fields. The second two bytes '648c' represent the identification number of the IP packet. If the packet is fragmented to smaller parts during the transmission, all the fragments share the same identification number. The first byte of the third two bytes '40' (0100 0000) is the fragmentation flag and offset byte. The second bit '1' is set to represent "don't fragment', the third bit '0' shows no more fragment since the packet is not fragmented. The second byte of the third two bytes '00' is the fragmentation offset in case of a fragmented datagram. The first byte of the fourth two bytes '40' is the TTL field (Time To Live) indicating the number of the hops the packet is allowed to pass through in the Internet. The second byte of the fourth two bytes '06' represents the transport layer protocol "TCP" is used. The next two bytes '51d9' are the header checksum field. The next four bytes 'c0a80167' represent the source IP address 192.168.1.103, and following four bytes 'c0a80173' are the destination IP address 192.168.1.115. The rest part starting from 0x0022 'c3' to the last byte 0x0065 '73' is the TCP packet, which is also the payload of the IP packet. The first two bytes 'c3ce' are the source port number, and the second two bytes '0016' are the destination port number 22 in decimal number system, which is exactly the known ssh server port. The next four bytes "d2e2e709' show the sequence number, and the following four bytes "6086c8ac" represent the acknowledgement number. The first 4 bits of the next byte "80" (1000 0000) show the TCP header length and the remaining 4 bits are reserved. The TCP header length expresses a 32-bit word, which must be multiplied by 4 to calculate the total byte value. Therefore, we can get 8*4=32 bytes, the TCP header length. The last byte '18" (Hex number, 00011000 in binary) is the TCP flag bits which represent "CWR, ECE, UR, ACK, PSH, RST, SYN and FIN" with each bit in order. The fourth and fifth bits are set to show this is a packet carrying data, as well as acknowledging the previously received packets. The first two bytes of the row 0x0030 '01f5' are the window size field. The second two bytes 'edf4' are the TCP checksum field, and the third two bytes '0000' represent the Urgent pointer. This field is 0 since the flag Urgent bit is not set. We all know that the minimum TCP header length is 20 bytes. Therefore, we can calculate the TCP options length for this packet, 32-20 = 12 bytes. The TCP packet payload is 12 bytes after the urgent pointer '0000', which is from the byte 0x0034 to the byte 0x0035. The bytes '0101 080a 34e6 1eab 9a5d 4cb2' are the TCP options. The payload bytes are from 0x0042 byte '05' to 0x0065 byte '73'.

4 Modelling Network Traffic

Computer network communication is a complex process, which is dominated by TCP/IP protocols. Different protocols may result in different type of packets. In the TCP/IP protocol suite, there are more than 100 different type of protocols. To serve the purpose of SSID, we may not need all different types of TCP/IP packets. In this section, we will examine a computer network traffic, and build a model that fits to the goal of SSID. Before modelling a network traffic, we discuss the process to encrypt a packet in a computer network communication.

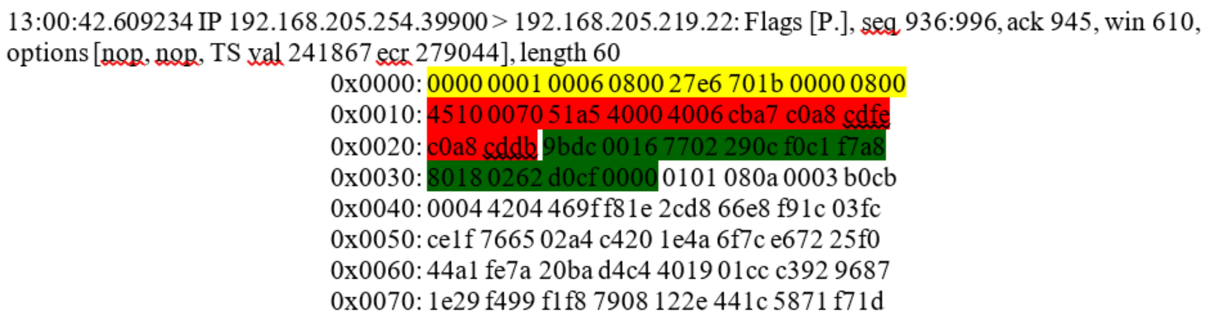
4.1 The Process of Packet Encryption

OpenSSH is widely used by regular users to access and operate a server remotely. Most intruders also use this tool to launch their attacks on a remote server. The difference between them is that a regular user may access a remote server directly. However, a malicious user may indirectly access a remote server via a long connection chain. One major advantage to use OpenSSH is that it can provide a secured network communication. Each packet sent from the first host (an intruder host) of a connection chain to the last host (a victim host) is encrypted with different keys in different connections. In order to understand the proposed algorithm to detect stepping-stone intrusion, we first introduce how to encrypt and deliver a packet in a TCP session including multiple connections. What shown in Figure 1 is a session chain established using OpenSSH, which is composed of N connections from Host 0 to Host N with each connection having its own encryption key. We assume Host 0 is operated by an intruder, and Host N is a victim. When a connection is established, an encryption key is selected. For example, when the intruder makes a connection from Host 0 to Host 1, the system asks the user to input a password for authentication, and if it is the first time connecting to Host 1 from Host 0, the system also reminds the user to click ”Yes” to accept a public key which is for a session key distribution. As long as the user connects to Host 1 successfully, an encryption key is generated and distributed from Host 0 to Host 1. Apparently, different connections have different encryption keys. For convenience, we assume the encryption keys for the connections Host 0 to Host 1, Host 1 to Host 2, ..., Host i-1 to Host i, Host i to Host i+1, ..., Host N-1 to Host N are EnpK0,1, EnpK1,2, ..., EnpKi-1,i , EnpKi, i+1,..., EnpKN-1, N, respectively. When an attack is launched, each packet is sent from Host 0 to Host 1 encrypted with EnpK0,1, from Host 1 to Host 2 encrypted with EnpK1,2, ..., from Host N-1 to Host N encrypted with EnpKN-1,N. In each stepping-stone host, a packet is received from its incoming connection. Then the packet is decrypted and re-encrypted, and sent out through its outgoing connection.

We cannot obtain the content of every packet at each stepping-stone due to the secured design of OpenSSH. OpenSSH is an application layer program, which provides a secured communication over an unsecured network channel. The secured SSL layer lays in between application layer and transport layer. A packet in the application layer can be encrypted first, then passed to the transport layer and be encapsulated into a segment. Each segment contains a transport layer header and payload, which is the encrypted application layer message. A transport layer segment can be passed to network layer and be encapsulated into a datagram, which contains a network layer header, and payload, which is the transport layer segment. Similarly, a network-layer datagram can be the payload of a data-link layer frame. We use a captured packet to show the three different headers and the encrypted payload. Figure 3 shows the header and payload information of a captured packet from a computer running a Linux OS. We can see the IP addresses, port numbers, TCP flags, sequence and acknowledgement numbers, windows buffer size and the packet length in transport layer. In Figure3, it clearly shows the data-link header (yellow), IP header (red), and TCP header (green). The application layer header and the packet contents are not readable because they are all encrypted.

4.2 The Length of an Encrypted Packet

We are interested in the length of a TCP packet since this length remains the same under the same encryption algorithm regardless of the values of the encryption keys and the contents of the packet. We found that for a certain encryption algorithm used in OpenSSH, the length of an encrypted string depends on not the content of the string, but the number of the characters in the string. We believe this observation is significant because this feature can be used to detect stepping-stone intrusion. Before we go further



```
13:00:42.609234 IP 192.168.205.254.39900 > 192.168.205.219.22: Flags [P.], seq 936:996, ack 945, win 610,
options [nop, nop, TS val 241867 ecr 279044], length 60
                                     0x0000: 0000 0001 0006 0800 27e6 701b 0000 0800
                                     0x0010: 4510 0070 51a5 4000 4006 cba7 c0a8 cdfe
                                     0x0020: c0a8 cddb 9bdc 0016 7702 290c f0c1 f7a8
                                     0x0030: 8018 0262 d0cf 0000 0101 080a 0003 b0cb
                                     0x0040: 0004 4204 469f f81e 2cd8 66e8 f91c 03fc
                                     0x0050: ce1f 7665 02a4 c420 1e4a 6f7c e672 25f0
                                     0x0060: 44a1 fe7a 20ba d4c4 4019 01cc c392 9687
                                     0x0070: 1e29 f499 f1f8 7908 122e 441c 5871 f71d
```

Figure 3: A captured packet via TCPdump

to discuss our novel detection algorithm, we explain the length of an encrypted packet in detail. For the convenience of our discussion, we use a simplified model of a stepping-stone intrusion as shown in Figure 4 where the stepping-stone host has two connections: the incoming connection C_{in} and the outgoing connection C_{out}. To verify the idea, we run OpenSSH to connect to the Stepping-stone host from Host 0 and connect out to Host 1 to make two connections as shown in Figure 4. We typed some characters to activate some packets at Host 0. The packets can be delivered to Stepping-stone via C_{in} and forwarded to Host 1 via C_{out}. An encryption algorithm is used in the two connections in which obviously encryption keys are different. All the three hosts run Ubuntu OS. We use TCPdump to capture all the packets coming from Host 0 at Stepping-stone, and also all the packets going to Host 1 from Stepping-stone.

Our goal is to check the lengths of the packets captured at the incoming and outgoing connections of Stepping-stone, respectively. In order to make grouped characters into one packet, we use copy and paste. We typed 26 characters separately, as well as grouped characters, such as ”ab”, ”abc”, ”abcd”, and so on. The IP address, port number, flags, timestamp, and the total length of each packet are recorded. Table 1 shows the header fields of the packets captured at the incoming and outgoing connections of Stepping-stone, respectively. From the header information of the packets captured, the IP addresses of Host 0, Stepping-stone, and Host 1 are 192.168.205.254, 192.168.205.219, and 192.168.205.236, respectively. Host 0 uses a port number 34652 sending packets to the SSH server in Stepping-stone. Stepping-stone uses a port number 55824 forwarding packets received from Host 0 to the SSH server of Host 1. If you check the timestamps of the packets from the incoming and outgoing connections C_{in} and C_{out}, you can observe a little bit time lag.

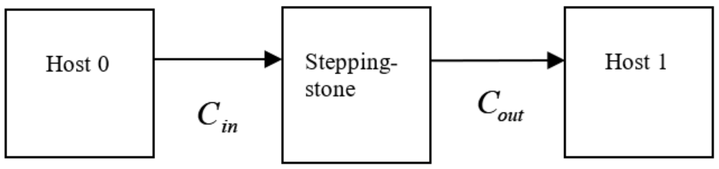


Figure 4: A Simplified model of a stepping-stone intrusion

From the results in Table 1, we conclude the following. 1) Different single characters have the same length of its encrypted packet. 2) Grouped characters (string) may have different lengths for its encrypted packet depending on the length of the string. 3) The same number of characters result in the same length of encrypted packet at the incoming and outgoing connections, respectively. Single characters ’x’, ’y’, and ’a’ have the same length of encrypted packet 36. When the length of the string is in between 3 and 10, the length of the encrypted string is 44. It is easy to see from Table 1 that the encrypted string with the number of characters in between 11 and 18 has length of 52. When we increased the length of the string to be 19, the length of the encrypted packet becomes 60. We did not show more characters of a string in Table 1, but we know that the length is increased as long as the number of the characters in a string reach a certain degree.

Table 1: Captured packets comparison between the incoming and outgoing connections

Character(s)	Incoming connection	Outgoing connection
x	14:27:10.936370 IP 192.168.205.254.34652 > 192.168.205.219.22: Flags [P.], length 36	14:27:10.936712 IP 192.168.205.219.55824 > 192.168.205.236.22: Flags [P.], length 36
y	14:27:18.461314 IP 192.168.205.254.34652 > 192.168.205.219.22: Flags [P.], length 36	14:27:18.461411 IP 192.168.205.219.55824 > 192.168.205.236.22: Flags [P.], length 36
a	12:57:52.262636 IP 192.168.205.254.39900 > 192.168.205.219.22: Flags [P.], length 36	12:57:52.262809 IP 192.168.205.219.58534 > 192.168.205.236.22: Flags [P.], length 36
abc	12:58:15.767057 IP 192.168.205.254.39900 > 192.168.205.219.22: Flags [P.], length 44	12:58:15.767348 IP 192.168.205.219.58534 > 192.168.205.236.22: Flags [P.], length 44
abcdefghijk	12:59:00.740578 IP 192.168.205.254.39900 > 192.168.205.219.22: Flags [P.], length 52	12:59:00.740870 IP 192.168.205.219.58534 > 192.168.205.236.22: Flags [P.], length 52
abcdefghijklmnopqrs	13:00:37.931633 IP 192.168.205.254.39900 > 192.168.205.219.22: Flags [P.], length 60	13:00:37.932061 IP 192.168.205.219.58534 > 192.168.205.236.22: Flags [P.], length 60

4.3 Packet Stream

From either the incoming connection or the outgoing connection of a sensor host, packets can be captured and put into a queue, which is called a packet stream. The Send/Echo packets can be identified and put into a queue based on their timestamp order. The packet queue obtained from the incoming connection of a sensor is called an incoming packet stream. Similarly, the one obtained from the outgoing connection of a sensor is called an outgoing packet stream. In each incoming/outgoing packet stream, it contains Send and Echo packets only. If we put all the Send packets into a queue, the queue can be called a Send stream. A queue containing Echo packets only is called an Echo stream.

4.4 Relayed Connections

As shown in Figure 1, Host i is used as a stepping-stone of the connection chain. Detection program runs at Host i, which is called a sensor. The connection from Host i-1 to Host i is called an incoming connection of Host i. Similarly, the connection from Host i to Host i+1 is called an outgoing connection of Host i. If an incoming connection and an outgoing connection of Host i belong to the same TCP connection session, they are called relayed connections/sessions. Detecting if a host is used as a stepping-stone is actually to find if there exist a relayed session pair. We already knew that the way to determine a relayed session pair is to see if the same packet appears on both the incoming and outgoing connections of a host. For an encrypted session, the content of a packet cannot be observed. So comparing the header fields of the encrypted packets from incoming and outgoing connections respectively can be a feasible way to find a relayed session pair.

5 Algorithm Design for SSID

As long as network traffic is modelled and the traffic data are collected, two packet streams including send and echo packets can be obtained. Each packet stream can be handled as a timestamp sequence. The statement to detect stepping-stone intrusion becomes a timestamp sequence comparison. In this section, we will present our algorithm to detect stepping-stone intrusion, as well as the approach to compare two timestamp sequences.

5.1 The Detection Algorithm

The way to determine if a host is used as a stepping-stone is to check if there is a relayed connection pair passing through the host. If we monitor an enterprise level server, we can observe tons of incoming connections, as well as a bunch of outgoing connections. Most servers provide services to their clients. This means, in most cases, the connections entering into an enterprise server are not necessary to connect out of the server. If we find an ssh session connecting into a server, as well as connecting out from the server, it is highly suspicious that the server is used as a stepping-stone. So stepping-stone intrusion detection needs to compare all the incoming connections of a server with all the outgoing connections of the server to determine if there exists any relayed connection pairs. In this paper, we focus on a simplified model as shown in Figure 5 in which the server has only one incoming connection and one outgoing connection. Our goal is to compare the two connections to see if the host H1 is used as a stepping-stone. As shown in Figure 5, the host H1 has an incoming connection C_{in} and an outgoing connection C_{out}. We run TCPdump at H1 to capture the packets from the connections C_{in} and C_{out}, respectively. When capturing packets, it is important to be aware that the source IP and destination IP addresses cannot be used in TCPdump filter because before we start capturing, we actually do not know which host connects to H1 and to which host H1 will connect. We also do not know the source port number used to connect to H1, as well as the port number H1 uses to connect out to a remote host. However, we can use destination port number 22 since they are all ssh connections, as well as the IP address of H1 which can be used for packet capturing from the incoming connection, as well as the outgoing connection.

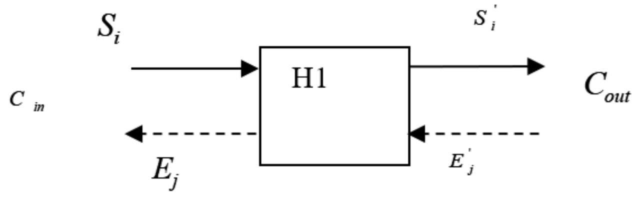


Figure 5: A Simplified Model of Stepping-stone with only one incoming connection and only outgoing connection

When we only use the length of captured packets to determine a stepping-stone, it is possible to introduce false-positive errors. The reason is that if two different users type the same command at their hosts respectively, it is hard to tell the packets captured at H1 actually coming from two different hosts. In order to fix this issue, we identify the packets captured at H1 as two different categories: Send and Echo. When an intruder or a user types any Unix/Linux command, the packet of each character from each keystroke can arrive at the host H1 via its incoming session, and be forwarded to the host connected by the outgoing connection. Finally, each packet invoked from the intruder's host comes to the end host of the connection chain which is the victim of the intruder. We call this type of packet a Send packet. For each received Send packet, the victim host echoes the request, and the echoed response can go back to the intruder's host along the connection chain reversely. We call this type of packet an Echo packet.

However, how can we identify a Send/Echo packet from a connection chain technically?

The way to identify if a captured packet is a Send or an Echo is to make use of both the packets flow direction and its corresponding TCP header flags. As shown in Figure 5, either the incoming connection or the outgoing connection has both Send and Echo packets. In the incoming connection of H1, a Send is defined as a packet that its destination IP is the IP address of H1, the destination port number is 22, and its TCP flags have PSH bit set up. An Echo is defined as a packet that the source IP address is the IP address of H1, the source port number is 22, and its TCP flags have PSH bit set up. In the outgoing connection of H1, a Send is defined as a packet that its source IP is the IP address of H1, the destination port number is 22, and its TCP flags have PSH bit set up. An Echo is defined as a packet that its destination IP address is the IP address of H1, the source port number is 22, and its TCP flags have PSH bit set up. Using these rules, it is trivial to write a TCPdump filter, or make our own program to capture not only the packets sent from H1, but also distinguish them to be Send or Echo packets. As shown in Figure 5, for the incoming connection, the Send packet is denoted as S_i, and the Echo packet as E_j. Similarly, for the outgoing connection, the Send packet is denoted as S_i', and the Echo packet as E_j'. The significance to introduce Send and Echo packets is that the length of each Send packet from different users may be the same, but as long as a command is executed at a victim host, the length of each Echo packet might be different since they are echoed from different servers, which may result in different packets sizes from the responses.

We put the captured packets from the incoming connection of the host H1 into a packet stream $\{p_{in}\}$ with m packets, as well as the ones from the outgoing connection into $\{p_{out}\}$ with n packets. If the two connections are relayed, m is either equal to or close to n. We check each packet in the input stream $\{p_{in}\}$, and then decide not only the type of each packet (Send or Echo), but also its length. We obtain the following timestamp sequence $C_{in} = \{(S_i \text{ } or \text{ } E_i, \text{ } Len)\}$ for the input stream $\{p_{in}\}$, as well as a timestamp sequence $C_{out} = \{(S_i' \text{ } or \text{ } E_i', \text{ } Len)\}$ for the output stream $\{p_{out}\}$.

The next step is to compare the similarity between the two timestamp sequences C_{in} and C_{out}. We adopted a sequence comparison algorithm proposed by S. Wu in 1990 [20] to compute the similarity between two timestamp sequences. We assume there are two sequences C_{in} with length M, and C_{out} with length N. It is also assumed that $N \geq M$ without loss of generality. The way to compare the two sequences C_{in} and C_{out} and is to find either the longest common subsequence (LCS) or a shortest edit script (SES). Here it is assumed that we find a SES from the comparison of two sequences. An edit script is to edit one sequence to another one via delete/insert actions in which "delete" action specifies which character in sequence C_{in} to be deleted and "insert" action specifies which character in sequence C_{out} to be inserted. A SES is an edit script whose length is the minimum among all the possible edit scripts that edit a sequence C_{in} to a sequence C_{out}. The similarity between two sequences can be defined as the number of delete/insert actions needed to edit the sequence C_{in} to the sequence C_{out} . For the details of sequence comparison algorithm, please refer to the paper [20].

We summarize the proposed algorithm to detect stepping-stone intrusion in **Algorithm 1**. If a connection is manipulated by an intruder, such as chaff perturbation manipulation, it may affect the similarity between the two timestamp sequences and further affect SSI detection performance. We will study the impaction to the performance of the proposed algorithm in Section 6. But we need to pay attention to a fact that is an intruder can only chaff a connection, but cannot remove any original packets from a connection. In Section 6, we will use some experimental results completed in both a local area network and the Internet to justify the performance of Algorithm 1.

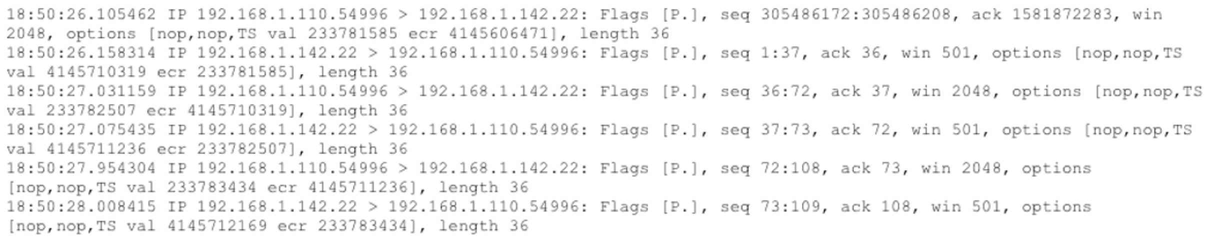
5.2 Parsing Algorithm

The step before comparing all the incoming and outgoing streams of traffic data is to process the file containing the captured packets and extract the useful information. We used a connection chain where the IP of the attacker (here we assume we know the attacker's IP address) is 192.168.1.110, which connected

```
Algorithm 1: The Detection Algorithm
  Input  : Similarity threshold $\Delta$
  Step 1: Capture packets using TCPdump and put them into a stream $\{p_{in}\}$ or $\{p_{out}\}$
  Step 2: Apply Parsing Algorithm (See Section 5.2 below) to the header information of each
    packet to convert each stream into a timestamp sequence or
  Step 3: Call a sequence comparison algorithm (see Section 5.3 below) to obtain the similarity $\beta$
    between the two timestamp sequences
  Step 4: if $\beta > \Delta$, a stepping-stone intrusion is detected
  End
```

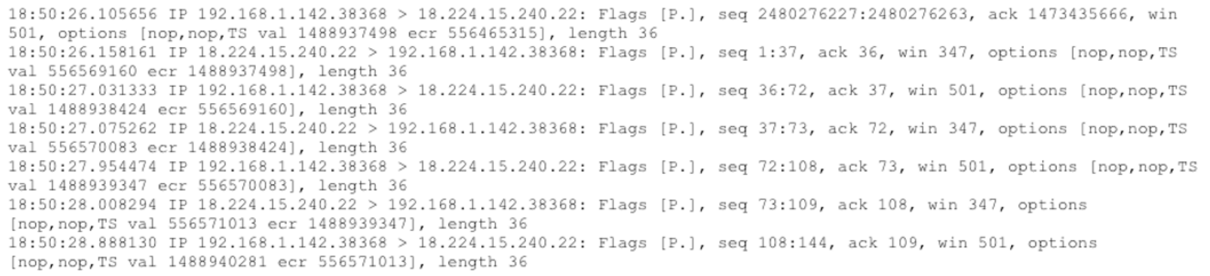
to the sensor with IP address of 192.168.1.142, which then connected to the victim computer with IP address: 18.224.15.240. In Figure 6, it shows an example of the packets captured on the incoming-connection side of the sensor, and in Figure 7, it shows an example of packets captured on the outgoing-connection side of the sensor.

For the file parser algorithm (**Algorithm 2**), we need to know the file name and what side of the connection it belongs to (incoming or outgoing) with respect to the sensor. We first access the collected data file and read through the file line by line until the end of the file. Each packet is then split into an array delimited by blank spaces. Once we have each packet in an array, we know the source IP and the destination IP are in index 2 and 4 respectively. Here, we have to handle the incoming data and outgoing data in different ways. If the packet's destination is the sensor, and the packet comes from the incoming side of the connection, it is a Send packet. If the same packet had the source same as the sensor IP, then it is an Echo packet, and vice versa for the outgoing side of the connection. We identify each packet as a Send or an Echo using tags 's' or 'e' respectively and append the packet length to it. We save this information for every packet in the given file and return it as a string for the next step.



```
18:50:26.105462 IP 192.168.1.110.54996 > 192.168.1.142.22: Flags [P.], seq 305486172:305486208, ack 1581872283, win
2048, options [nop,nop,TS val 233781585 ecr 4145606471], length 36
18:50:26.158314 IP 192.168.1.142.22 > 192.168.1.110.54996: Flags [P.], seq 1:37, ack 36, win 501, options [nop,nop,TS
val 4145710319 ecr 233781585], length 36
18:50:27.031159 IP 192.168.1.110.54996 > 192.168.1.142.22: Flags [P.], seq 36:72, ack 37, win 2048, options [nop,nop,TS
val 233782507 ecr 4145710319], length 36
18:50:27.075435 IP 192.168.1.142.22 > 192.168.1.110.54996: Flags [P.], seq 37:73, ack 72, win 501, options [nop,nop,TS
val 4145711236 ecr 233782507], length 36
18:50:27.954304 IP 192.168.1.110.54996 > 192.168.1.142.22: Flags [P.], seq 72:108, ack 73, win 2048, options
[nop,nop,TS val 233783434 ecr 4145711236], length 36
18:50:28.008415 IP 192.168.1.142.22 > 192.168.1.110.54996: Flags [P.], seq 73:109, ack 108, win 501, options
[nop,nop,TS val 4145712169 ecr 233783434], length 36
```

Figure 6: An example of incoming packets collected



```
18:50:26.105656 IP 192.168.1.142.38368 > 18.224.15.240.22: Flags [P.], seq 2480276227:2480276263, ack 1473435666, win
501, options [nop,nop,TS val 1488937498 ecr 556465315], length 36
18:50:26.158161 IP 18.224.15.240.22 > 192.168.1.142.38368: Flags [P.], seq 1:37, ack 36, win 347, options [nop,nop,TS
val 556569160 ecr 1488937498], length 36
18:50:27.031333 IP 192.168.1.142.38368 > 18.224.15.240.22: Flags [P.], seq 36:72, ack 37, win 501, options [nop,nop,TS
val 1488938424 ecr 556569160], length 36
18:50:27.075262 IP 18.224.15.240.22 > 192.168.1.142.38368: Flags [P.], seq 37:73, ack 72, win 347, options [nop,nop,TS
val 556570083 ecr 1488938424], length 36
18:50:27.954474 IP 192.168.1.142.38368 > 18.224.15.240.22: Flags [P.], seq 72:108, ack 73, win 501, options [nop,nop,TS
val 1488939347 ecr 556570083], length 36
18:50:28.008294 IP 18.224.15.240.22 > 192.168.1.142.38368: Flags [P.], seq 73:109, ack 108, win 347, options
[nop,nop,TS val 556571013 ecr 1488939347], length 36
18:50:28.888130 IP 192.168.1.142.38368 > 18.224.15.240.22: Flags [P.], seq 108:144, ack 109, win 501, options
[nop,nop,TS val 1488940281 ecr 556571013], length 36
```

Figure 7: An example of outgoing packets collected

Algorithm 2: The Parsing Algorithm

```
function fileReader(File file, String channelType): String
Begin
    int srcIpIdx:= 2
    int dstIpIdx:=4
    String sensorIP= //ip address of the sensor machine
    String output := ””
    while file has nextLine
    Begin
        temp:= read nextLine from File
        aLine:= split temp on ” ”
        if(channelType == ”outgoing”)
        Begin
            if(aLine[srcIpIdx] == sensorIP)
            Begin
                output:= $output +' s' + aLine[len(aLine) - 1] +' \n'$
            End
            else if (aLine[dstIpIdx] == sensorIP)
            Begin
                output:= $output +' e' + aLine[len(aLine) - 1] +' \n'$
            End
        End
        else if(channelType == ”incoming”)
        Begin
            if(aLine[srcIpIdx] == sensorIP)
            Begin
                output:= $output +' e' + aLine[len(aLine) - 1] +' \n'$
            End
            else if (aLine[dstIpIdx] == sensorIP)
            Begin
                output:= $output +' s' + aLine[len(aLine) - 1] +' \n'$
            End
        End
    End
    fileReader:= output
End
```

5.3 Sequence Comparison Algorithm

For the purpose of connection matching, we adopt the O(NP) sequence comparison algorithm proposed in [20]. Given any two sequences, A and B, of length M and N, respectively, this algorithm calculates the shortest edit distance from the smaller sequence to the larger one. Here, the edit distance is the total number of deletions and insertions that is required to change sequence A into B. In this algorithm, the two sequences are casted into a grid-like graph that's called an edit graph. The grid can have horizontal, vertical, or diagonal edges. The horizontal edges correspond to an insertion and the vertical ones correspond to a deletion. Diagonal edges mean the two sequences are similar at the diagonal. If we give every horizontal and vertical edge a weight/cost of 1 and every diagonal edge a 0, the solution for calculating the shortest edit distance can be given by the shortest path from source(0,0) to sink(M,N) in the edit graph.

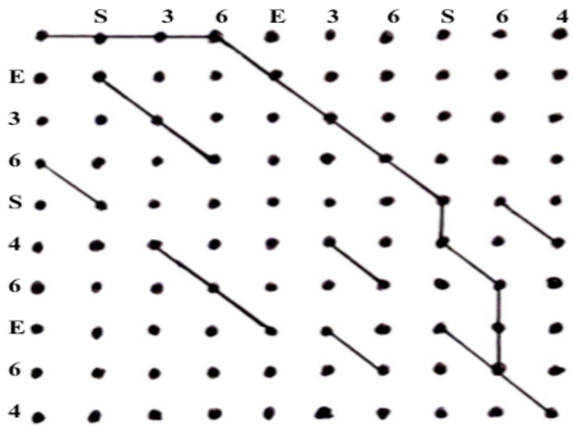


Figure 8: Edit Graph of A and B

Let A = 'E 3 6 S 4 6 E 6 4' and B = 'S 3 6 E 3 6 S 6 4'. The shortest edit script to transform A into B is: we first insert 'S 3 6' to the beginning of A, keep 'E 3 6 S' as it is, delete '4', keep '6' as it is, delete 'E 6' and keep the '4' in the end. In the edit script, we deleted 3 elements and inserted 3, giving the edit distance of 6. We can calculate this number by counting the number of vertical and horizontal edges on the edit graph shown in Figure 8. Let each diagonal in the edit graph be denoted by k, where k=y-x. The diagonals are numbered from -M to N. Let P denote the number of deletions in the edit script, D denote the shortest edit script, and Δ be the difference between N and M. Wu's algorithm only examines the vertices between diagonals -P and $\Delta + P$. This algorithm computes sets of vertices with the furthest y value with a certain D-value until the sink is reached within diagonals -P and $\Delta + P$.

Algorithm 3 (Sequence Comparison Algorithm) starts by initializing fp[], which is a set of D-values for each point in the diagonals from -M to N, to -1 since we do not know their D-values at the start. We also define a function "snake(k, y)" that computes the longest sequence on a diagonal k starting from y. Back in the main program, we use the snake function, and fp[k+1] and fp[k-1] to compute fp[k]. We start from diagonal 0 and divide the edit graph into three different parts which will be handled differently. For all k from -p to Δ, we call the function snake(k, max-of(fp(k - 1,p) + 1, fp(k + 1,p - 1)). After that part is done, we compute snake(k, max-of(fp(k - 1,p) + 1, fp(k + 1,p - 1)) for k from Δ+p down to Δ. Finally, we call snake(Δ, max-of(fp[Δ-1]+1, fp[Δ+1])) for fp on diagonal Δ.

6 Experimental Verification and Result Analysis

In order to test the performance of the proposed algorithm, we designed an experiment to collect and analyse network packets. The first part of the experiment is to test the performance of the algorithm in a local area network. The second part is to test the performance of the algorithm in the Internet environment. In this section, we discuss the experimental setup, data collection, packets manipulation, the comparison result, and the result analysis.

6.1 Experimental Setup, Data Collection and Manipulation

We set up three connection chains with each one consisting of three computer hosts: Attacker, Sensor, and Victim. The three connection chains share the same sensor as shown in Figure 9. The algorithm was

Algorithm 3: The Sequence Comparison Algorithm

```
function   edit-distance(String A, String B)
Begin
        M = length-of-A
        N = length-of-B
        int fp[-(M+1),......,(N+1)]:= -1
        int p:= 1
        repeat
                p:= p+1
                for k:= -p to Δ
                        fp[k]:= snake(k, max-of(f[k-1]+1,fp[k+1]))
                for k:= Δ + p to Δ, step -1
                        fp[k]:= snake(k, max-of(fp[k-1]+1, fp[k+1]))
                fp[Δ]:= snake(Δ, max-of(fp[delta-1]+1, fp[Δ+1]))
        until fp[Δ]= N
End
function   snake(k, y)
Begin
        int x:=y-k
        while x < M and y < N and A[x] = B[y]
                Begin
                        x:=x+1
                        y:=y+1
                End
                snake:=y
End
```

tested under two different networks: a local area network, and the Internet. In a local area network setup, the attacker, sensor, and victim hosts are all located in the same network. In the context of the Internet, the attacker and the sensor hosts are both in the same local area network, but the victim hosts are AWS servers from Amazon.com. The AWS servers were physically located in Ohio, Washington, and Florida, respectively.

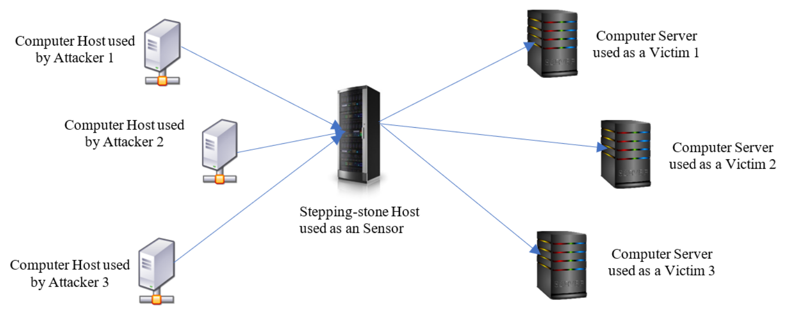

Computer Host used by Attacker 1
Computer Host used by Attacker 2
Computer Host used by Attacker 3
Stepping-stone Host used as an Sensor
Computer Server used as a Victim 1
Computer Server used as a Victim 2
Computer Server used as a Victim 3

Figure 9: Experimental Setup

In both local area network and the Internet context, the attacker and sensor hosts are both under our control so we can make network traffic from attackers' hosts, and collect the network packets from the sensor host. We simulate three attackers using three different scripts to access three different victims respectively in three different tasks. The first attacker logs into the victim as a root user, accesses and downloads the shadow file from the ./etc directory. The second attacker opens a file already in the victim machine and appends data onto it. And the third attacker deletes and creates a modified version of a file already on the victim computer. The command scripts used by the three attackers are listed below, respectively:

First attacker:

```
pwd
whoami
sudo su
ls
cd /etc
ls -a
scp -p shadow attacker_username@attacker_IP:/home/seed/Documents
exit
```

Second attacker:

```
whoami
pwd
cd /home/seed/Documents
ls
nano text_file.txt
ls
cat hello.txt
exit
```

Third attacker:

```
whoami
pwd
cd /home/seed/Documents
ls
nano hello.txt
ls
cat hello.txt
exit
```

The attackers executed their different tasks simultaneously and their network traffic were collected in the sensor host and stored in separate files. Each text file contains the packet header information for the packets captured. As shown in Figure 6, some sample packets for an incoming connection are captured. And Figure 7 shows some packets for an outgoing connection. We used TCPdump to capture all the incoming and outgoing packets for all three of the connection chains. The packets were filtered for only Send and Echo packets collected. The commands used for the incoming and outgoing packets collection are listed below, respectively, where '-x' can be 1, 2, or 3 to represent different connection chains.

- tcpdump -i interface-id -nn '(tcp[13]&8!=0)' and dst sensor-ip and dst port 22 > incoming-x.txt
- tcpdump -i interface-id -nn '(tcp[13]&8!=0)' and src sensor-ip and src port 22 > outgoing-x.txt

In the above commands, the 'tcp[13]&8!=0' filter ensures we capture only those packets that have their push-flag up. The packets can have a [P] representing a push packet or a [P.] representing a combination of push and Ack packet. We filter out all the Ack packets since they do not contain any useful information for the SSI detection purpose. We manipulated the data collected by chaff-perturbation. The goal of chaff-perturbation is basically to insert some meaningless packets to defeat an SSI detection algorithm. In our simulation, we randomly inject 10%, 20%, 30%, ... packets into the original packets collected to obtain a chaffed data set respectively. The manipulated data set with different chaff rates can be used to test the resistance performance to chaff-perturbation for the algorithm proposed. To get a complete resistance performance evaluation, we chaffed not only one side of the sensor (either the incoming or the outgoing connection), but also both the incoming and outgoing connections. The former one is called chaff-single, and the latter one is called chaff-both. In this paper, we show the performance results of the proposed SSID algorithm to resist intruders' chaff-perturbation in both chaff-single and chaff-both.

6.2 Experimental Results and Its Analysis on a Local Area Network

We first verified the resistance performance of the proposed SSID algorithm on a local area network. As shown in Figure 9, all three Attackers, one Sensor, and three Victims are located in our department lab. Each attacker made its own network traffic to its corresponding victim via the same sensor. For each connection, all the incoming and outgoing network traffics are collected and stored into incoming-x.txt and outgoing-x.txt respectively where 'x' can be 1, 2, or 3 to represent the connection number. We captured data for each of the attacker scenarios as described in Section 6.1 and repeated the experiment for twenty times. Each data set identifies the incoming and outgoing traffic for the experiment in different files. For simplification, we use short names to name them as i1, i2, i3 for the incoming collected data for attacker one, two, and three respectively. Similarly, o1, o2, and o3 identify the outgoing data for attacker one, two, and three respectively. We ran the proposed algorithm to get the similarities between the incoming and outgoing network traffic for the same connection chain, as well as for different connection chains. If the similarity for the same connection chain is apparently larger than the one for different connection chains, stepping-stone intrusion can be detected. Table 2 shows the similarity average results over twenty-time experiments between each incoming and each outgoing connection of the sensor.

Table 2: SSID Similarities in a local area network

Incoming traffic / Outgoing traffic	i1	i2	i3
o1	0.940787	0.240007	0.433973
o2	0.208546	0.846652	0.520239
o3	0.418891	0.663608	0.920904

In Table 2, it clearly shows that if an incoming and an outgoing connection are in the same chain, the similarities are above 84%, otherwise, they are less than 66%. This indicates the proposed algorithm can detect stepping-stone intrusion.

In order to assess the performance of the proposed algorithm to resist intruders' chaff-perturbation manipulation, we injected meaningless packets from 10%, 20%, ..., upto 70%, then computed the similarities between the connections in the same chain, and then compare with the similarities between the connections from different chains. We can chaff single side or both sides of the sensor. Table 3 shows the similarity results by chaffing single side. In Table 3, column 2 through column 8 of row 1 represents the chaff rates. Table 4 shows the similarity results by chaffing both sides from 10% to 140%. In Table 4, column 2 through column 15 of row 1 represents the chaff rates. In these two tables, "Sim 1" represents the similarity between the incoming traffic and the outgoing traffic in the connection chain made by attacker one. Similar representations are for Sim 2, and Sim 3, respectively. Term ”UnSim 1” represents the maximum similarity between the incoming connection 1 and the outgoing connection 2 and 3 respectively. Similar definition holds for ”UnSim 2”, and ”UnSim 3”. Figure 10 and Figure 11 are the charts to show the similarities in Table 3 and Table 4 respectively. Figure 10 shows a performance crossing between Sem 1 and UnSem 3 when the chaff rate increased to 70%. This means the proposed algorithm can resist intruders’ chaff perturbation at most 70% if single side is manipulated. What shown in Figure 11 tells us that, if both sides are manipulated, the proposed algorithm can resist intruders’ chaff perturbation up to 140%.

Table 3: SSID similarities in a local area network under single-side chaff

Similarity \ Chaff Rate	0.1	0.2	0.3	0.4	0.5	0.6	0.7
Sim 1	0.902559	0.867671	0.833115	0.800531	0.770316	0.731533	0.71475
Sim 2	0.833218	0.829717	0.827477	0.816155	0.799215	0.781755	0.765408
Sim 3	0.905536	0.887944	0.864960	0.838407	0.815215	0.787652	0.762895
UnSim 1	0.448398	0.477159	0.502036	0.52663	0.549517	0.571408	0.594105
UnSim 2	0.624611	0.591216	0.56061	0.533237	0.507746	0.484474	0.463203
UnSim 3	0.554379	0.586153	0.615367	0.642404	0.666057	0.69031	0.711866

Table 4: SSID similarities in a local area network under both-side chaff

Similarity \ Chaff Rate	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4
Sim 1	0.899	0.887	0.880	0.875	0.872	0.871	0.868	0.866	0.865	0.863	0.861	0.859	0.859	0.855
Sim 2	0.808	0.795	0.790	0.781	0.781	0.781	0.777	0.776	0.779	0.777	0.781	0.782	0.782	0.782
Sim 3	0.884	0.870	0.865	0.856	0.855	0.852	0.846	0.843	0.843	0.839	0.838	0.839	0.840	0.839
UnSim 1	0.417	0.417	0.416	0.415	0.414	0.414	0.414	0.413	0.413	0.411	0.411	0.413	0.411	0.410
UnSim 2	0.662	0.661	0.659	0.659	0.658	0.658	0.656	0.656	0.657	0.657	0.658	0.659	0.662	0.664
UnSim 3	0.518	0.518	0.516	0.515	0.515	0.516	0.514	0.516	0.515	0.516	0.518	0.518	0.517	0.519

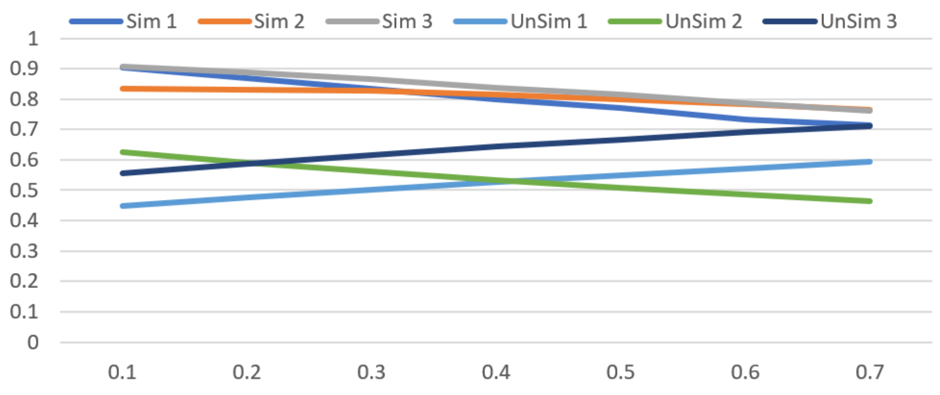


Figure 10: Resistance to single-side chaff-perturbation in local area network

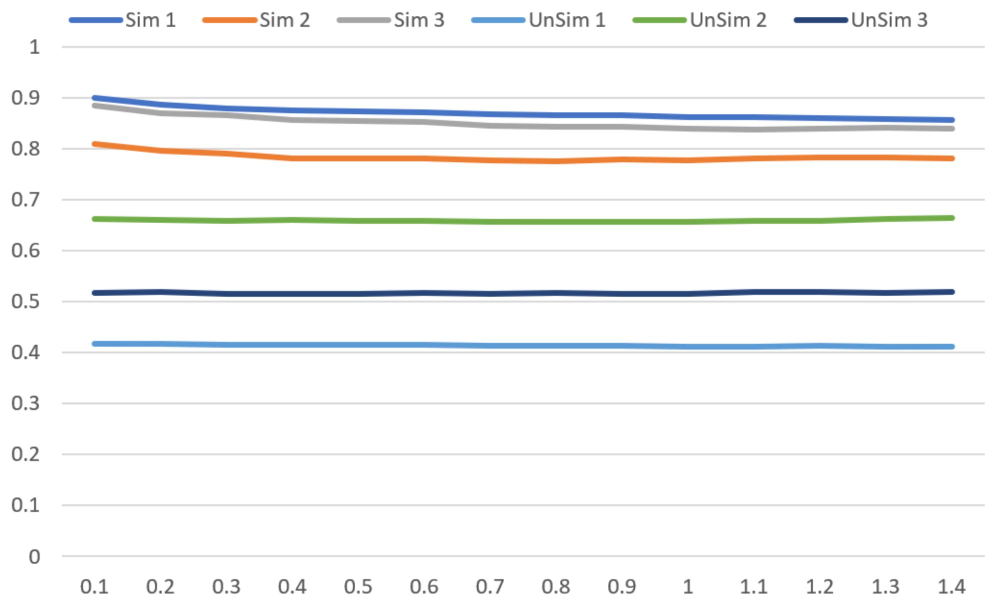


Figure 11: Resistance to both-side chaff-perturbation in local area network

6.3 Experimental Results and Its Analysis on the Internet

The resistance performance of the proposed algorithm was also assessed using the network traffic from the Internet. It has the similar network connection setup as in a local area network. The only difference is that the three victim hosts are AWS servers located in different states of the United States. The reasons that we assess the detection algorithm in the Internet are as follow. First, all the attackers launch their attacks via the Internet, other than a local area network. Second, a connection chain established in the Internet may cause a longer response delay than in a local area network. Third, a packet split, and/or packet merging occurred in the Internet may affect the detection performance of the proposed algorithm. We repeated the same process as we did in Section 6.1 to collect the incoming and outgoing network traffic from the sensor. The data sets collected were processed via the SSID algorithm, and the obtained results are shown in Table 5, Table 6, Table 7, Figure 12, and Figure 13, respectively.

Table 5: SSID Similarities on the Internet

Incoming traffic / Outgoing traffic	i1	i2	i3
o1	0.729038	0.185432	0.337353
o2	0.131143	0.896734	0.501174
o3	0.238080	0.539846	0.949879

From the results shown in Table 5, we see the proposed algorithm can detect stepping-stone intrusion under the context of the Internet, and obtain a similar performance as in a local area network. In Table 6, the results show the proposed algorithm presents better performance in the Internet than in a local area network in terms of resistance to intruders' chaff-perturbation manipulation. In Figure 12, it shows there is a crossing between Sim 2 and UnSim 3 when the single-side chaff rate reaching 80%. This indicates, under the context of the Internet, the proposed algorithm can resist intruders' manipulation up to 80% of single-side chaff rate which is a little higher than the one in the context of a local area network. However, when both sides of the sensor are chaffed, the results shown in Table 4 and Figure 13 presents a similar

Table 6: SSID similarities on the Internet under single-side chaff

Chaff Rate Similarity	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
Sim 1	0.736491	0.741859	0.746223	0.746883	0.743977	0.739247	0.730983	0.721077
Sim 2	0.871032	0.848337	0.820099	0.792302	0.764496	0.74029	0.715852	0.692721
Sim 3	0.922392	0.894375	0.862214	0.831892	0.801985	0.773674	0.746302	0.721515
UnSim 1	0.248555	0.259386	0.269311	0.279442	0.28987	0.299796	0.309645	0.319922
UnSim 2	0.507653	0.479227	0.453611	0.430719	0.411298	0.392751	0.375563	0.3603
UnSim 3	0.531461	0.556525	0.584458	0.607655	0.629783	0.651992	0.67061	0.684988

Table 7: SSID similarities on the Internet under both-side chaff

Chaff Rate Similarity	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4
Sim 1	0.706	0.697	0.692	0.688	0.686	0.683	0.683	0.681	0.680	0.679	0.679	0.678	0.680	0.679
Sim 2	0.807	0.782	0.769	0.773	0.760	0.756	0.758	0.756	0.757	0.760	0.764	0.765	0.764	0.759
Sim 3	0.910	0.897	0.889	0.884	0.881	0.878	0.877	0.872	0.871	0.868	0.864	0.859	0.863	0.864
UnSim 1	0.236	0.236	0.235	0.235	0.234	0.235	0.234	0.234	0.234	0.234	0.234	0.234	0.233	0.233
UnSim 2	0.503	0.502	0.500	0.502	0.502	0.502	0.505	0.507	0.508	0.513	0.516	0.518	0.519	0.519
UnSim 3	0.499	0.496	0.496	0.497	0.496	0.498	0.499	0.500	0.504	0.504	0.507	0.509	0.513	0.512

performance as in a local area network.

6.4 Comparison with Existing Detection Algorithms

As we discussed in the Introduction session, there were many methods proposed to detect stepping-stone intrusion. However, most approaches cannot resist intruders' chaff-perturbation attack. Some of them may resist chaff attack to a certain degree, but their performances are worse than the algorithm we proposed. Table 8 shows the comparison results between our algorithm and the algorithms recently developed.

Table 8: Comparison of Performance Results with Existing Detection Algorithms

Compared items Algorithms	Able to detect stepping-stone intrusion?	Resistant to chaff attacks? If Yes, up to what chaff-rates?
The algorithm proposed in this paper	Yes	Yes, up to 80%
Context-based TCP/IP packets matching, 2010	Yes	Yes, up to 20%
A dynamic programming technique, 2010	Yes	No
Packet mining approach, 2012	Yes	Yes, up to 40%
Random Walk approach, 2015	Yes	Yes, up to 40%
Packet cross-matching, 2016	Yes	Yes, up to 30%
K-Means data mining and clustering, 2020	Yes	Yes, up to 20%

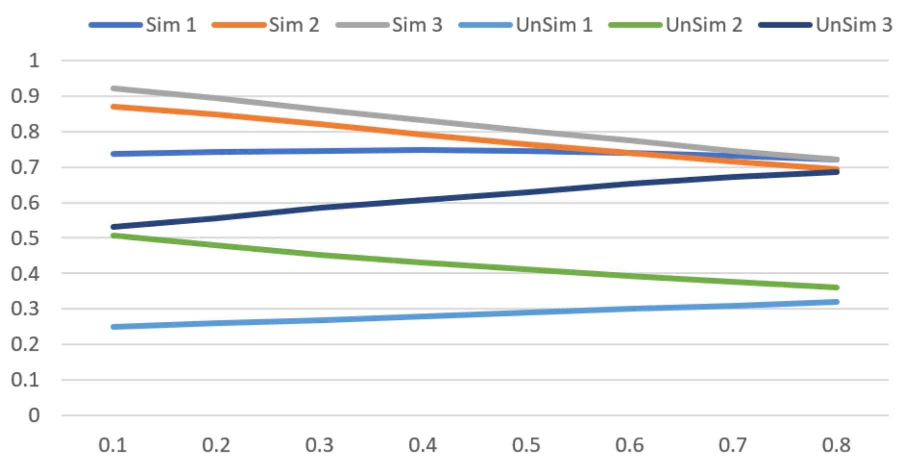


Figure 12: Resistance to single-side chaff-perturbation on the Internet

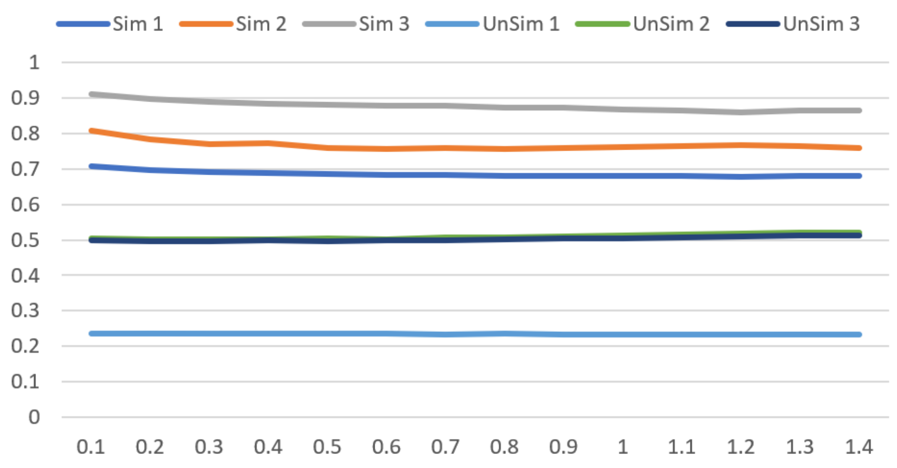


Figure 13: Resistance to both-side chaff-perturbation on the Internet

7 Conclusion and Future Work

Most intruders tend to launch their attacks through stepping-stones in order to avoid detection and capture. Since 1990s, there have been many algorithms developed to detect stepping-stone intrusions. Most of the methods proposed can be defeated by intruders' session manipulation, such as chaff-perturbation. In this paper, we proposed a novel approach by modelling computer network traffic and exploiting encrypted packets to detect stepping-stone intrusions. Network traffic can be modelled as a timestamp sequence of the length of Send and Echo packets. Stepping-stone intrusions can be identified by comparing the sequence of the incoming connection with the one of the outgoing connections of a sensor. The experimental results show that the proposed algorithm can not only be able to detect stepping-stone intrusion effectively, but also resist intruders' single-side chaff- perturbation up to 70% in a local area network, as well as 80% under the context of the Internet. More surprisingly, the proposed algorithm presented better performance in resisting intruders' both-side chaff-perturbation than in single-side. The experimental results also show, when the both sides of the sensor are chaffed, and the chaff rate was increased to 140%, the proposed algorithm was still not defeated. This indicates that the proposed algorithm presented a much stronger resistance performance in both-side than in single-side.

In the future, first, we will focus on the optimization of the sequence comparison algorithm. This will further improve the detection performance of the proposed algorithm. We will also explore more information hidden in encrypted network traffic to detect stepping-stone intrusion, and resist intruders' manipulation. Second, we will study under different encryption algorithms, how to detect stepping-stone intrusion by using the length of encrypted packets since the length may be changed.

Author Contribution

Dr. J. Yang did the algorithm design and experimental results analysis for this paper; Dr. L. Wang worked on supervising students to conduct experiments and collect data, and helped writing the paper; Mr. S. Shakya set up the network environment and conducted experiments to collect data.

Data Availability

The network traffic packets captured for this research can be found from the following shared Drive: `https://drive.google.com/drive/folders/1k763JnVBYK7OOHHplYX5JrSq2YO9jji_`

Conflicts of Interest

We, as the authors, declare no conflict of interest regarding the publication of this paper.

Acknowledgments

This work of Drs. Jianhua Yang and Lixin Wang is supported by the National Security Agency NCAE-C Research Grant (H98230-20-1-0293) with Columbus State University, Georgia, USA.

References

[1] J. Yang, L. Wang, S. Shakya, and M. Workman. Identify encrypted packets to detect stepping-stone intrusion. In *Proc. of the 35th International Conference on Advanced Information Networking and Applications (AINA'21), Toronto, Canada*, volume 226 of *Lecture Notes in Networks and Systems*, pages 536–547. Springer, Cham, May 2021.

[2] Y. Zhang and V. Paxson. Detecting stepping-stones. In *Proc. of the 9th USENIX Security Symposium (USENIX Security'00), Denver, CO, USA*, pages 67–81. USENIX Association, August 2000.

[3] S. Staniford-Chen and L. T. Heberlein. Holding intruders accountable on the internet. In *Proc. of the 1995 IEEE Symposium on Security and Privacy (IEEE S&P'95), Oakland, CA, USA*, pages 39–49. IEEE, May 1995.

[4] K. Yoda and H. Etoh. Finding connection chain for tracing intruders. In *Proc. of the the 6th European Symposium on Research in Computer Security (ESORICS'00), Toulouse, France*, volume 1985 of *Lecture Notes in Computer Science*, pages 191–205. Springer, Berlin, Heidelberg, October 2000.

[5] A. Blum, D. Song, and S. Venkataraman. Detection of interactive stepping-stones. algorithms and confidence bounds. In *Proc. of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID'04), Sophia Antipolis, France*, volume 3224 of *Lecture Notes in Computer Science*, pages 258–277. Springer, Berlin, Heidelberg, September 2004.

[6] X. Wang, D.S. Reeves, S.F. Wu, and J. Yu. Sleepy watermark tracing: An active network based intrusion response framework. In *Proc. of the 2001 IFIP International Information Security Conference (SEC'04), Paris, France*, volume 65 of *IFIP International Federation for Information Processing*, pages 369–384. Springer, Boston, MA, June 2004.

[7] X. Wang and D.S. Reeves. Robust correlation of encrypted attack traffic through stepping stones by manipulation of interpacket delays. In *Proc. of the 10th ACM Conference on Computer and Communications Security (ACM CCS'03), Washington, DC, USA*, pages 20—-29. ACM, October 2003.

[8] X. Wang. The loop fallacy and serialization in tracing intrusion connections through stepping stones. In *Proc. of the 2004 ACM symposium on Applied computing (ACM SAC'04), Washington, DC, USA*, pages 404—411. ACM, March 2004.

[9] T. He and L. Tong. Detecting encrypted stepping-stone connections. *IEEE Transaction on Signal Processing*, 55(5):1612–1623, 2007.

[10] D. L. Donoho. Detecting pairs of jittered interactive streams by exploiting maximum tolerable delay. In *Proc. of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID'02), Zurich, Switzerland*, volume 2516 of *Lecture Notes in Computer Science*, pages 17–35. Springer, Berlin, Heidelberg, October 2002.

[11] K. H. Yung. Detecting long connecting chains of interactive terminal sessions. In *Proc. of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID'02), Zurich, Switzerland*, volume 2516 of *Lecture Notes in Computer Science*, pages 1–16. Springer, Berlin, Heidelberg, October 2002.

[12] J. Yang and S.-H. S. Huang. A real-time algorithm to detect long connection chains of interactive terminal sessions. In *Proc. of the 3rd ACM International Conference on Information Security (Infosecu'04), Shanghai, China*, pages 198–203. ACM, November 2004.

[13] J. Yang and S. S.-H. Huang. Mining tcp/ip packets to detect stepping-stone intrusion. *Journal of Computers and Security*, 26:479–484, 2007.

[14] Y. Zhang, J. Yang, S. Bediga, and Stephen S.-H. Huang. Resist intruders' manipulation via context-based tcp/ip packet matching. In *Proc. of the 24th IEEE International Conference on Advanced Information Networking and Applications (AINA'10), Perth, Australia*. IEEE, April 2010.

[15] Y. Kuo, S. S. H. Huang, W. Ding, R. Kern, and J. Yang. Using dynamic programming techniques to detect multi-hop stepping-stone pairs in a connection chain. In *Proc. of the 24th IEEE International Conference on Advanced Information Networking and Applications (AINA'10), Perth, Australia*. IEEE, April 2010.

[16] Y. Sheng, Y. Zhang, and J. Yang. Mining network traffic efficiently to detect stepping-stone intrusion. In *Proc. of the 26th IEEE International Conference on Advanced Information Networking and Applications (AINA'12), Fukuoka, Japan*. IEEE, March 2012.

[17] J. Yang and Y. Zhang. Rtt-based random walk approach to detect stepping-stone intrusion. In *Proc. of the 29th IEEE International Conference on Advanced Information Networking and Applications (AINA'15), Gwangju, South Korea*, pages 558–563. IEEE, March 2015.

[18] J. Yang. Resistance to chaff attack through tcp/ip packet cross-matching and rtt-based random walk. In *Proc. of the 30th IEEE International Conference on Advanced Information Networking and Applications (AINA'16), Crans-Montana, Switzerland*, pages 784–789. IEEE, March 2016.

[19] L. Wang, J. Yang, X. Xu, and P.-J. Wan. Mining network traffic with the k-means clustering algorithm for stepping-stone intrusion detection. *Wireless communications and Mobile Computing*, 2021:6632671:1–6632671:9, March 2015.

[20] S. Wu, U. Manber, G. Myers, and W. Miller. An o(np) sequence comparison algorithm. *Information Processing Letters*, 35(6):317–323, September 1990.

Author Biography

Jianhua Yang is currently working at TSYS School of Computer Science, Columbus State University (CSU), Columbus, GA USA as a full Professor. He received his Ph.D. degree on Computer Science from University of Houston, USA at 2006, M.S. and B.S. degree on Electronic Engineering from Shandong University, China at 1987, and 1990 respectively. Dr. Yang has published more than 60 peer-reviewed journal papers and conference proceedings on cybersecurity. He has been awarded by NSA, NSF, and DoD with amount of more than half million dollars. His current research interest is computer network and information security.

Lixin Wang is currently an Associate Professor of computer science at Columbus State University, Columbus, GA USA. He holds a Ph.D. degree in Computer Science from Illinois Institute of Technology, Chicago IL. His research interests include Network Security, Intrusion Detection, Wireless Networks, Algorithm Design and Analysis. He has been conducting top quality research and published 47+ peer-reviewed high-quality research papers, most of which are published on leading Computer Science journals or top-tier Computer Science conferences. Since 2011, Dr. Wang has been awarded ten federal grants (from NSF, NSA, DoE, USAR, or DoEd of USA) as the PI or Co-PI with the total awarded amount more than $2.6 million USD.

Suhev Shakya is a student of computer science at Columbus State University, Columbus, GA, USA. He is projected to graduate with a bachelor's degree in computer science in 2022. He has worked in several research and commercial projects in fields such as cybersecurity, machine learning, bioinformatics, and video game development. His current research interest is Artificial Intelligence and its applications in security, automated systems, and video games.