Matching TCP Packets and Its Application to the Detection of Long Connection Chains on the Internet

Jianhua Yang and Shou-Hsuan Stephen Huang
Department of Computer Science, University of Houston
E-mail: {jhyang, shuang}@cs.uh.edu

Abstract

Network attackers usually use long connection chains to hide their identity. One way to stop such attacks is to prevent the attackers from using computers as "stepping-stones" for their attacks. We can determine the number of hosts in a connection chain by monitoring the packets sent from a host to the target host and their "Echo" packets. Due to many network protocol issues, it is not possible to match all such pairs correctly. An algorithm to match a "Send" packet with its corresponding "Echo" packet is presented in this paper. The Roundtrip Times of these matched packets can be used to determine the number of intermediate hosts. The algorithm gives us a way to stop such connections in real-time.

1. Introduction

The Internet has already been an important part of this world. Security is becoming more and more important along with widely use of the Internet in recent 10 years. Research in [13] showed that various attacks through the Internet have increased greatly. In the early times, attackers usually launch their attacks directly. But it is easy to expose the attackers themselves, instead of directly attacking, most attackers recently launch their attacks by using a technique called stepping-stone [1]. It is a kind of technique to launch attacks not from their own computers but from intermediary hosts that the attackers previously compromised. The compromised computers are called stepping-stones. The benefit of using stepping-stones to launch attack is that the attackers run little risk of detection. They can avoid their responsibilities for their criminals because of known spatial, political and cooperation reasons. Some attackers even erase their logs on the stepping-stone they chained. Even if logs remain on a particular host, the victims can only use it to trace back one link in the chain. We need some techniques not only to detect stepping-stones that are used by attackers, but also to detect them in real-time. These kinds of techniques are called stepping-stone detection.

Some approaches are proposed to detect stepping-stones in [1,2,3,12,14]. That is to detect whether the machines monitored are going through by some connection chains. Method in [2] is to identify intruders with connection chains by comparing the content of each connection. It is supposed to apply only on non-encrypt sessions, such as the connections established by using Telnet, or rlogin. Another method is proposed in [1] to detect stepping-stone on encrypt sessions. It is called time-based approach because what it used are only distinctive characteristics (such as packet size, time stamp, etc.) of interactive traffic, rather than using contents of a connection. It has the advantages of applying on encrypt sessions, not requiring tightly synchronized clocks, and being robust against retransmission variation. It still has some limitations. (1) It couldn’t distinguish between stepping-stones used by intruders and those used by legitimate users. (2) It only applies to detecting interactive stepping-stones, and may not apply to machine-driven attacks launched through a chain of stepping-stones. (3) It is easily to defeat this technique by manipulating the outgoing packets, such as randomly delay each packet, or randomly add some meaningless packets to the connection that is established by the attacker. One new approach is proposed in [14]. It could detect the attacker who tries to elude the detection by adding some noises to the connection used. Its main idea is to detect pairs of jittered interactive streams by exploiting maximum tolerable delay. The problem with this approach is that the connection chain needs to be kept long enough. The attackers can still evade detection by keeping the connections sufficiently short. The good news is that if the connections are kept in a short time, what the attacker does is limited. Kwong H. Yung [3] proposed one method to detect intruders in a long connection chain by echo-delay comparison. Yung’s technique [3] could give good results only when the network traffic is uniform. And this algorithm cannot be implemented in real time.

J. Yang and S. Huang proposed an algorithm to detect stepping-stone in real time [12], and also overcome the network fluctuations. It still has some limitations. First of all is that it is only available on campus network. What is used in [12] is the round trip time gap between two matched packets. That gap reflects how many hosts are
chained in some sense. It shows in [12] some results on local network, rather than on the Internet. We found if we apply the algorithm proposed in [12] to the Internet, it works incorrect. The communication on the Internet is more complicated than on the campus network because of network load, historical fluctuation, workload of the chained hosts.

In this paper, we proposed another algorithm to let the idea in [12] of detecting stepping-stone work on the Internet. The key issue of this algorithm is the packet matching. Some factors that affect the packet matching are [8,9,10,11]: (1) lost packet retransmission, (2) packet cumulative acknowledgement/echo, (3) session transmit window, (4) packets communication between hosts (such as ignore packets, keep alive packet sent from client side, key re-exchange, these data are not intended for the target machine), and (5) multiple Echo packets from server side.

Matching all the packets is impossible. We don’t need to match all the packets going through a TCP connection in a host for the purpose of detecting stepping-stone. So a matching condition is proposed to simplify TCP packet matching problem. Compare to Yung’s method [3], our algorithm actually attempts to match the packets accurately, and then decides the number of hosts connected downstream. Compare to the method [12], this algorithm can be applied to the Internet. This technique addresses most of the concerns mentioned above and produces more accurate result than the previous work [1, 2, 3, 12]. In Section 2, we discuss some challenges on packet matching. In Section 3, the packet matching algorithm is described in details. The proof of the packet matching lemma is given in Section 4. In Section 5, step function method to detect number of hosts is discussed. Some experimental results done on Internet are given in Section 6. Finally, in Section 7, conclusions and future work will be mentioned.

2. Challenges of Packet Matching

Matching a Send packet and its corresponding Echo packet is more difficult than we originally thought. We will explain the challenges of the matching in this section.

Suppose a user logs in at Host 1, and connects to Host n through Host 2, … , and Host n-1. Host k is the node that we could put our program to capture the packets going through it. We monitor one port of Host k that connects port 22 of Host k+l. If there is a packet sent from Host k to Host k+l, most probably, this packet is going to be acknowledged by Host k+l first, and then echoed by Host k+l. Suppose we call the packet sent from Host k a ‘Send’ packet, the acknowledge packet from Host k+l a ‘Ack’ packet, and the packet echoed from Host k+l an ‘Echo’ packet. In a simple case with only these Send-Ack-Echo packets, the Send packet and the Echo packet would form a pair. The difference of their timestamps thus represents the roundtrip time. If there are only these three packets, it is very easy to match them. Unfortunately, most cases are much more complicated than that.

If a packet is sent from Host k to Host n, there are many hosts in between, how to match the packet sent from Host k and echoed from Host n? Some times, even if just match the packet sent from Host k to Host k+l, it is not easy thing because of complex of TCP protocol. We really face some challenges when we try to match TCP packets going through one host in a long connection chain.

To understand the challenges we face, we first explain the TCP protocol [9] and OpenSSH architecture briefly. OpenSSH is tool that uses SSH as a protocol for secured remote login and other secure network services over an insecure network [8]. Most hackers like to use OpenSSH to connect several hosts to form a chain. There are three-stage processes of data transport: synchronization, data transfer, connection closed. What we are interesting is the phrase of data transfer. Flow control and congestion control are used to guarantee that the data transfer is reliable. There are several challenges making data transfer complicated. They are [8,9,10,11]: (1) lost packet retransmission, (2) packet cumulative acknowledgement/echo, (3) session transmit window. (4) packet communication between hosts (such as ignore packets, keep alive packet sent from client side, key re-exchange, these data are not intended for the target machine), and (5) multiple Echo packets from server side.

Any lost packets during transmission are retransmitted either automatically by the sending client having not received an acknowledgement or on request of the receiving server. Retransmission of the same packet continues until either an acknowledgement is received or until the connection timeout expires. So we are faced with one Echo packet that could match with two or more Send packets.

Not every TCP packet is individually acknowledged. Instead, cumulative acknowledgement may take place. It has several advantages; one of the most important of them is that it reduces the number of Ack messages, thereby reducing the possibility of network congestion. This network control mechanism benefits the network traffic, but makes one-to-one packet matching impossible. The same
problem occurs for Echo packets too.

For data flow control and congestion control, TCP maintains a transmit window. The size of the window determines how many unacknowledged octets of data the transmitter is allowed to send before it must cease transmission and wait for acknowledgement. Thereby if this size is set to one, it means that each packet is sent if and only if the previous Send packet is acknowledged or echoed. Most probably this size is not one, so several packets can be allowed to send continuously without receiving any Ack packet. So several Send-Ack-Echo overlaps each other making packet matching difficult.

Ignore packet is a very special type packet. The only usage of this kind of packet is to be used as an additional protection measure against advanced traffic analysis techniques [10,11]. If the server side receives an Ignore packet, it only acknowledges this packet without any other action. If we do not process the ignore packet well, it will affect all the packets matching after the Ignore packet.

Keep-alive and Key re-exchange are the packets that are different from the previous cases. They do not affect the packet matching, but the packets matched in these two cases are not what we expect because the packets are only sent to the neighbor host not the connection destination host. For most of time of the session, the key used for encryption is not changed, but it may be changed during the data transfer. It is recommended [11] that the key been changed and exchanged after each gigabyte of transmitted data or after each hour of connection time, whichever comes sooner. In this case, there would be an extra packet sent to the neighbor host, rather than the destination host. Thereby there is an echoed packet coming from downstream neighbor host, and also there is a packet pair in which the packet is matched. This matched packet is not what we want because it only indicates the RTT (Round trip time) to the nearest neighboring host, not a round-trip to the target machine. How to filter out this kind of packet is another challenge we face in our packet matching algorithm.

When a command is executed at the target host, the result may be sent back in several packets. This also complicated the packet matching. We will discuss how to process it in Section 3.

3. TCP packet matching algorithm

Though we stated many challenges in matching some packets, there are plenty of packets going through the chain of connection. We don’t have to match 100% of these packets to detect a new connection in the chain. If we can match a majority of the packets, it is sufficient for most practical purposes. There is a tradeoff that we have to balance carefully. We can try to match as much as we can. But the penalty of just one mismatch is very high; all subsequent pairs are mismatched. So we decided to be more conservative in our algorithm presented in this section. We collect only the matches that we are truly confident about their correctness and sacrifices on the matching rate.

The matching algorithm is not straightforward because of the following reasons. (i) The Send and Echo packets does not form a one-to-one relationship. It may be one-to-many or many-to-one. (ii) The intervals of Send-Echo pairs sometimes overlap each other. In the following paragraphs, we will discuss how to design our algorithm to avoid the problems mentioned in Section 2.

In this case of a failed Send packet, the Echo packet should match with the Resend packet, rather than the failed Send. Here is one typical scenario:

```
Send P1
Ack P1
Send P2
Send P3
Echo Packet Q1
... 
Resend P2
Resend P3
Echo Packet Q2
```

The Echo Packet Q1 clearly should match P1, not P2, P3. The Echo Packet Q2 should match one of Resend packets, rather than Send P2, or P3. The problems we are facing are (i) we cannot distinguish Send from Resend, and (ii) it is difficult to decide which packet is the resent packet of which one. In this case, all the resend would satisfy the following “sequence number conditions”:

\[
\text{Echo.seq} = \text{Send.ack} \quad (1a) \\
\text{Echo.ack} > \text{Send.seq} \quad (1b)
\]

In such a situation, we will simply match Q2 with the first resend. Conditions (1a) and (1b) are to guarantee there is no Echo between the Send and the Echo. We call this condition packet matching condition. For Cumulative Packet

![Figure 2. Relationships among RTT, PDT](image-url)
Echo, several Send packets are echoed by one echo packet, so it is difficult to know which Send packet is matched by the Echo packet. The complexity is not only limited to this point, we sometimes don’t know which packets are cumulatively echoed. Here is a typical scenario:

Send P1  
Ack P1  
Send P2  
Ack P2  
Send P3  
Ack P3  
Echo Packet Q1  
Echo Packet Q2

We are faced with the task of matching two Echo packets with the three Send packets in this case. Maybe Packet Q1 would match P1, and the rest Send packets are cumulatively echoed by Echo Packet Q2. It is also possible that P1, P2 are cumulatively echoed by Packet Q1, and Send packet P3 is matched by Echo Packet Q2. Because we don’t have enough information to make a decision, we will only allow Q1 to match with P1. Note Conditions (1a) and (1b) hold in this case too.

If the transmission window size is not one, there are going to be many packets sent without being acknowledged first. So there will be a lot more Resends and Cumulative Echo that will make our analysis more complicated.

For multiple echo packets, we would ignore all the echo packets except the first one. Most likely there is no more Send to be matched with the additional Echo.

For protocol control packet (keep-alive, ignore, and key re-exchange) sent to the neighbor host, we can match them easily because the response is normally very fast due to the short hop. The problem is this match does not represent a RTT to the target host. We decide not to do anything in the main algorithm and try to filter out this type of match after the matching algorithm is done.

Before we present the algorithm, let’s look at a very simple example of packet exchange. Figure 2 illustrate data transfer between two hosts, one is a client, the other is as server.

We define RTT (Round trip time) as the time interval between the time a packet is sent and the time a corresponding echoed packet is received, PDT (Propagation delay time) as the time interval between time the packet is sent from the client and time the packet is received by the server, RDT (Response delay time) as the time interval between time the packet is received by the server and the time acknowledged packet is sent to the client, and HPT (Host processing delay) as the time the host need to process that packet.

Most probably PDT is relative stable (with about 10% of variation) even if there is a little bit fluctuation on the network. However RDT + HPT may vary depending on the packet received. The content of the packet may have to be sent down the chain. If we know exactly how to match Send and Echo packets, it is straightforward to compute RTT. We know that there is no guarantee of a one-to-one correspondence of Send and Echo packets. If there is such one-to-one correspondence, then matching becomes trivial. For every Echo packet, match it with the first unmatched Send packet. Indeed, in our experiments, there are many such cases.

Table 1 below shows a case where the trivial algorithm failed. In this case, we have to deliberately create the data that causes this undesirable situation to occur. Even though this is unlikely to happen, it does serve the purpose of illustration. Column (a) indicates the type of the packet captured. Column (b) shows the direction of the packet. Column (c) shows the time stamp with date/hour/minutes truncated. Column (d) is the length of the packet. Column (e) is the TCP flag. Columns (f) and (g) are the sequence number and acknowledgement number with the leading 4 digits truncated.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send</td>
<td>51.426275</td>
<td>134</td>
<td>PA</td>
<td>f0ba</td>
<td>db95</td>
<td></td>
</tr>
<tr>
<td>Ack</td>
<td>&lt; 51.426671</td>
<td>60</td>
<td>A</td>
<td>db95</td>
<td>F10a</td>
<td></td>
</tr>
<tr>
<td>Send</td>
<td>51.435684</td>
<td>134</td>
<td>PA</td>
<td>f10a</td>
<td>db95</td>
<td></td>
</tr>
<tr>
<td>Ack</td>
<td>&lt; 51.436083</td>
<td>60</td>
<td>A</td>
<td>db95</td>
<td>F15a</td>
<td></td>
</tr>
<tr>
<td>Send</td>
<td>51.440245</td>
<td>134</td>
<td>PA</td>
<td>f15a</td>
<td>db95</td>
<td></td>
</tr>
<tr>
<td>Ack</td>
<td>&lt; 51.440566</td>
<td>60</td>
<td>A</td>
<td>db95</td>
<td>F1aa</td>
<td></td>
</tr>
<tr>
<td>Send</td>
<td>51.445048</td>
<td>134</td>
<td>PA</td>
<td>f1aa</td>
<td>db95</td>
<td></td>
</tr>
<tr>
<td>Ack</td>
<td>&lt; 51.445377</td>
<td>60</td>
<td>A</td>
<td>db95</td>
<td>f1fa</td>
<td></td>
</tr>
<tr>
<td>Send</td>
<td>51.449840</td>
<td>134</td>
<td>PA</td>
<td>f1fa</td>
<td>db95</td>
<td></td>
</tr>
<tr>
<td>Ack</td>
<td>&lt; 51.450167</td>
<td>60</td>
<td>A</td>
<td>db95</td>
<td>F24a</td>
<td></td>
</tr>
<tr>
<td>Send</td>
<td>51.454649</td>
<td>134</td>
<td>PA</td>
<td>f24a</td>
<td>db95</td>
<td></td>
</tr>
<tr>
<td>Ack</td>
<td>&lt; 51.454973</td>
<td>60</td>
<td>A</td>
<td>db95</td>
<td>F29a</td>
<td></td>
</tr>
<tr>
<td>Send</td>
<td>51.459566</td>
<td>134</td>
<td>PA</td>
<td>f29a</td>
<td>db95</td>
<td></td>
</tr>
<tr>
<td>Ack</td>
<td>&lt; 51.459901</td>
<td>60</td>
<td>A</td>
<td>db95</td>
<td>F2ea</td>
<td></td>
</tr>
<tr>
<td>Echo</td>
<td>51.678849</td>
<td>102</td>
<td>PA</td>
<td>db95</td>
<td>F2ea</td>
<td></td>
</tr>
</tbody>
</table>

---RTT=252574---

| Echo| < 51.722553 | 102 | PA| dbc5 | F2ea |

---RTT=286869---

| Ack | > 51.722647 | 54  | A | f2ea | dbf5 |

In this case, seven letters were sent in seven Send packets, but only two Echo packets returned. The trivial algorithm will match the two Echo packets with the first two Send packets. The first cumulative Echo probably matches with the first few Send packets; and the second Echo with the remaining Send packets. However, we do not know where the cut-off is. As a result, the second RTT computed (using the second Send) is slightly higher than the first RTT value. It is better to match the second Echo with the last Send. Even worse, this will leave five unmatched Send packets to be used later. All of the subsequent matches will be inaccurate. The example demonstrates the difficulty in matching the packets.
Our algorithm (see Algorithm 1 below) uses a queue to save the Send packets. When an Echo is received, it is used to match with one of the Send (not necessarily the first one). The algorithm may also purge some Send and/or Echo packets as necessary. As we stated before, when we have doubted about the packets, we purge them. We impose the conditions (1a) and (1b) on the packets. Most likely the condition is too strong and we are not taking advantages of all Echo packets. We shall discuss how the algorithm deals with the problems mentioned in the previous section. For Resend, our algorithm will match with the first Resend packet that satisfies (1a) and (1b). For cumulative Echo/Ack, our algorithm will match the Echo with the first Send and ignores the remaining Send and Echo packets, this may result in a slightly higher RTT. If the target host sends back multiple Echoes, the algorithms will ignore all but the first Echo. The only problem the algorithm does not deal with directly is the issue of communication with the next host (such as key exchange, keep-alive, ignore-me). We should add another algorithm (not included in Algorithm 1 to keep it simple) to filter out this type of RTT.

When this happens we will see a significant drop in RTT, and the value is very close to RTT to the next host.

Another feature that was not shown in the algorithm is the division of packet sequence into small bursts. In most

```
Initialize a SendQ queue;
while (there are more packets) {
    Capture the next packet P;
    // Identify packet type
    if P is a Send packet{
        add P to SendQ;
    } else if P is an Ack packet{
        // Ignore it
    } else if P is an Echo packet{
        boolean matched = false;
        while ((not matched) and (not empty(SendQ))) {
            Q = dequeue (SendQ);
            if ((Q.ack# = P.seq#) and (Q.seq# < P.ack#)){
                Packets P and Q are matched;
                Compute round-trip time between P and Q;
                matched = true;
            } else {
                // No match, get the next send packet
            }
        } // No match, get the next send packet
    }
}
```

**Algorithm 1. Packet Matching Algorithm**
interactive sessions, users are likely to pause between commands. The time gap is generally an order higher than the RTT. Thus we can take advantage of the time gap to isolate the Echo/Send matching. This will help in reducing the mismatches. Any mismatch will cause the subsequent match to be incorrect. So we added the extra precaution.

4. Proof of Packet Matching Condition

The following lemma gives a complete condition for packet matching.

**Lemma:** Let A and B be client and server respectively. If there is a TCP connection C from A to B, Send is a packet in C from A to B, and Echo is a packet in C Send from B to A, we say Send and Echo are matched if and only if (1a) and (1b) are satisfied.

\[
\begin{align*}
\text{Echo.seq} &= \text{Send.ack} \quad (1a) \\
\text{Echo.ack} &> \text{Send.seq} \quad (1b)
\end{align*}
\]

We need to prove two parts: (1) if two packets Send and Echo in C is matched, (1a) and (1b) hold, (2) if (1a) and (1b) hold, packets Send and Echo in C must match.

**Proof.** Once the TCP connection is established, suppose A is client side and B is server side, there is a TCP connection from A to B, and there is a send sequence number and receive sequence number. The initial send sequence number (ISS) is chosen by the data sending TCP, and the initial receive sequence number (IRS) is learned during the connection establishing procedure [9], suppose they are \(s_1\), \(a_1\) respectively.

In data communication, the sender of data keeps track of the next sequence number to use in the variable SND.NXT. The receiver of data keeps track of the next sequence number to expect in the variable RCV.NXT. The sender of data keeps track of the oldest unacknowledged sequence number in the variable SND.UNA. If the data flow is momentarily idle and all data sent has been acknowledged then the three variables will be equal. When the sender creates a segment and transmits it the sender advances SND.NXT. When the receiver accepts a segment it advances RCV.NXT and sends an acknowledgment. When the data sender receives an acknowledgment it advances SND.UNA. The extent to which the values of these variables differ is a measure of the delay in the communication [9].

(1) If packets matched, the matching condition must hold.

If a packet with length \(n_1\) is sent from A side to B side, it uses \(s_1\) to stand for sender side unacknowledged sequence number, and \(a_1\) is used to indicate the receiver side unacknowledged number. We denote this as

\[
\begin{align*}
\text{Send} &\quad s_1 \quad a_1 \\
\text{Ack} &\quad a_1 \quad s_2 \quad (2a) \\
\text{Echo} &\quad a_1 \quad s_2 \quad (2c)
\end{align*}
\]

If B side acknowledges the packet sent from A side, it needs to point out the sender side (B side at this time) unacknowledged sequence number, and the receiver side (A side at this time) next sequence number. We denote this as

\[
\begin{align*}
\text{Ack} &\quad a_1 \quad s_2 \quad (2b)
\end{align*}
\]

Where \(s_2 = s_1 + n_1 + 1\).

B side sequence number \(a_1\) does not changed until the echo packet from B side is acknowledged in A side. We denote the echo action as

\[
\begin{align*}
\text{Echo} &\quad a_1 \quad s_2 \quad (2c)
\end{align*}
\]

This echo packet is going to be acknowledged by pointing out two numbers: one is next sequence number \(s_2\), another is next expected sequence number in server side (B side at this time). We denote this as

\[
\begin{align*}
\text{Ack} &\quad s_2 \quad a_2 \quad (2d)
\end{align*}
\]

Where \(a_2 = a_1 + n_2\), and \(n_2\) the packet size received by A side. The condition \(n_2 = n_1\) holds unless there is another send packet in between.

We say (2c) matches with (2a) because the packet echoed in (2c) corresponds to the packet sent in (2a), and condition (1a), (1b) holds.

This is a very basic and simple case in data communication. We call it one-one to be case. That is one send and one echo case. The other three general cases are one-many, many-one, and many-many. Let us prove that condition (1a), (1b) still holds for general cases.

We express one-many case as following

\[
\begin{align*}
\text{Send} &\quad s_1 \quad a_1 \quad (3a) \\
\text{Ack} &\quad a_1 \quad s_2 \quad (3b) \\
\text{Echo} &\quad a_1 \quad s_2 \quad (3c) \\
\text{Echo} &\quad a_2 \quad s_3 \quad (3e) \\
\text{Ack} &\quad s_3 \quad a_3 \quad (3f)
\end{align*}
\]

Where \(s_i = s_{i-1} + n_i\), and \(a_i = a_{i-1} + m_i\). It is obvious that (3a) and (3c), (3a) and (3e) match. But only the first match is useful for calculating RTT. We neglect the second match. The condition (1a), and (1b) holds for this case.

Many-one case is as following

\[
\begin{align*}
\text{Send} &\quad s_1 \quad a_1 \quad (4a) \\
\text{Ack} &\quad a_1 \quad s_2 \quad (4b) \\
\text{Send} &\quad s_2 \quad a_1 \quad (4c) \\
\text{Ack} &\quad a_1 \quad s_3 \quad (4d) \\
\text{Send} &\quad s_3 \quad a_1 \quad (4e) \\
\text{Ack} &\quad a_1 \quad s_4 \quad (4f) \\
\text{Echo} &\quad a_1 \quad s_4 \quad (4g)
\end{align*}
\]

In this case, we know there is a probability that (4g) matches all the three Send packets. It is still possible that (4g) matches (4a), and (4c) or only (4a). Whatever (4g) matches, there is one point we must be aware is that (4g) must match (4a) anyway. The condition (1a), and (1b) holds.

Many-many case is as following

\[
\begin{align*}
\text{Send} &\quad s_1 \quad a_1 \quad (5a)
\end{align*}
\]
If the matching condition holds, the pair must match matches (5a) and the matching condition holds.

(2) If the matching condition holds, the pair must match

If the matching condition holds, that is

\[
\text{Send.ack} = \text{Echo.seq} \quad (1a) \\
\text{Send.seq} < \text{Echo.ack} \quad (1b)
\]

Condition (1a) indicates that the receiver’s sequence number doesn’t change. This also means that this echo is the first echo packet for all the previous send packets. This echo must match one or some of the send packets from the first send packet. It must match with the first send packet. Second condition (1b) indicates that the echo packet is not empty.

5. Apply Packet Matching Algorithm to Detecting Long Connection Chain

To achieve the real time objectives, our algorithm has to use the information collected from the packets as soon as possible. If we can match the packets going through Host i, such as in Figure 3, the number of intermediate hosts can then be estimated by RTT distribution.

Based on our analysis before, for each intermediate host, the cost consists of two parts: (1) the network transmission time PDT, and (2) the host processing time RDT + HPT, which include the encryption/decryption time, cost of going up/down the network protocol hierarch, computer system delay. As Figure 2 shows, we use the time the Echo packet is received to define RTT, rather than use the time the Ack packet is received, the reason is that latter one can not represent the whole network downstream round trip time, because the Ack packet always comes from the neighbor host whatever how many hosts connected downstream. In [3], RTT based on Ack packet is used to estimate round trip time to Host i+1 is not accurate because HPT is ignored. The author tried to compensate for this by (i) taking the maximum of all such acknowledgement time, and (ii) multiplying the time by a factor of 2. Essentially, the paper is making an assumption that the processing time is approximately the same as the network transmission time. Thus the result presented in [3] is still valid to a degree. Even though we can explain why a factor of 2 works well, we don’t like this approach. What we use is step function method.

If we start monitoring the packet transmission from the beginning that the chain is established through Host i, we should see an increase of the RTT as the user connects to more and more hosts (see Figure 4 below). In other words, we use the changes of the RTT to signal the change in the connection chain. If we monitor the chain continuously, we should get a step function with each step corresponds to one host connection. What we need to do is just count how many steps what we have, we could exactly know how many hosts connected in the downstream chain.

Unfortunately, there is a problem with this step function method on detecting long connection chain. The drawback is that if the two neighbor hosts connected in one chain are located in a LAN, the step is not apparent. It is very difficult to judge if there is a step. The good news is that we do not need to take care of this case, because if two hosts or more are connected by one chain, it is easy to detect them by other methods because they are located in a local area network. So for most of hackers, they would not do this connection.

Our algorithm is designed to run in real time, so how to count how many steps we have online is still necessary. All we needed to do is to detect when the “jump” (up or down) happened in the roundtrip time array. The up-jump represents an additional connection in the (downstream) chain.

We designed an algorithm to detect “jumps” in the packet roundtrip array found in algorithm 1. The algorithm can be used in real-time as the values in the array are filled. It only examines the last 2*w elements in determine whether there is a jump. Intuitively, we split the 2*w elements into two windows (left and right) of size w each. Within the windows, we select the minimum of the w values. This is to eliminate the network fluctuations. If the difference between the two minima exceeds a threshold, we declare there is a jump between the left and right windows. For details please refer to Algorithm 2 in [12]. If we accumulate all the jumps, we can easily determine the number of intermediate hosts. This estimate can then be used to terminate an OpenSSH session if the chain exceeds a predefined size.

6. Experiment results and Analysis

We implemented our algorithms using Libpcap [4,5,6, 7] to capture the packet for us similar to Snort. We conducted experiments to verify the effectiveness of the algorithms. Results of the experiment are presented here.

In our experiment we connect a PC to five hosts in a sequence. The following is a typical setup. We have varied the setup and included other hosts, the results are con-
sistent with the one presented here. Dannis is a Windows system and the rest are either Unix or Linux.

Dannis (Houston, TX) \rightarrow Aol08 (Houston, TX) \rightarrow Mex (Mexico) \rightarrow Bayou (Houston, TX) \rightarrow Mex (Mexico) \rightarrow Themis (Houston, TX)

By looking at the RTTs computed in the program, we can easily see that the matching algorithm worked. Figure 4 below shows a sample trace of the RTTs as we logged on to more and more hosts. Once the chain is established as the above shows, we simulate one hacker to use site Dannis to attack host Themis, other machines except Dannis and Themis are used as stepping stone. We tested Algorithm 1 on Aol08 for about 10 minutes. We capture the packets of outgoing and incoming connections to/from Aol08. We count the number of packets sent from Aol08 to Mex port 22, the number of packets matched with our matching algorithm, and how many packets are discarded. Table 2 summarizes the result.

From Table 2, there are up to 40% of the Send packets discarded. The discarded rate depends on the length of the chain and the user’s typing pattern. If the chain is long, the percentage of discarded packets should be a little bit bigger than 40%. Otherwise, if the chain is short, that number may be much smaller than 40%. The rate is an over-estimation because we count all the packets including resent packets and cumulative Echo packets.

**Table 2. Result of verifying Algorithm 1**

<table>
<thead>
<tr>
<th>Number of Trials</th>
<th>Total packets sent from Aol08</th>
<th>Matched packets in Aol08</th>
<th>Percentage of Packets discarded in Aol08 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1557</td>
<td>1036</td>
<td>33.5</td>
</tr>
<tr>
<td>2</td>
<td>1236</td>
<td>832</td>
<td>32.7</td>
</tr>
<tr>
<td>3</td>
<td>1398</td>
<td>923</td>
<td>33.9</td>
</tr>
<tr>
<td>4</td>
<td>1146</td>
<td>1011</td>
<td>11.8</td>
</tr>
<tr>
<td>5</td>
<td>1327</td>
<td>822</td>
<td>38.1</td>
</tr>
<tr>
<td>6</td>
<td>1092</td>
<td>936</td>
<td>14.3</td>
</tr>
<tr>
<td>7</td>
<td>1267</td>
<td>1013</td>
<td>20.0</td>
</tr>
<tr>
<td>8</td>
<td>1550</td>
<td>1466</td>
<td>5.4</td>
</tr>
<tr>
<td>9</td>
<td>1448</td>
<td>923</td>
<td>36.3</td>
</tr>
<tr>
<td>10</td>
<td>1233</td>
<td>841</td>
<td>31.8</td>
</tr>
</tbody>
</table>

We have to admit the conditions (1a) and (1b) are stronger than necessary. Take the example given in Table 3 as an example. There should be no doubt the two pairs of Send and Echo match. But the two conditions prevent the second pair to match in the algorithm. This also contributes to the high rate of discarded packets.

In another experiment, we use Algorithm 1 to match the packets and compute the RTT and then use Algorithm 2 to determine the number of hosts in the connection chain. Figure 4 can easily convince us that we can indeed determine the length of the connection chain.

**Table 3. An example for packet matching**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send</td>
<td>&gt;</td>
<td>51.637581</td>
<td>134</td>
<td>PA</td>
<td>d499</td>
<td>4830</td>
</tr>
<tr>
<td>Ack</td>
<td>&lt;</td>
<td>51.785836</td>
<td>60</td>
<td>A</td>
<td>d4e9</td>
<td>4830</td>
</tr>
<tr>
<td>Send</td>
<td>&gt;</td>
<td>51.785894</td>
<td>134</td>
<td>PA</td>
<td>d4e9</td>
<td>4830</td>
</tr>
<tr>
<td>Echo</td>
<td>&lt;</td>
<td>51.864846</td>
<td>102</td>
<td>PA</td>
<td>4830</td>
<td>d539</td>
</tr>
<tr>
<td>Ack</td>
<td>&gt;</td>
<td>52.010147</td>
<td>54</td>
<td>A</td>
<td>d539</td>
<td>4860</td>
</tr>
<tr>
<td>Echo</td>
<td>&lt;</td>
<td>52.059129</td>
<td>102</td>
<td>PA</td>
<td>4860</td>
<td>d539</td>
</tr>
</tbody>
</table>

7. Conclusion

We have designed an algorithm to match TCP packets in order to compute the Roundtrip Time between a host and the target host downstream. By observing the changes in the RTTs, we can determine the number of hosts in the connection chain. This real-time technique can be used to stop network intrusions while the intruder is connected to the target machine.

Our approach is different from the previous methods [2, 3]. It has the advantages of (i) the ability to detect intruders in real-time, (ii) the connection may be encrypted, (iii) the estimation of the downstream chain length is more accurate, (iv) the ability to tolerate the network traffic fluctuation, network load, workload of the chained hosts. Our algorithm does not use one particular connection as a yardstick to measure against the total length.

There are some limitations and restrictions. We must be able to monitor the packet throughout a connection session in order for this approach to work. If the fluctuation of a connection is higher than the additional time to connect to the next host, we will need a better algorithm to detect an
additional host. We are currently working on (i) relaxing Conditions (1a) and (1b) to match more pairs, (ii) studying the fluctuation of RTT, and (iii) determining if some of the techniques can be used to estimate the connection chain upstream.

Acknowledgements. We would like to thank Mr. Chandler Wilkerson, System Administrator in Computer Science Department, the University of Houston, for some helpful discussions and suggestions.

References