Matching TCP/IP Packets to Detect Stepping-Stone Intrusion

Jianhua Yang†, and Shou-Hsuan Stephen Huang††

†The Department of Mathematics and Computer Science, Bennett College, Greensboro, NC 25409, U.S.A.
††The Department of Computer Science, University of Houston, Houston, 77204 U.S.A.

Summary
We propose a "Step-Function" method to detect network attackers from using a long connection chain to hide their identities when they launch attacks. The objective of the method is to estimate the length of a connection chain based on the changes in packet round trip times. The key point to compute the round trip time of a connection chain is to match a Send and its corresponding Echo packet. We propose a conservative and a greedy matching algorithm to match TCP/IP packets in real-time. The first algorithm matches fewer packets but the quality of the matching is high. The second one matches more packets with some uncertainty on the correctness. The two algorithms give us almost identical results in determining the length of a long connection chain.

Key words:
Intrusion detection, Stepping-stone, Packet-matching, Conservative algorithm, Greedy algorithm

1. Introduction

Computer and network security has become more and more important as people depend on the Internet to conduct business, and the number of Internet attacks has increased significantly [13]. Instead of attacking a computer directly, most attackers launch their attacks through intermediary hosts that they have previously compromised [1] to hide themselves; those compromised computers are called stepping-stones. One way to stop such attacks is to use stepping-stone intrusion detection techniques, which has been being developed since 1995.

The first approach proposed in 1995 by Staniford-Chen and Heberlein [2] used the thumbprint to detect stepping-stone intrusion; this is the summary of the content of a connection. By comparing the thumbprints of two connections, this approach can determine if a given computer is being used as a stepping-stone. With the use of the secured shell, however, the content could be encrypted, thus rendering the thumbprint approach not applicable. Zhang and Paxson [1] then proposed a time-based method to detect stepping-stone intrusion on interactive sessions which used distinctive characteristics, such as packet size and timestamps to identify a connection. This method can be applied to encrypted sessions, and has the advantages of not requiring tightly synchronized clocks, and being robust against re-transmission variation. However, it suffers from a high false positive rate, is not available in real time, and is vulnerable to intruder manipulation, such as random delay and chaff perturbation. Another approach, similar to Zhang and Paxson's method, was proposed in 2000 to detect stepping-stone intrusion by computing the deviation between two connections [15]. However, it suffers from the same problem as Zhang and Paxson's method.

Yung [3] proposed a method which is time-based and can be applied to encrypted sessions; it detects stepping-stone intrusion in a long interactive connection chain by echo-delay round trip time (RTT) comparison. However, Yung's technique can give good results only when network traffic is relatively uniform, and his algorithm cannot be implemented in real time. The approach tries to decrease the false positive rate by using a statistical mathematical method to get the minimum echo RTT of a whole connection chain and maximum acknowledgement RTT of the connection to the downstream neighboring host. But it suffers from a high false negative rate, because the method used to match Send and Echo packets sometimes gives us an incorrect match, thus computing an incorrect RTT.

In this paper, firstly, we propose a "Step-Function" approach to detect stepping-stone intrusion in real time. This approach uses the changing RTTs between matched packets to estimate the length of a connection chain. The idea of using the changes of RTTs to signal the compromised hosts is demonstrated with experimental results from the Internet. To compute the correct RTT of a packet, we need to match a Send packet with its corresponding Echo packet. Thus, we proposed two packet-matching algorithms, a conservative one and a greedy one, to match TCP Send and Echo packets. The conservative packet matching algorithm can match a packet precisely but only for a small subset of the packets; the greedy algorithm can match more packets, but with some RTTs whose correctness we are unsure of. We prove

Manuscript received October 15, 2006.
Manuscript revised October 25, 2006.
.

that all the matched packets given by the conservative algorithm are correct matches, and then show that the results of the two algorithms are essentially the same for the purpose of determining the length of a connection chain. We tested these two algorithms on the Internet to detect long interactive connection chains with satisfactory results. Finally, we propose an algorithm to count the number of levels of RTTs of a connection chain. Compared to Yung's method [3], our approach estimates the length of a connection chain more precisely. We have successfully generalized the results of [12], addressed most of the concerns mentioned above, and produced more accurate results with a zero false positive rate and a lower false negative rate than the previous work [1, 2, 3, 12].

Matching all Send-Echo packets correctly is impossible because of some factors that affect packet matching [8, 9, 10, 11]. Fortunately, we don't need to match all the packets going through a TCP connection in a host for the purpose of detecting stepping-stone intrusion. All we need is enough data to establish a distinct level in RTTs, which are computed by matched TCP/IP packet's timestamps. If the conservative algorithm fails to produce enough matched packet pairs, we can always use the greedy algorithm to do it.

The rest of this paper is arranged as follows. In Section 2, we define some notations used in this paper. Section 3 discusses the difficulties involved in packet matching. In Section 4, we discuss the conservative and greedy packet matching algorithms. Section 5 shows the results when the two algorithms are tested on the Internet to detect stepping-stone intrusion. Finally, in Section 6, conclusions and future work are presented.

2. Preliminaries

Our research began with several assumptions. First, this research object is limited to an interactive session, which is made by Telnet, rlogin, rsh, ssh, or other similar tools. Secondly, there is no valid reason for a user to connect through more than three or four connections before reaching a destination which could be reached directly. Thirdly, it makes sense to assume that any users (including intruders), when connecting to a host, may need to pause to read, think, or respond to the previous operation; the time gaps between two continuous operations caused by human interaction are measured in seconds; these gaps are considerably larger than a typical round trip time of a network.

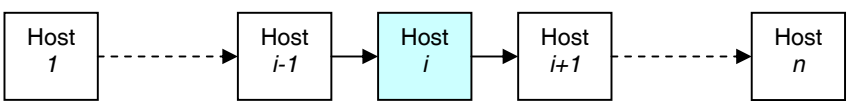

Fig. 1 A connection chain sample.

Suppose a user logs in from Host 1, and eventually connects to Host n, which is the destination, through Host 2, … , and Host n-1, as shown in Fig. 1. We here formally give definitions of the following terms: connection, chain, downstream, upstream, Send, Echo, Ack, and packet match.

Connection: When a user from a host logs into another host, we call this a connection session between the two hosts.

Chain: Given n hosts H_1, ..., H_n, a sequence of connections is defined as a chain $C = <C_1, C_2, C_3, \ldots, C_{n-1}>$ where C_i is a connection from Host H_i to Host H_{i+1} for i = 1, .., n-1.

Downstream and upstream: If a direction is along a user's login direction (as shown in the arrows in Fig. 1), it is called downstream. Otherwise, it is called upstream.

Send: A packet is defined as *Send* if it propagates downstream and has flags both 'Push (P)' and 'Acknowledgement (A)' or only 'P' [9].

Echo: A packet is defined as *Echo* if it propagates upstream and has flags both 'Push (P)' and 'Acknowledgement (A)' or only 'P'.

Ack: A packet is defined as Ack if it propagates either downstream or upstream and only has flag 'A'.

Matched packet: If a given Echo is directly triggered by a Send, then the Echo is defined as a matched packet of the Send. The method to find matched packets is called a packet-matching algorithm.

3. Challenges to Matching TCP/IP Packets

The packet-matching problem is to find the corresponding Echoes for each Send in a TCP/IP packet stream. The packets transmitted on the Internet are complex, but they can be decomposed into four simple cases. The simplest one is that each Send is followed by exactly one Echo; it is trivial that the Echo is the right one to match the Send. The second one is that several Sends are followed by exactly one Echo; in this case this Echo is supposed to match with the first Send. The third one is that one Send is followed by several Echoes; the first of the Echoes is supposed to match the Send. The final and most complex one is that several Sends are followed by several Echoes, in which case it is not as clear how to match them, but the first Echo is supposed to match the first Send.

In the TCP/IP communication on the Internet, the first case presents only when the chain is short, and matching the whole TCP/IP packets is trivial. However, the above four cases intersect each other along a chain that becomes long, where matching whole TCP/IP packets is impossible. There are many issues [8, 9, 10, 11] to affect matching TCP/IP packets; here we list the five

main reasons: (1) lost packet re-transmission; (2) packet cumulative acknowledgement and echo; (3) session transmit window; (4) packets communication between adjacent hosts (such as ignore packet, keep alive packet sent from client side, key re-exchange, these data are not intended for the target machine); and (5) multiple Echoes from a 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. Re-transmission of the same packet continues until either an acknowledgement is received or until the connection timeout expires. So we are faced with one Echo that could match with two or more Sends.

Every TCP packet is not always individually acknowledged; instead, cumulative acknowledgement may take place. The most important advantage of this mechanism is that it reduces the number of Ack messages, thereby reducing the possibility of network congestion. This network control mechanism benefits network traffic, but makes one-to-one packet-matching impossible. The same problem occurs for Echo packets too.

To control data flow 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. In this way, if this size is set to one, it means that each packet is sent if and only if the previous Send has been acknowledged or echoed. In most installation, this size is not one, so several packets can be allowed to send continuously before receiving any Ack. Several Send-Ack-Echo can overlap each other, making packet-matching difficult.

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

Keep-alive and Key re-exchange are packets that differ from the previous cases. They do not affect packet-matching, but the packets matched in these two cases are not what we expect because the packets are only sent to the neighboring host, rather than to the connection destination host. In most of a session's time, the key used for encryption is not changed, but it may be changed during the data transfer. It is recommended [11] that the key be 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 neighboring host, rather than the destination host. In this way, there is an echoed packet coming from the downstream neighboring 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 to the nearest neighboring downstream host, not the RTT to the target machine. It is easy to remove packets like these by setting a filter. When a command is executed at the target host, the result may be sent back in several packets, which also complicates the packet-matching.

In summary, the problems of packet-matching are inherited from the fact that Send and Echo packets may be in a many-to-many relationship, not one-to-one. It is impossible to match them deterministically even with a complete log. Therefore, it is extremely difficult to implement in real–time. It is significant to note that if we made a mistake in packet-matching at one point of a packet stream, the mistake would affect all the packet-matching after that point. To prevent this kind of mistake from occurring, our policy is to limit the mistake to a certain range. That is, we divide a packet stream into some sub-streams, one of which is the scope where we match the packets. If we make a mistake, it only affects the packet-matching within a specific sub-stream. Another benefit of dividing a packet stream into sub-streams is that in each sub-stream, there is at least one matched packet pair, for which we are confident that its match is correct. The third hypothesis we made in Section 2 guarantees that it is possible to divide a packet stream into some sub-streams online. When we are pretty sure about the packet matching correctness in each sub-stream, we match those packets, while discarding all those whose correctness we are not sure about.

This method for packet matching is called the Conservative Algorithm. Otherwise, once when we are not sure about the correctness of a packet-matching, we can still match those packets based on the time order, the sequence number, and the size of a packet. This method to match packets is called the Greedy Algorithm.

4. Conservative and Greedy Packet-matching Algorithms

Each (Send or Echo) packet between Host i and Host i+1 carries a sequence number and an acknowledgement (receive sequence) number. The initial send sequence number is chosen by the data sending TCP, and the initial receive sequence number is obtained during the connection establishing procedure [9]. Once the connection establishing procedure is done, the connection is going to enter the data communication phase.

In data communication, the sender of data keeps track of the next sequence number (SSN), which is going to be increased only by a Send. The receiver of data keeps track of the next sequence number, which is the acknowledgement number (RAN) of an Echo. Similarly, for each Echo there is a sequence number (RSN), which is used to keep track of the next available address to send an

Echo, while the sender side acknowledgement number (SAN) is used to keep track of the next unacknowledged Echo. Therefore, for each Send we have an SSN and an SAN, while for each Echo we have an RSN and an RAN. An SSN is increased only when a Send is sent while an SAN does not change unless there is an Ack/Echo received. Similarly, an RSN is increased only when an Echo is sent while an RAN does not change unless there is an Ack/Send received. Several continuous unacknowledged/un-echoed Sends should have the same SAN but with a different SSN. If and only if the first Echo satisfies the following conditions:

Send.SAN = Echo.RSN (1a)

Send.SSN < Echo.RAN (1b)

The first Echo is going to match either some of the Sends or all of the Sends. However, we are pretty sure that the first Echo definitely matches with the first Send; this is the basic idea of the Conservative Algorithm.

4.1 Conservative Algorithm

Though we have stated many challenges in matching all TCP packets, we do not have to match 100% of these packets to detect a new connection in the chain. If we can match a significant portion of the packets, it is sufficient for the purpose of estimating the length of a connection chain. There are two choices: (1) match only those that we are sure of their correctness; or (2) include some that we are not completely sure about. In the first algorithm, we collect only the matches that we are truly confident in their correctness and we sacrifice on the matching rate.

During an interactive terminal session, it is reasonable to divide a TCP/IP packet stream into some segments based on the third hypothesis we made: each segment is started with one Send. The gap between two continuous segments is supposed to be considerably larger than the RTT of a network. It is also safe to assume that no Echo packet will match a Send packet across the segment gaps. If two consecutive Send packets have timestamps difference that is more than TG (a predefined time gap threshold), we will assume the existence of a gap. In our experiments, TG was set to one second, which worked well.

We designed Algorithm 1 to match TCP packet, based on segment gap and the two conditions stated above. In this algorithm, an empty Send queue, which is used to store an unmatched Send, is initialized. Once a packet is captured, we first need to determine if it is a Send or an Echo. If it is a Send, we then decide if this is the first packet of a new segment by comparing the gap between this Send and the closest previous Send with the predefined threshold TG. If it is not a new segment, we simply add this Send to the end of the Send queue. Otherwise, we clear the Send queue to prepare for a new segment. If the packet captured is an Echo, we extract the first packet of the Send queue to match with the Echo by using conditions (1a) and (1b). If they are matched, we remove the Send from the queue; otherwise, we keep it and get to capture the next one.

The problem of this algorithm is its matching rate (MR) is low; MR is defined as the ratio between the number of matched packet pairs and the number of Sends captured. Many Sends that are supposed to be matched are discarded by Algorithm 1 because of the strict matching conditions (1a) and (1b). Once we get into a situation in which we cannot determine the proper matching, we clear the Send queue (by using the Boolean variable CorrectMatch in the algorithm). The advantage of this algorithm is that all the matches are correct. We are going to prove this point.

Suppose we use E, S to stand for Echo, and Send, respectively, each segment is going be expressed as either Case 1: $\{S_1\ E\ldots..\}$, or Case 2: $\{S_1\ S_2 \ldots S_n\ E\ldots\}$.

Proof: In Case 1, S_1 is the first Send of this segment, while E is the first Echo after S_1. When E is coming, S_1 is the only Send in the Send queue; if conditions (1a) and (1b) are satisfied, E must match with S_1. Suppose E does not

```
Initialize a SendQ queue;
CorrectMatch = true;  //Clear match flag
while (there are more packets) {
   Capture the next packet P;
   if P is a Send packet {
      Compute Time Gaps TG since last Send;
      if(TG > Threshold){
       Reset the SendQ;
       CorrectMatch = true;
    } else {add P to SendQ;}
   } else if P is an Ack packet{// Ignore it
   } else if P is an Echo packet{
      Q = dequeue (SendQ);
      if ((Q.ack# = P.seq#) and (Q.seq# < P.ack#)
           and (CorrectMatch)){
         Packets P and Q are matched;
         Compute round-trip time between P and Q;
      } else { // No match, set confusing match flag
         CorrectMatch = false;
      }
   }
}
```

Algorithm 1. The Conservative Algorithm

```
Initialize a SendQ queue;
while (there are more packets) {
   Capture the next packet P;
   if P is a Send packet {
      Compute Time Gap TG;
      If (TG > Threshold){ Reset the SendQ;}
      else {add P to SendQ;}
   } else if P is an Ack packet{
       // Ignore it
   } else if P is an Echo packet{
      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;
      } else if(((Q.ack# =< P.seq#)
            and (Q.seq# < P.ack#)){
         Packets P and Q are matched;
         Compute RTT between P and Q;
      } else {//No match;}
   } else {Return;}
}
```

1.1 Algorithm 2. Greedy Algorithm

match with S_1; it is supposed to match a Send before S_1, and this packet should belong to another sub-stream. This is in conflict with the assumption that each Echo in one segment only matches with the Send in this segment. So we proved E must match with S_1 in Case 1, whatever we have after E.

In Case 2, there are n Sends before E, for which we are not sure how to match the n Sends, but E must match with the first Send if (1a) and (1b) are satisfied. Suppose E does not match the first Send; then it is going to match any one after or before the first one. As we already proved, it is impossible to match the Send before the first Send, so the only possibility is to match the Send after the first Send. The first Send will not have any Echoes because any Echo after E is supposed to match the Send after the matched Send, rather than the first Send. This conflicts with the assumption that any Send is supposed to have at least one Echo. So we have proved E must match with S1 in Case 2.

4.2 Greedy Algorithm

The main reason that Algorithm 1 gives us low MR is that once we are confused about how to match the Send in a Send queue, we are going to discard all the Sends of the queue. Let us modify the packet matching policy in the following way. Once we get into confusion on which one in the Send queue is supposed to match, we are going to match the very first Send, and the following conditions must be satisfied:

Send.SAN < Echo.RSN (2a)

Send.SSN < Echo.RAN (2b)

These two conditions guarantee that the Echo is after the Send and it is not the first Echo.

We call this algorithm the Greedy Algorithm; Algorithm 2 gives more details and is presented in the following. Most of Algorithm 2 is similar to Algorithm 1, but when an Echo does not match the first Send of a Send queue using conditions (1a) and (1b), instead of discarding the Echo, we are going to match the first Send if conditions (2a) and (2b) are satisfied. The advantage of Algorithm 2 is that we can get a higher MR than Algorithm 1, but the problem is that we are not sure about the correctness of each matched pair unless conditions (1a) and (1b) are satisfied.

The Greedy Algorithm tends to give us a higher RTT when it was confused in matching the packets in a send queue. Let me give an example to explain this. Suppose there is a segment, {S_1 S_2 S_3 …S_n E_1 S_{n+1} E_2 ….}, in which we already know that E_1 matches with S_1, and E_2 matches with S_3. But after processing by the Greedy Algorithm, the result should be that E_1 is going to match with S_1 using conditions (1a) and (1b), while E_2 is going to match S_2 using conditions (2a) and (2b). The RTT between E_2 and S_2 obtained by the Greedy Algorithm is larger than it is supposed to be because S_2 is before S_3. The higher RTT does not hurt the purpose for stepping-stone intrusion detection; we are going to explain the detailed reason for this in Section 5.

4.3 Justification for Greedy Algorithm

We have proved that the matched packet pairs by the Conservative Algorithm are correct. However, we cannot claim the same for the Greedy Algorithm. In case we do not collect enough data points, we can use the later because it gives us a high matching rate.

In this section, we are going to evaluate the performance of the Greedy Algorithm by comparing its results with those of the Conservative Algorithm in an experiment which is designed with the both algorithms running on the same host concurrently. We concern with two benchmarks: MR, and accuracy rate (AR), which is the ratio between the number of correctly matched packet pairs and the number of the whole matched pairs. The problem is how to determine which matched pair is the correct one. In the experiment of this paper, we use Telnet results to examine the correctness of the Greedy algorithm

because there is no content encryption for TCP packets from Telnet session, and thus make it possible to examine the correctness of packet-matching.

In this experiment, our connection chain spanned from Houston to Mexico and California: UH1→Mex→UH2→Epic→UH3→Mex, where UH1, UH2, UH3 are located in Houston, Texas, Mex is located in Mexico, and Epic is located in California. We collect and match TCP packets at UH1. From the experiment, we confirmed that (1) the Greedy Algorithm produces significantly more RTTs; (2) all pairs matched in the Conservative Algorithm are also matched in the Greedy Algorithm; and (3) the MR and AR depend on the number of hosts; longer RTT means more hosts connected. Table 1 shows the MRs and ARs of a typical experiment with five connections in the chain. As the RTT increased for the Conservative Algorithm, the MR dropped from 100% to about 21% while the AR did not change; for the Greedy Algorithm, which has a very high MR, the MR dropped a little while AR changed significantly, from 99% to 86%.

Table 1: Comparison of MR and AR between the Conservative and Greedy algorithms

Number of Hosts connected from UH1	RTT (ms)	Conservative (%)		Greedy (%)	
		MR	AR	MR	AR
1	61	100.0	100.0	100.0	98.7
2	120	70.0	100.0	100.0	96.1
3	172	38.1	100.0	100.0	92.3
4	222	27.5	100.0	98.6	90.0
5	282	21.6	100.0	96.0	85.5

5. Application to Stepping-stone Intrusion Detection

5.1 Detecting long interactive connection chain on the Internet

If we start monitoring the packet transmission from the beginning when 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 Fig. 2). In other words, we can use the changes of the RTTs to signal the change in the connection chain [12]. If we monitor the chain continuously, we should get a step function with each step corresponding to one host connection. If we can count how many steps we have, we should exactly know how many hosts are connected in the downstream chain. The question is whether we can get a step-function by monitoring all the packets passing through Host i, where a RTT is computed by the Conservative and Greedy Algorithms. We conducted an experiment to verify this point and compare the results obtained by the two algorithms.

In our experiment, we connected to five hosts from local host UH1. The connection chain, UH1 → UH2 → Mex → UH3 → Epic → UH4, is a typical setup. We varied the setup and included other hosts, but the results were consistent with those presented side-by-side for comparison in Fig. 2. It can be seen that most of the additional data points from the Conservative Algorithm are very close to the data points collected from the Greedy Algorithm, even though there are a few exceptions, and all of them are higher than “normal” data points at different levels.

We monitored the packets (using both Algorithms 1 and 2) at UH2 for about twenty minutes. We captured the Send and Echo packets of outgoing connection of UH2. The numbers shown in Fig. 2 and Table 1 are based on experiments with minimum keystrokes, i. e., we logon to a host and immediately Telnet/SSH to another machine. We

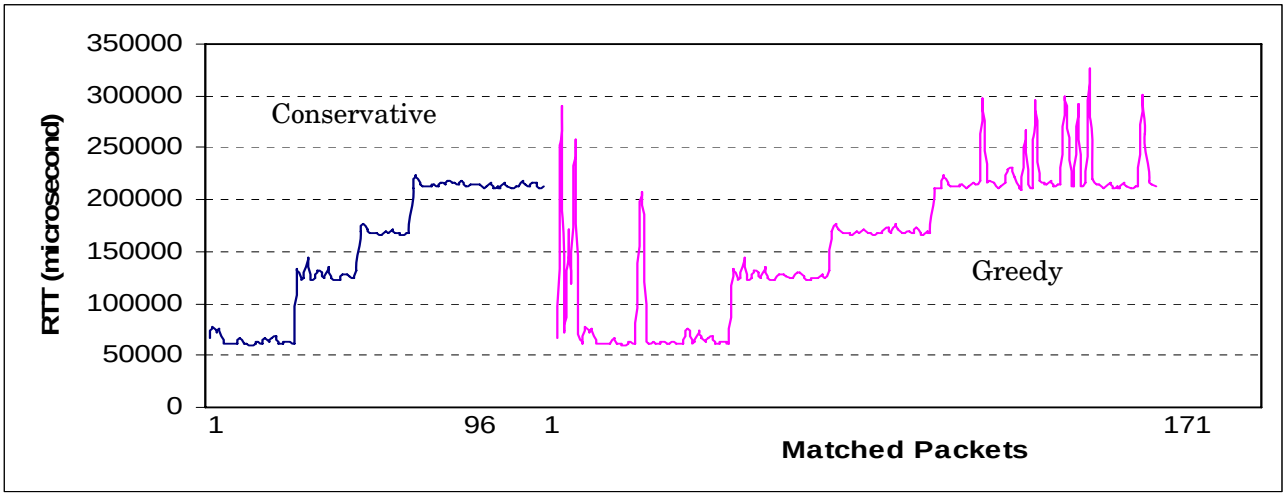

Fig. 2 Apply the algorithms to detect the length of connection chains.

were able to obtain enough data points in Fig. 2 to establish distinct levels. In case the data points are not enough (because of the number of connections or hacker manipulation), the Greedy Algorithm can be used to obtain more data points (with less quality). Using the matching algorithms, we were able to collect an array of packet roundtrip times, such as the one showed in Fig. 2. In this figure, we can clearly confirm our conjecture that the roundtrip time is "almost" a step-function. Four steps can be seen, formed by the lower bound of the four segments. For obvious reasons, Fig. 2 is not a perfect step-function.

It is clear that the “steps” on the two curves in Fig. 2 are almost identical. There are some data points that are considerably higher than their neighbors. The first group is at the beginning of the Greedy Algorithm curve; these are probably due to synchronization of the hosts. The second group of these fluctuations occurs at level four on the right; they represent the incorrect matching of the Send/Echo packets, which are corresponding to what we predicted.

5.2 Counting the steps in a chain

We needed to use a method to detect when the “jump” (up or down) happened in the roundtrip time array, and eventually to count the number of the steps. The up-jump represents an additional connection in the (downstream) chain. We designed Algorithm 3 to detect “jumps” in the packet roundtrip array found in Algorithms 1 or 2. Algorithm 3 can be used in real-time as the values in the array are filled. It only examines the last 2*w elements in determining whether there is a jump in the round-trip time. Intuitively, we split the 2*w elements into two windows (left and right) of size w each. Within the windows, we selected the minimum of w values 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.

Algorithm 3 uses a window of size 6 (w = 3 on each side) that worked reasonably well in our experiments. The larger the window size, the better the algorithm. In Section 4.2, we note that if we occasionally get some higher RTTs, it does not hurt the purpose for stepping-stone detection. From Algorithm 3, we already know the only case that this algorithm gives us a wrong number on counting the steps in a round trip time array is under the situation that there are w (here, we select w=3) consecutive higher RTTs, but this probability is very low. On the other hand, suppose we have w consecutive higher RTTs; the mistake can only make us overestimate the length of a chain, unlike the case with w consecutive lower RTTs; we will not lose any intrusion.

Algorithm 3 above was simplified to find only one jump. It can be generalized to find all jumps in the array. We claim that the algorithm is real-time because we have to look ahead w packet data in order to determine whether a jump occurs. If we accumulate all the jumps, we can easily determine the number of intermediate hosts. This estimate can then be used to terminate a telnet or ssh session if the chain exceeds a predefined size.

```
  // The simplified algorithm below finds only one (the first)
“jump”
  // Given: Packet roundtrip time array rtt[]
  //
  threshold = average of some randomly selected values in array
rtt[];
  for (each element artt, i>=5) {
     minLeft = min(rtt[i-5], rtt[i-4], rtt[i-3]);
     minRight = min(rtt[i-2], rtt[i-1], rtt[i]);
     diff = minRight – minLeft;
     if (diff > 0) {
        jumpDir="up";    // login on to a new host
     } else {
        diff *= -1;
        jumpDir = "down";   // logout from host
     }
     if (diff > threshold) {
        if (jumpDir == "down") {
           there is a jump down between rtt[i-3] and rtt[i-2];
        } else {
           there is a jump up between rtt[i-3] and rtt[i-2];
        }
     }
  }
```

Algorithm 3: Computing jumps in a roundtrip time.

6. Conclusions and Future Work

We have described a Step-Function method that uses RTTs to estimate the length of a long interactive connection chain by counting the number of steps. An algorithm used to examine the RTT array to count the steps has been proposed; it can work in real time and worked well on the Internet with the experiments we did by selecting window size w=3. We also have proposed the Conservative and the Greedy Algorithms to match TCP/IP packets online for computing the RTTs of a TCP interactive session. The Conservative Algorithm can give us correct matches, which have been proved, but with lower MR. The Greedy Algorithm can give us more matched packets but for some we are not confident about their correctness. We evaluated the performance of the Greedy Algorithm, and the results showed that this algorithm is more useful and practical than the conservative one for the purpose of stepping-stone intrusion detection.

The approach used here to detect stepping-stone intrusion by observing the changes of RTTs to determine the number of hosts in an interactive connection chain is different from that of previous methods [1, 2, 3]. Our approach has the following advantages: (i) the ability to detect intruders in real-time, (ii) the ability to handle encrypted terminal sessions, (iii) the ability to estimate the length of a chain accurately, and (iv) the ability to tolerate network traffic fluctuation, network load, and workload of chained hosts.

There are still some limitations and restrictions. We must be able to monitor a 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 approach to detect the additional host.

References

[1] Yin Zhang and Vern Paxson, “Detecting Stepping Stones,” Proc. 9th USENIX Security Symposium, Denver, CO, 2000, pp.67-81.

[2] Stuart Staniford-Chen and L. Todd Heberlein, “Holding Intruders Accountable on the Internet,” Proc. 1995 IEEE Symposium on Security and Privacy, Oakland, CA, 1995, pp.39-49.

[3] Kwong H. Yung, “Detecting Long Connecting Chains of Interactive Terminal Sessions,” Proc. International Symposium on Recent Advance in Intrusion Detection, Zurich, Switzerland, 2002, pp.1-16.

[4] Z. J. Tang, Designing and Implementing of Network Intrusion Detection System, Publishing House of Electronics Industry of China, Beijing, China, 2002.

[5] Behrouz Forouzan, TCP/IP Protocol Suite (Second Edition) ,McGraw-Hill, New York, 2002.

[6] Lawrence Berkeley National Laboratory (LBNL), “The Packet Capture library,” ftp://ftp.ee.lbl.gov/libpcap.tar.z, accessed March 2004.

[7] Data Nerds Web Site, “Winpcap and Windump,” http://www.datanerds.net, accessed July 2004.

[8] T. Ylonen, “SSH Protocol Architecture, draft IETF document,”http://www.ietf.org/internet-drafts/draft-ietf-secsh-architecture-16.txt, accessed June 2004.

[9] University of Southern California, Transmission Control Protocol, RFC 793, 1981.

[10] Martin P. Clark, Data Networks, IP and the Internet Protocols, Design and Operation, Wiley, New York, 2003.

[11] T. Ylonen, “SSH Transport Layer Protocol, draft IETF document,”http://www.ietf.org/internet-drafts/draft-ietf-secsh-transport-18.txt, accessed June 2004.

[12] Jianhua Yang and Shou-Hsuan Stephen Huang, “A Real-Time Algorithm to Detect Long Connection Chains of Interactive Terminal Sessions,” Proc. 3rd International Conference on Information Security (Infosecu’04), Shanghai, China, 2004, pp.198-203.

[13] CERT, “Explosion of Incidents,” http://www.cert.org, accessed July 2004.

[14] David L. Donoho, Ana Geogina Flesia, et al., “Multiscale Stepping-Stones Detection: Detecting Pairs of Jittered Interactive Streams by Exploiting Maximum Tolerable Delay,” Proc. International Symposium on Recent Advance in Intrusion Detection, Zurich, Switzerland, 2002, pp.49-64.

[15] K. Yoda and H. Etoh, “Finding Connection Chain for Tracing Intruders,” Proc. 6th European Symposium on Research in Computer Security (LNCS 1985), Toulouse, France, 2000, pp.31-42.

Jianhua Yang received the B.E. and M.E. degrees, from Shandong Univ., China in 1987 and 1990, respectively. He received his Ph.D. from Univ. of Houston, U.S.A. in 2006. He is currently an assistant professor in the Dept. of Mathematics and Computer Science, Bennett College, U.S.A. His research interest includes computer, network, and information security. He is a member of IEEE Computer Society.

Shou-Hsuan Stephen Huang is professor of Computer Science at the University of Houston. His research interests include Data Structures and Algorithms, Intrusion Detection and computer security. Stephen Huang received his Ph. D. degree from the University of Texas-Austin. He is a senior member of the IEEE Computer Society. Dr. Huang can be reached at shuang@cs.uh.edu.