Detecting Connection-Chains: A Data Mining Approach

Ahmad Almulhem and Issa Traore
(Corresponding author: Ahmad Almulhem)

ISOT Research Lab, Department of Electrical and Computer Engineering
University of Victoria, Victoria, B.C., V8W 3P6, Canada
(Email: {almulhem, itraore}@ece.uvic.ca)
(Received Feb. 6, 2008; revised Apr. 23, 2008, and accepted June 9, 2008)

Abstract

A connection-chain refers to a mechanism in which someone recursively logs into a host, then from there logs into another host, and so on. Connection-chains represent an important vector in many security attacks, so it is essential to be able to detect them. In this paper, we propose a host-based algorithm to detect them. We adopt a black-box approach by passively monitoring inbound and outbound packets at a host, and analyzing the observed packets using association rule mining. We first explain the proposed algorithm in greater details, then evaluations are presented to demonstrate its efficiency and detection capabilities. We conduct the evaluation using public network traces, and show that by appropriately setting underlying parameters we can achieve perfect detection, meaning a true positive rate (TPR) of 100% and a false positive rate (FPR) of 0%.

Keywords: Connection chains, network forensics, network security, stepping stones

1 Introduction

The term *connection chain* refers to a mechanism in which someone recursively logs into a host, then from there logs into another host, and so on [6]. Due to the design of tcp/ip suite, the origin of the chain is effectively concealed as we move down the chain. As such, a connection chain provides an effective interactive channel to remotely manipulate a host without revealing someone's origin. From a network security perspective, connection chains are then a security risk, because they can be used by attackers to stay anonymous.

Detecting and tracing connection-chains is a challenging yet important task for a number of applications. The following is a sample:

- Network Forensics: Tracing connection-chains plays a crucial role in network forensics applications. Particularly, it has the potential of revealing an attack's path as well as the involved hosts. Investigation then typically proceeds by isolating affected hosts and collecting data from them. Ideally, such tracing also may lead to the origin of an attacker especially insiders. Coupled with collected evidences, the attacker may also be prosecuted in a court of law.

- Liability: If a host owned by an organization were exploited as part of a connection chain, the attack would appear to be originating from this organization. As a result, they may be held liable for such attack. Detecting connection-chains can help to enforce policies of transit traffic.

- Deterrence: Anonymity is a main concern of serious attackers. In fact, it is the whole purpose of establishing a connection-chain in the first place. An effective tracing tool will deter some attackers in fear of exposing their true origin.

In the literature, different approaches have been proposed to detect and trace connection chains. These approaches can be broadly classified into *host-based* [4, 5, 12], *network-based* [3, 6, 9, 17, 26, 27, 28, 29, 30, 31, 32], and *system-based* [5, 11, 21, 25]. *Network-based* approaches operate on packets at the network level; *host-based* approaches function inside hosts; *system-based* approaches employ both host-based and network-based components. We refer the interested reader to our review paper [2] for a taxonomy and a detailed discussion of these approaches. In this paper, we are specifically interested in host-based approaches.

In general, the main disadvantage of the host-based approaches proposed so far in the literature is that they are operating system specific. Specifically, they are expected to be re-designed and re-implemented differently for different operating system. Also, it is not obvious if they can be applied to proprietary operating systems such as MS Windows.

In this paper, we propose a technique to detect connection-chains at a host. The technique avoids being

operating system specific, by employing a black-box approach. In essence, inbound and outbound packets are *passively* monitored to detect if there is a connection-chain. The technique is inspired by concepts from association-rule mining in the data mining literature. It also has the following features:

- Portable: The approach is independent of any operating system.
- Real-time: The algorithm is efficient for real-time processing.
- Preserve Privacy: For a packet, the algorithm only uses its arrival time and its header to operate. It neither stores nor uses its payload.
- Robust: The algorithm resists encoding (encryption/compression), because neither packets' payloads nor their lengths are used in the analysis.
- Relative Interpretation: Instead of a crisp yes/no answer, a *confidence* measure is attached to possible connection-chains. This particular feature enables the setting of a user-defined threshold ($minconf$) in order to reduce false positives.

The rest of the paper is organized as follows. In Section 2, we survey and discuss related work. In Section 3, we present background materials that are related to this work. We then provide an overview of our approach in Section 4. In Section 5, the details of the algorithm and used data structures are explained. In Sections 6 and 7, we present our evaluations and experimentations of the algorithm. In Section 8, we discuss how the proposed framework can handle various forms of evasions. Finally, we conclude in Section 9.

2 Related Work

In this section, we summarize and discuss related works on connection-chains detection. As indicated earlier, the proposed approaches can broadly be classified into *host-based*, *network-based*, and *system-based*. But since our work is a host-based approach, we will limit our discussion to only host-based approaches. We refer the interested reader to [2] for a detailed coverage of related works on network-based and system-based approaches.

At a host, a connection chain appears as a pair of connections through which some traffic pass back and forth. To identify a pair of connections among a set of connections, two categories of approaches have been proposed in the literature. In the first category, the link between two connections is found by searching the running processes at the concerned host [5, 12]. The idea is that if an outbound connection c_o is created by an inbound connection c_i, then the processes p_i attached to c_i and p_o attached to c_o are somehow linked. Depending on the operating system, the processes' tree is searched to discover if such link does exist. This approach is quite simple and gives accurate results. It, however, may fail if the concerned processes are created in an unusual way. For instance, in Unix, the search can be broken if the processes are created using deep nested pipes and local sockets [5].

The second category of approaches recognizes that, by default, operating systems do not have a function or a data structure that tells whether an outbound connection has been created by an inbound connection. Therefore, these approaches propose modifying an operating system to support linking an outbound connection to an inbound one. For instance, Buchholz and Shields proposed the following modification [4]. For each process, a new data structure, called `origin`, is stored in its process table. For processes created by a remote connection, `origin` holds information related to that connection. For locally created processes, `origin` is undefined. Surely, this approach would make identifying connection chains a matter of checking a data structure. It is, however, unattractive, because of the expected costs of modifying an operating system. It may even break already running software.

As indicated earlier, the main disadvantage of proposed host-based approaches is that they are operating system specific.

3 Background

In this section, we present background materials which are related to this work. In particular, we first present the terminology used throughout this paper. Then, we present an attack model of connection-chains.

3.1 Terminology

In this paper, a *connection* refers to an *established* `tcp` connection, which is uniquely identified by its two end points [18]. It constitutes a bidirectional channel that enables both ends to send and receive data. To distinguish between the two directions, we refer to each one as a *flow*. The flow from the remote host to the local host is referred to as an *inbound flow*, while the other flow is referred to as an *outbound flow*. Packets flowing in each flow are denoted as *inbound packets* and *outbound packets* respectively.

In TCP/IP suite, applications like `telnet` [19], `rlogin` [13] and `ssh` [14] are used to log into a host and acquire a *virtual terminal* (or simply a *terminal*) on that host. The terminal (also called *console* or *shell*) is useful to execute commands and other programs *interactively*. For convenience, we refer to such applications as *terminal applications*. If a user runs a terminal application on host h_0 to log into another host h_1, a terminal on host h_1 is obtained and a connection c_0 is established. The user then may use the terminal at host h_1 to log into another host h_2. This procedure may be repeated creating a series of connections as follows:

$$|h_0| \longleftarrow c_0 \longrightarrow |h_1| \longleftarrow \cdots \cdots \longrightarrow |h_{n-1}| \longleftarrow c_{n-1} \longrightarrow |h_n|$$

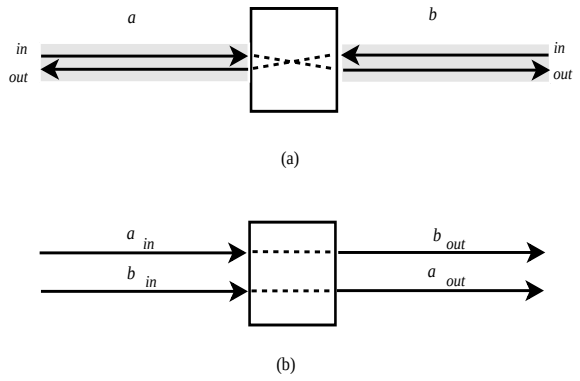


Figure 1: Two views of a connection-chain at a host: (a) A connection-view. (b) A flow view.

This series of connections is called a *connection-chain* [6], and the intermediary hosts are called *stepping-stones* [32].

3.2 Attack Model

Consider the host shown as a rectangle in Figure 1. Suppose that this host is exploited as a stepping-stone, and the connections a and b are part of the corresponding connection-chain. Let a_{in}, a_{out}, b_{in} and b_{out} be the corresponding inbound and outbound flows. Since a and b are bidirectional connections, packets in one connection should re-appear "later" in the other connection. Specifically, an inbound packet coming on a_{in} is expected to reappear as an outbound packet on b_{out}. Similarly, an inbound packet coming on b_{in} is expected to reappear as an outbound packet on a_{out}. The packets' flow inside the host is shown as dotted line. Figure 1 (a) depicts a *connection view* of the connection-chain. Figure 1 (b) depicts a *flow view*, where inbound and outbound flows are sorted out.

An important question is how soon a packet in one connection will re-appear in another connection if the two connections are part of a connection-chain. Actually, it is not possible to exactly determine this delay ahead of time. However, we know that for a connection-chain to work, this delay has to be bounded [7]. Otherwise, a tcp connection will timeout and start retransmission or even disconnect. In other words, we expect the delay to be random, but not to exceed some constant value. Throughout this paper, we refer to this constant as Δ.

In reference to the flow view shown in Figure 1 (b), our algorithm tries to identify outbound packets that might have been triggered by inbound ones. This is considered the case when the difference between the timestamps of an outbound packet and inbound one is $\leq \Delta$.

4 Approach Overview

In this section, we present an overview of our approach. We begin with a brief review of association-rules mining, which forms the basis of the proposed approach.

4.1 Mining for Association Rules

In data mining, *association analysis* is a methodology used to discover interesting relationships in large data sets [24]. The term *association rules* is used to denote the discovered relationships, while the process itself is called *mining for association rules* [1].

The following formulation is adopted from [24]. Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of n *items*. Let $T = \{t_1, t_2, \ldots, t_N\}$ be a set of N transactions, where each *transaction* t_i contains a subset of items from I, i.e. $t_i \subseteq I$. An *itemset* is defined as a set of items. A *k-itemset* is an itemset that contains k items. For instance, $\{bread, milk, eggs\}$ is a 3-itemset.

A transaction t_i is said to contain an itemset X, if $X \subseteq t_i$. An important property of an itemset is its *support count*, which refers to the number of transactions that contain a particular itemset. Mathematically, the support count $\sigma(X)$ of an itemset X is given by the following formula:

$$\sigma(X) = |\{t_i | X \subseteq t_i, t_i \in T\}|,$$

where $|.|$ denotes the number of elements in a set.

An *association rule* is an implication of the form $X \rightarrow Y$, where X and Y are disjoint itemsets, i.e. $X \bigcap Y = \phi$. The quality of an association rule is measured by its *support* and *confidence*, which are defined as follows:

$$\begin{aligned} support &\equiv s(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{N} \\ confidence &\equiv c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}. \end{aligned}$$

Intuitively, the support implies that X and Y occur together in $s\%$ of the total transactions. On the other hand, the confidence implies that, of all the transactions containing X, $c\%$ also contain Y. It should be noted that there are alternative indices to measure the quality of association rules besides the mentioned *support-confidence framework*. For an account of various indices, the interested reader may refer to [24]. Also, it is important to note that the implication in an association rule does not necessarily mean *causality*. It simply indicates a co-occurrence relationship between the items in the antecedent and consequent of the rule. The problem of association rules mining can be formally stated as follows:

Definition 1. Association Rule Mining*: Given a set of transactions T, find all the rules having support $\geq$ minsup and confidence $\geq$ minconf, where minsup and minconf are user-defined support and confidence thresholds respectively.*

4.2 Mining For Connection-Chains

In our approach, we closely follow the classic formulation of association rules mining that was presented in the previous section. In this instance, the items of interest

correspond to a set of connections, while the connection chains correspond to the desired association rules. Additionally, a confidence measure is used to denote the strength of a particular connection chain. Formally, let $C = \{c_1, c_2, \ldots, c_n\}$ be the set of active *connections* at a given host. Also, let $T = \langle t_1, t_2, \ldots \rangle$ be a *sequence* of transactions, where a *transaction* is restricted to one of the following two classes of transaction types:

- *A 1-tuple Transaction*: A transaction of the form $[c_i]$, where $c_i \in C$.
- *A 2-tuple Transaction*: A transaction of the form $[c_i, c_j]$, where $c_i \neq c_j$ and $c_i, c_j \in C$. In this type of transaction, the order is not significant (i.e. $[c_i, c_j] = [c_j, c_i]$).

Unlike the original formulation, the transactions here are generated dynamically as packets flow in the connections. As such, we refer to the collection of transactions as a sequence instead of a set. A 1-tuple transaction $[c_i]$ is generated whenever an *inbound* packet is received on the corresponding connection. On the other hand, a 2-tuple transaction $[c_i, c_j]$ is generated whenever an *inbound* packet in one connection is followed by an *outbound* packet in the other connection within a Δ amount of time. We will provide more details about the generation of these transactions when we discuss the proposed algorithm in the next section.

In theory, for a set of n connections, there are n 1-tuple transaction types, and $\binom{n}{2}$ 2-tuple transaction types, where the symbol $\binom{\ }{\ }$ denotes the *combination* (*choose*) operator. For example, let the set of connections be $\{a, b, c\}$. Then, there are 3 1-tuple transaction types: $[a]$, $[b]$ and $[c]$. Also, there are 3 2-tuple transaction types: i.e. $[a, b]$, $[a, c]$ and $[b, c]$.

In our formulation, we make a distinction between a transaction type t_i and how many times t_i actually occurred. We use the *support count* $\sigma(t_i)$ to refer to how many times a transaction of type t_i has occurred. For example, let the set of connections be $\{a, b, c\}$. Then, we might have the following support counts: $\sigma([a]) = 10, \sigma([b, c]) = 2$, etc.

As mentioned earlier in the introduction, a connection chain appears at a host as a pair of connections through which traffic flows back and forth. As a result, in our framework we view a *connection-chain* as an association rule of the form $\{c_i, c_j\}$, where $c_i \neq c_j$ and $c_i, c_j \in C$. Note that a set notation is used to represent a connection chain instead of an implication ($\rightarrow$). This is to emphasize the fact that a connection chain does not imply any direction. For instance, given a connection chain $\{c_i, c_j\}$, packets are investigated in both directions (i.e. $c_i \rightarrow c_j$ and $c_j \rightarrow c_i$).

The confidence of a connection-chain $\{c_i, c_j\}$ is defined as follows:

$$confidence(\{c_i, c_j\}) = \frac{\sigma([c_i, c_j])}{\sigma([c_i]) + \sigma([c_j])}, \quad (1)$$

where $c_i \neq c_j$ and $c_i, c_j \in C$. Intuitively, the numerator represents those times when packets in both connections occur within Δ amount of time, while the denominator represents the times they occur solely. Typically, a true connection chain is expected to have a high confidence close to 1.

5 Algorithm and Implementation

In this section, we explain how our approach is implemented. In particular, we discuss the used data structures, algorithm details, and relevant implementation issues.

5.1 Data Structures

In the proposed algorithm, we employ the following two main data structures.

- **A Connection-Chain Graph (*ccGraph*)**: An undirected weighted graph $G(N, E, W)$, where
  - a node $n \in N$ represents an active connection c_i. For 1-tuple transactions of type $[c_i]$, the support count $\sigma([c_i])$ is stored here.
  - an edge $e \in E$ exists between two nodes v (representing a connection c_i) and u (representing a connection c_j), if there are packets suspected to be flowing between the two connections. An edge's weight $w \in W$ corresponds to the support count $\sigma([c_i, c_j])$ for the 2-tuple transactions of type $[c_i, c_j]$.
- **Inbound Packets Set (*inSet*)**: A set of *current* inbound packets. When a new inbound packet is received, it is added to this set. For a particular connection, *inSet* contains only the most recent inbound packet on that connection. In some respects, this set acts as a time-sliding window of the current inbound packets. The size of the window is approximately Δ time unit.

Figure 2 depicts an overall flow of the algorithm. It also shows how these data structures fit in the whole picture. In essence, the input to the algorithm is a stream of packets seen at a host's network interface. Inbound packets are buffered into *inSet* for later comparisons. Outbound packets are not buffered. Instead, they are compared with the already buffered inbound packets to determine if there is a correlation. In addition, *ccGraph* maintains an up-to-date status of active connections and possible correlations between them. In particular, *ccGraph* holds the support counts for the two types of transactions that was discussed earlier.

5.2 Algorithm

In Figure 3, we summarize the whole algorithm as a pseudocode. For each packet $p \in P$, the following operators are defined:

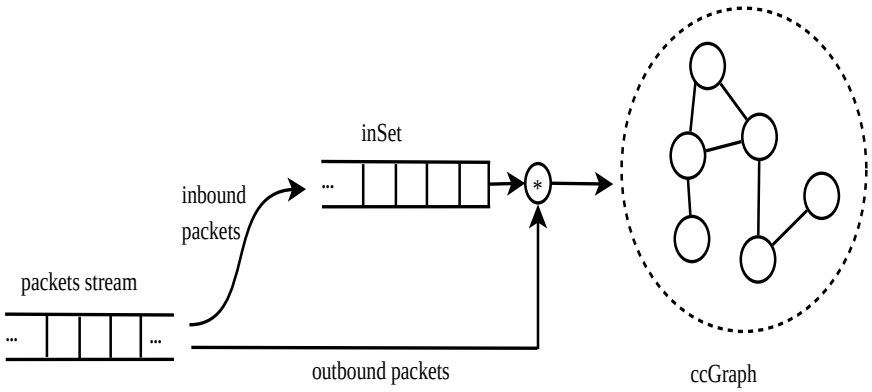


Figure 2: Overall flow of the algorithm and used data structures

```
1: procedure PROCESSPACKET p
2: if d(p) = in then
3:    increment σ([c(p)]) in ccGraph
4:    add p to inSet
5: else if d(p) = out then
6:    for all q ∈ inSet do
7:       if t(p) − t(q) ≤ Δ then
8:          if c(p) ≠ c(q) then
9:             increment w(c(p), c(q)) in ccGraph
10:         end if
11:      else
12:         remove q from inSet
13:      end if
14:   end for
15: end if
16: end procedure
```

Figure 3: The overall algorithm as a pseudocode

- $t(p)$: the time-stamp of p.
- $c(p)$: the connection to which p belongs.
- $d(p)$: the direction of p; either inbound (in) or outbound (out).

The received packets are processed on a first-come-first-served basis according to their timestamps. Each packet generates a transaction that depends on its direction. An inbound packet p generates a 1-tuple transaction of the type $[c(p)]$. The support count for this type of transaction $\sigma([c(p)])$ is stored into the corresponding node in *ccGraph*. Additionally, the packet itself is buffered into *inSet* as mentioned earlier. Intuitively, an inbound packet either results in incrementing the count of that connection, or creating a new node (connection) if it does not exist. Additionally, it is buffered in *inSet* for later comparison with outbound packets.

On the other hand, an outbound packet p generates a 2-tuple transaction of type $[c(p), c(q)]$, if $q \in inSet$, $c(p) \neq c(q)$, and $t(p) - t(q) \leq \Delta$. The support count for this type of transaction $\sigma([c(p), c(q)])$ is stored as a weight $w(c(p), c(q))$ of the edge between $c(p)$ and $c(q)$ in *ccGraph*. Intuitively, we keep counts of every pair of outbound and inbound packets, if the outbound packet comes after the inbound one by at most Δ.

At any time, connection-chains are edges in *ccGraph* that have a *confidence* exceeding some user-defined threshold ($minconf$). The confidence is computed according to Equation (1).

5.3 Implementation

The proposed algorithm operates by silently analyzing inbound and outbound packets at a host. Therefore, the algorithm requires a way to capture the packets. The *packet capture library (libpcap)* provides a simple and portable API for packet capturing [10]. It is available on most operating systems, and can be used within many high-level languages. Technically, the packet capture part is the only part that interface with the operating system. Otherwise, the inner details of an operating system (such as processes) are irrelevant.

In order to evaluate our approach, we implemented the algorithm in Java 1.5 [23]. Java was chosen for its portability, as the Java Virtual Machine (JVM) is already available on many platforms including PCs, mobile phones, etc. For the capture part, we used a Java binding of the *libpcap* library, called *Jpcap* [8]. For the data structures, *ccGraph* was implemented using two hash tables; one for the nodes (connections), and one for the edges (connection-chains). *inSet* was also implemented using a hash table.

6 Evaluation

In this section, we present an extensive evaluation of our approach. We first discuss the used dataset and our experimentation settings. Then, we provide detailed analysis of the proposed algorithm in terms of false positives, true positives and processing time.

6.1 Data and Settings

In general, choosing an appropriate dataset to evaluate a detection algorithm is a nontrivial task. Firstly, the data has to be a faithful representation of the real situation. Secondly, the data has to be labeled (i.e. true and false positives have to be correctly marked at first). Unfortunately, there is no public dataset (or even simulation tools) specifically designed for connection chains detection algorithms. As such, we had to be creative in evaluating our approach.

We used a public network trace to assess the algorithm in terms of processing time and detection capability. The trace is called *LBNL-FTP-PKT*, and is available from [15]. The trace contains all incoming anonymous FTP connections (i.e. to port 21) to public FTP servers at the Lawrence Berkeley National Laboratory during a ten-day period in Jan 10-19, 2003. It contains 3.2 million packets flowing in 22 thousand connections. The connections are between 320 distinct servers and 5832 distinct clients.

We believe the selected trace fits our needs very well. First, it is reasonably large to assess the algorithm. Secondly, it only contains the interactive part of ftp sessions. This type of traffic is similar to interactive traffic generated by applications like telnet/ssh, and hence similar to traffic seen in connection-chains. Finally, the trace does not contain any connection chains. As such, it is already labeled (i.e. any detected connection chain is false positive). We did, however, simulate some connection chains to evaluate the true positive detection rate as we show later.

Initially, we sliced the trace into 320 subtraces by server ip address. Each subtrace contains the packets exchanged with the corresponding server. Then, we run the algorithm on those subtraces, as if the algorithm was running on the corresponding server.

In order to fully test the algorithm, we set $minconf$ to 0 in our test suite. This allowed us to study all connection chains detected by the algorithm regardless of their confidences. In some respects, $minconf = 0$ actually corresponds to the worst case scenario. Additionally, by analysing inbound and outbound packets of those servers, we estimated the response time of the servers to be between 10-90 msec. We used this value as a guidance to set Δ in our test suite. In particular, Δ is varied as 1, 10, 50, 100, 200, and 500 msec. The selected values are intended to investigate the effect of varying Δ, when Δ is set *below*, *around*, and *above* the estimated true Δ value.

Finally, the tests were run on a laptop with the following specifications: a 1.3Ghz Intel Pentium m-processor, 2 GB RAM, and 80 GB 7200 RPM Hard drive.

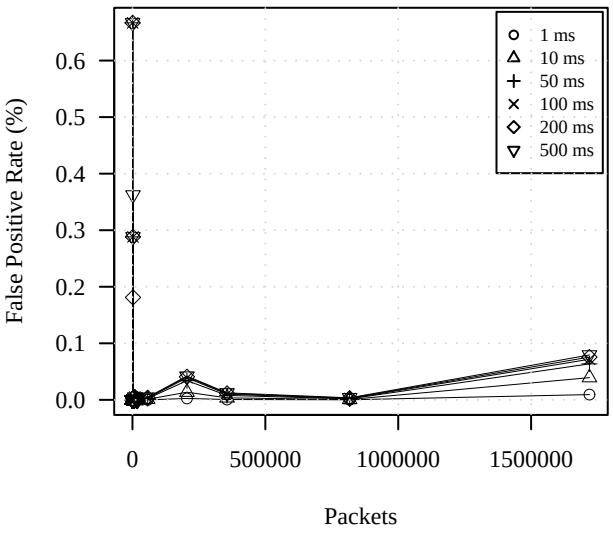


Figure 4: The false positive rates of the 320 subtraces sorted in increasing order of the number of packets. Notice that the false positive rate is less than 0.1% in most of the cases. Also notice that, in general, increasing Δ increases the false positive rate.

6.2 False Positives Analysis

It is important for a detection algorithm to have a low *false positive rate*[1] (i.e. generates fewer false positives). In this part, we investigate this aspect of the proposed algorithm. We also investigate the effect of varying Δ.

The experimentation was performed as follows. For every subtrace (320 subtraces), we run the algorithm with a Δ of 1, 10, 50, 100, 200, and 500 msec (i.e. a total of $6 \times 320 = 1920$ cases). For every case, we observed the number of connection chains detected by the algorithm. Because the original subtraces do not contain connection chains, any detected connection chain is a false one. Accordingly, we computed the false positive rate (FPR) as follows:

$$FPR = \frac{number\ of\ false\ positives}{number\ of\ possible\ negative\ instances} \times 100. \quad (2)$$

For a particular subtrace S_i, the number of possible false instances equals $\binom{C_i}{2}$, where C_i is the number of connections in S_i.

In Figure 4, we plot the FPR results as a function of number of packets in each subtrace. In this figure, we notice that the algorithm indeed has a very low FPR. For most of the cases, the FPR does not exceed 0.1%. For some cases with fewer packets, the FPR is slightly

[1] The *false positive rate* refers to the fraction of negative instances that were falsely reported by the algorithm as being positive.

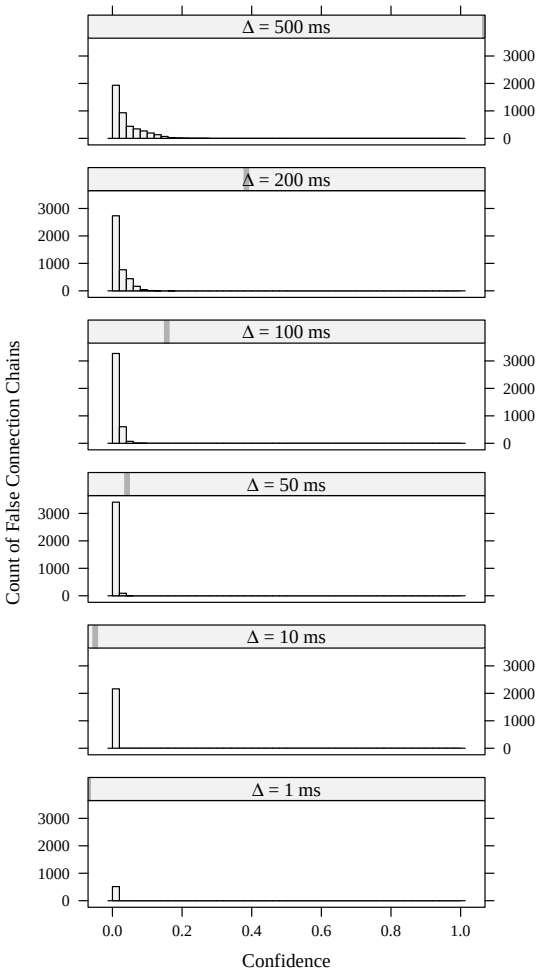


Figure 5: The confidences of false connection chains are shown as a set of histograms for different values of Δ. They approximately follow a decaying exponential distribution.

higher (< 0.7%). In the figure, we also notice that varying Δ does affect the FPR. In general, increasing Δ increases FPR. This is actually expected because increasing Δ would increase a connection's buffering time in $inSet$. Consequently, a connection would have greater chance to be correlated with other connections. However, notice that the FPR is still very low even when Δ = 500 msec (i.e. 5 times the ideal value).

Next, we studied the confidences of those false positives. For this purpose, we picked the largest subtrace among the 320 subtraces. The subtrace contains 1.7 million packets and 3391 connections, comprising traffic exchanged with server 131.243.2.12. We run the algorithm with different values of Δ; namely 1, 10, 50, 100, 200, and 500 msec. We then recorded the detected false positives and their confidences.

In Figure 5, the distribution of the confidences are plotted as histograms. Here, we noticed that these confidences exhibit a similar distribution. They approximately follow a decaying exponential distribution. In other words, the majority of them have confidences close to zero, while few have larger confidences. In fact, this is a desirable feature because the majority of false connection chains can be eliminated using a low $minconf$ threshold. Recall that $minconf$ is set to 0 in our experimentation. Therefore, the algorithm actually detected connection chains regardless of their confidences.

6.3 True Positives Analysis

Another important aspect of a detection algorithm is to have a high *true positive rate*[2] (i.e. detects all (or most) true instances). In this subsection, we investigate this aspect of the proposed algorithm. We also investigate the effect of varying Δ. Here, we simulated some connection chains, because the original trace does not contain true connection chains. In Section 7, we re-evaluate the algorithm using another dataset, which contains real connection chains (not simulated).

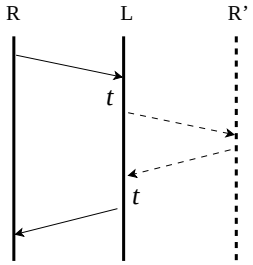


Figure 6: The process of simulating a connection chain. L, R and R' respectively stand for the local host, a remote host and a *fictitious* remote host. Original packets are shown as solid arrows, while the simulated ones are shown as dotted arrows.

The process used to simulate connection-chains is depicted in Figure 6. In this figure, L, R and R' respectively stand for the local host, a remote host and a *fictitious* remote host. The steps to create a simulated connection chain {R,R'} are as follows:

- For an inbound packet (R,L), create an outbound packet (L,R'). The time-stamp of the new packet is set to original time-stamp **plus** some random time t.
- For an outbound packet (L,R), create an inbound packet (R',L). The time-stamp of the new packet is set to original time-stamp **minus** some random time t.
- Merge those generated packets into the original trace.

Following the above steps, packets would seem to be flowing between the two remote hosts R and R' through

[2]The *true positive rate* refers to the fraction of true instances detected by the algorithm vs. all possible true instances

the local host L. In the figure, original packets are shown as solid arrows, while the simulated ones are shown as dotted arrows. Obviously, a table is maintained to keep a consistent mapping between hosts R and R'.

The experimentation was performed as follows. We first picked the largest subtrace[3]; i.e. the one we used earlier in the previous subsection. We then randomly picked 88 connections from the subtrace and applied the above simulation process. For the random time t, we used a uniform random variable between 10-90 msec (an estimate of the server response time). Recall that the original subtrace contains 3391 connections. Accordingly, the modified subtrace contains $3391 + 88 = 3479$ connections and $\binom{3479}{2} = 6049981$ possible connection chains. Only 88 out of the 6049981 possible connection chains are true connection chains ($\approx 0.002\%$). Those are the ones that we actually simulated.

The modified subtrace is then used as an input to the algorithm. As in the previous parts, different values of Δ were considered; namely 1, 10, 50, 100, 200, and 500 msec. For each case, we then recorded the detected connection chains and their confidences. We also computed the FPR according to Equation (2), and the true positive rate (TPR) as follows:

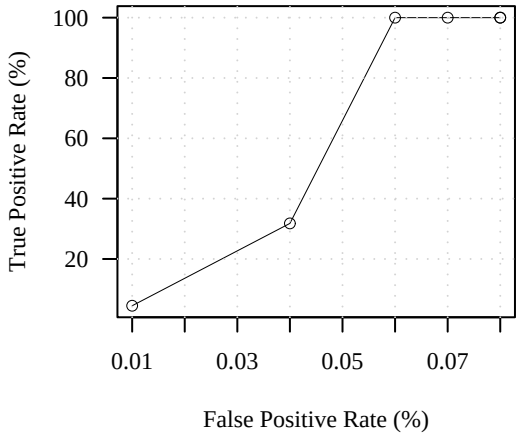
$$TPR = \frac{number\ of\ true\ positives}{number\ of\ possible\ true\ instances} \times 100,$$

where the number of possible true instances equals 88 (i.e. the number of simulated connection chains).

The TPR and FPR results are depicted as a ROC curve[4] in Figure 7. In this figure, it is apparent that Δ does affect both the FPR and TPR. As we noticed in the previous subsection, FPR generally increases as Δ increases. However, notice that the rate of this increase slows down as Δ gets larger than the ideal Δ (100 msec). For instance, increasing Δ from 100 to 200 msec results in less than 0.01% increase in FPR. Secondly, notice that setting Δ to a very low value can result in missing true connection chains (i.e. reduces the TPR). For instance, only 4.5% and 31.8% of true connection chains were detected when Δ is set to 1 and 10 msec respectively. Based on these two points, we conclude that it is safe to set Δ to a higher value than the true one in order to detect all true connection chains, especially that the increase in FPR is not significant.

Next, we investigated the confidences of the detected true and false connection chains. We summarized these confidences in Table 1. For each value of Δ, we list several statistics about the confidences of the true and false connection chains.

In general, notice that true connection chains have higher confidences. In Figure 8, the *ranges* (min-max) of confidences are visualized. For each value of Δ, a band



Δ (ms)	1	10	50	100	200	500
TPR(%)	4.5	31.8	100.0	100.0	100.0	100.0
FPR(%)	0.01	0.04	0.06	0.07	0.08	0.08

Figure 7: ROC curve showing how the TPR and FPR vary when different values are used for Δ, under worst case scenario ($minconf = 0$).

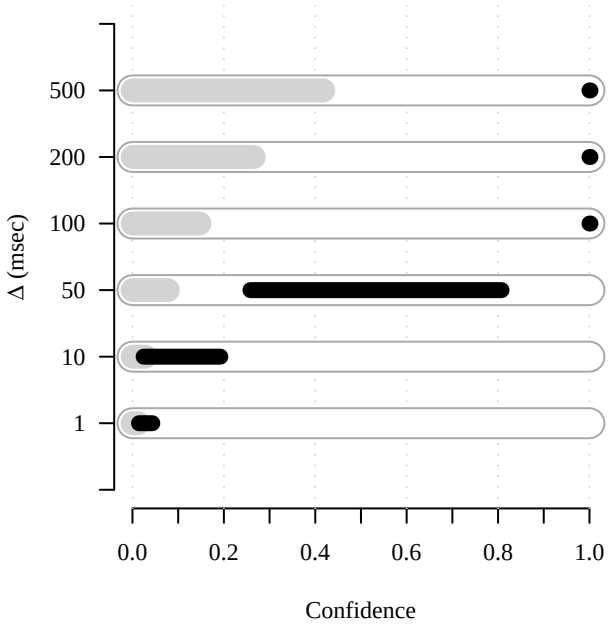


Figure 8: The ranges (min-max) of confidences of true and false connection chains for different values of Δ. For each value of Δ, a grey region indicates the range for false connection chains, while a black region indicates the range for true ones.

[3]This subtrace was chosen because of its large size, although other subtraces would give similar results.

[4]ROC curve stands for *Receiver Operating Characteristic* curve, a plot of the fraction of true positives vs. the fraction of false positives

Table 1: A summary of the algorithm's output showing confidence statistics for different values of Δ under worst case scenario: $minconf = 0$.

		Confidence					
		Min	1st Quartile	Median	Mean	3rd Quartile	Max
$\Delta = 1$ ms	True	0.01429	0.01857	0.02389	0.02639	0.0317	0.04348
	False	0.0002823	0.0007423	0.0009671	0.001242	0.00151	0.0122
$\Delta = 10$ ms	True	0.02439	0.03584	0.06797	0.07214	0.08378	0.1923
	False	0.0003401	0.001433	0.002322	0.002967	0.003913	0.02817
$\Delta = 50$ ms	True	0.2581	0.4756	0.4093	0.4929	0.5595	0.8077
	False	0.0003804	0.002959	0.006042	0.009131	0.01292	0.07726
$\Delta = 100$ ms	True	1.0	1.0	1.0	1.0	1.0	1.0
	False	0.0003623	0.004518	0.01006	0.01599	0.02237	0.1467
$\Delta = 200$ ms	True	1.0	1.0	1.0	1.0	1.0	1.0
	False	0.0003623	0.008181	0.01796	0.03009	0.04302	0.2653
$\Delta = 500$ ms	True	1.0	1.0	1.0	1.0	1.0	1.0
	False	0.0004968	0.01471	0.03562	0.05958	0.08803	0.4173

is shown that spans the range of all possible confidences. Inside each band, the grey region indicates the range for false connection chains, while the black region indicates the range for true connection chains.

In Figure 8, notice how the confidences of true and false connection chains overlap when Δ is set to very low values (1 and 10 msec). However, once Δ is set around or above the ideal value, true connection chains are clearly separated. In this case by appropriately setting the confidence threshold ($minconf$) in the separation area, we achieve perfect detection rates. For instance, for $\Delta = 100$ msec, by setting $minconf = 0.5$ we obtain TPR = 100% and FPR = 0%. Also, notice that increasing Δ beyond the ideal value decreases the *separation* between the confidences of the true and false connection chains. In this case, the maximum separation occurs at the ideal value of Δ (100 msec). However, notice that this separation is reasonably large even when $\Delta = 500$ msec (i.e. 5 times the ideal value). In essence, large separation is desirable because it gives greater flexibility in setting the $minconf$ threshold. Such threshold is used to reduce (or eliminate) false connection chains.

6.4 Processing Time Analysis

The proposed algorithm is intended for real-time processing of a live stream of packets. Therefore, we are interested in investigating the processing time per packet, and how does this time scales as a subtrace increases in size. We are also interested in studying how the processing time is affected by varying Δ.

The experimentation was performed as follows. For every subtrace (320 subtraces), we run the algorithm with a Δ of 1, 10, 50, 100, 200, and 500 msec (i.e. a total of $6 \times 320 = 1920$ cases). For a particular subtrace S_i, we then observed the processing time T_i. Although the original subtraces do not contain connection chains, the algorithm does detect false ones. This is because $mincof$ is set to 0 as we mentioned earlier. As such, the observed processing times actually account for all parts of the algorithm.

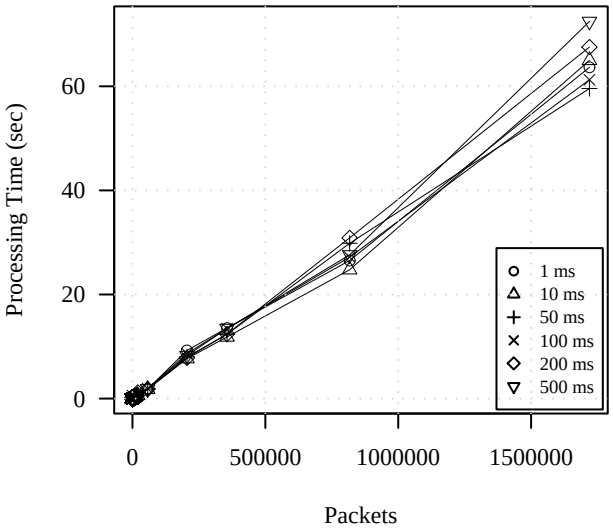


Figure 9: The processing time of the 320 subtraces sorted in increasing order of the number of packets. Note that the processing time exhibits a linear trend as subtraces increase in size, and also that varying Δ does not significantly impact on the processing time.

The observed processing times are plotted in Figure 9. In this figure, we notice the following points:

- The processing time exhibits a linear trend as subtraces increase in size.
- The processing time per packet is constant. It basically corresponds to the slope of the lines. Mathematically, it is given by $\frac{T_i}{|S_i|}$ $seconds/packet$, where

Table 2: The processing time of the largest subtrace with and without simulated connection chains.

pkt	cc	time (sec)	Δ (sec)	simulated
1719596	514	47.7	0.001	0
1719596	2164	51.9	0.010	0
1719596	3505	47.1	0.050	0
1719596	3953	48.2	0.100	0
1719596	4167	55.7	0.200	0
1719596	4373	53.3	0.500	0
1727775	526	47.2	0.001	1
1727775	2239	51.3	0.010	1
1727775	3698	53.1	0.050	1
1727775	4167	54.0	0.100	1
1727775	4398	55.3	0.200	1
1727775	4621	59.7	0.500	1

$|S_i|$ is the number of packets.

- For this trace, the average processing time per packet is about 35 μsec/packet.
- Finally, varying Δ does not seem to have a significant effect on the processing time.

Next, we investigated the processing time of the algorithm when true connection chains exist. For this purpose, we used the largest subtrace mentioned earlier. We run the algorithm using this subtrace with (and without) the simulated connection chains. As we did previously, Δ was varied using the following values: 1, 10, 50, 100, 200, and 500 msec. The results are shown in Table 2, where

- pkt: the number of packets processed.
- cc: the number of connection chains detected.
- time: the total processing time in seconds.
- delta: the value of delta used in seconds.
- simulated: 0 for original subtrace; 1 for original subtrace + simulated connection chains.

In this table, we notice that the processing time is not affected by the existence of true connection chains. Instead, the factor that really affects the processing time is the number of packets processed. Neither Δ nor the existence of true connection chains seems to play a big factor. As such, we conclude that the processing time results, obtained earlier in Figure 9, are valid whether true connection chains exist or not.

Based on the above observations, we conclude that the algorithm is actually efficient for real-time operation. This is because the average processing time per packet is both constant and low.

7 Evaluation Using Real Attack

In this section, we evaluate the proposed framework using a real attack that was posted as a forensic challenge by the Honeynet project [22]. The attack was presented as challenge#18 in a series of 34 challenges called *scan of the month challenges*.

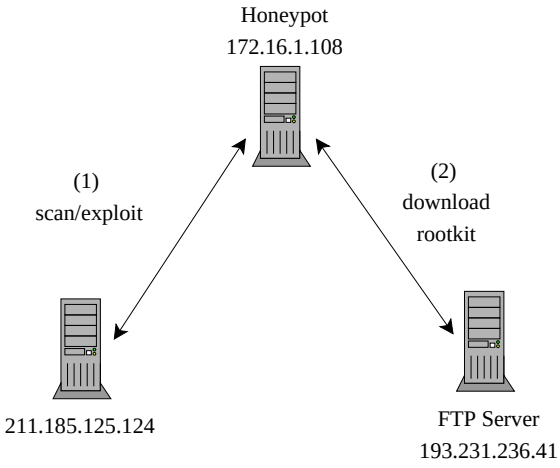


Figure 10: A diagram showing the IP addresses involved in the attack, and the attack steps.

7.1 Attack Overview

For this attack, Honeynet project provided a file of captured traffic to/from a honeypot. The file is in the standard *pcap* format [10]. In total, there were 993 packets communicated between the honeypot and 7 remote IP addresses. Among the 7 IP addresses, two IP addresses were involved in a successful attack against the honeypot. Specifically, one IP address was used to scan and compromise the honeypot, and one IP address was accessed to download a rootkit.

The steps of the attack and the involved IP addresses are shown in Figure 10. Briefly, the attack proceeded as follows:

1) 211.185.125.124 scanned the honeypot for rpc service (port 111), and confirms that *statd* is running. It then executed a successful exploit(rpc.statd) which bound a root shell at port (39168). It finally connected to the honeypot (via port 39168), and obtained a root shell.

2) Via the obtained shell, the attacker connected (ftp) to an ftp server on 193.231.236.41, and downloaded/installed a rootkit.

7.2 Results

Since the captured traffic contains 7 IP addresses, there are, technically, $\binom{7}{2} = 21$ *possible* connection chains. Among these 21 possible connection chains, there is only one *true* connection chain, which is depicted in Figure 10.

We run the algorithm on the original trace provided by Honeynet project. As in the previous section, we also

Table 3: The confidences of the true connection chain for different values of Δ.

$\Delta(ms)$	1	10	50	100	200	500
Confidence	-	0.3335	0.3666	0.406	0.4391	0.7644

vary Δ here, and consider the same values of Δ; namely 1, 10, 50, 100, 200 and 500 msec. Also, $minconf$ is set to 0 here in order to study all connection chains detected by the algorithm regardless of their confidences. Then, for every Δ, we record the detected connection chains and their corresponding confidences.

In this attack, the algorithm successfully revealed the true connection chain with a false positive rate (FPR) of 0%. Except for $\Delta = 1$ msec, the algorithm only detects one connection chain which is the true connection chain depicted in Figure 10. The confidences of this connection chain for different values of Δ are shown in Table 3. For $\Delta = 1$ msec, the algorithm does not detect any connection chains.

8 Evasions

In this section, we discuss several evasion techniques that might be used by an intelligent adversary to evade the proposed algorithm.

8.1 Chaffing

Given a stream of packets, chaffing refers to mixing the stream with fake packets which are then discarded by the recipient. This technique was originally introduced to achieve confidentiality without using encryption [20]. In particular, an eavesdropper will not be able to distinguish original packets from fake ones.

In the context of connection chains, an adversary may employ chaffing as an evasion technique. In particular, the added fake packets are designed to confuse a connection chain detection algorithm.

Chaffing is expected to mostly affect pure timing-based (network-based) approaches. For instance, consider the IPD algorithm proposed by Wang et al. [26]. This algorithm correlates connections based on the inter-packet delays (IPD); i.e. packets' inter-arrival times. It relies on the fact that IPDs are unique for each user, and preserved through a connection-chain. Clearly, injecting chaff packets will disturb the IPDs, and hence the algorithm would deliver poor TPR/FPR.

Chaffing evasion, however, has a serious limitation. In particular, interactive traffic has distinctive statistical characteristics [16]. Specifically, the majority of packets' inter-arrival times fits a Pareto distribution with shape parameter $\beta \approx 0.9$. Therefore, adding fake packets would disturb this unique statistical model [7]. Accordingly, a simple statistical monitor might be used to detect their presence.

Concerning our approach, chaffing does not affect the confidence measure (see Equation (1)). For illustration, consider the following example. Let $\{a, b\}$ be a connection chain through a host h, as shown below:

$$[\leftarrow a \rightarrow \boxed{\ h\ } \leftarrow b \rightarrow]$$

Assume a stream of packets $P = \langle p_1, p_2, p_3 \rangle$ flows from a to b. Recall that packets can not stay longer than Δ inside h. Otherwise, the connection chain will break. Specifically, a tcp connection will timeout and start retransmission or even disconnect. Accordingly, when the algorithm processes P, the connection chain $\{a, b\}$ will be detected with a confidence given by

$$[confidence(\{a, b\}) = \frac{\sigma([a, b])}{\sigma([a]) + \sigma([b])} = \frac{3}{3 + 0} = 1.0]$$

Now assume, the original stream is chaffed as follows: $\hat{P} = \langle p_1, c_1, p_2, c_2, p_3 \rangle$, where c_1 and c_2 are chaff packets. Since a packet can not stay longer than Δ inside h, packets (original or chaffed) arriving at a must be relaid to b within Δ. Therefore, when the algorithm processes $\hat{P}$, the connection chain $\{a, b\}$ will be detected with a confidence given by

$$[confidence(\{a, b\}) = \frac{\sigma([a, b])}{\sigma([a]) + \sigma([b])} = \frac{5}{5 + 0} = 1.0]$$

As shown in the above example, the confidence measure is not affected whether chaff packets exist or not.

8.2 Adding Delay

Instead of adding fake packets, an adversary may employ another evasion technique to confuse a connection chain detection algorithm. In particular, he/she may add random delay to packets' arrival-times.

Similar to chaffing, this evasion technique is problematic for pure timing-based (network-based) approaches, which use absolute packets' arrival-times. For instance, consider the ON/OFF algorithm proposed by Zhang and Paxson [32]. This algorithm uses packets' arrival-times to detect pairs of connections that exhibit coincident OFF to ON transitions. Clearly, delaying packets will disturb packets arrival-times, and the algorithm in turn.

Similar to chaffing, delaying packets also suffers from the same limitation. Specifically, delaying packets will disturb the unique statistical characteristics of interactive traffic (i.e. its adherence to the Pareto statistical distribution).

Concerning our approach, adding delay to packets' arrival-times does not affect the detection algorithm. This is because the algorithm does not use absolute arrival-times. Instead, packets are matched within a Δ time interval. Given that packets can not be delayed more than Δ inside a host as mentioned in the previous section. The packets will be matched even though their arrival-times have been changed. In some respects, the buffer (inSet)

used by the algorithm, filters out any jitters or small delays.

There is however a subtle issue here. Initially, we do not know the value of Δ. Therefore, we will be using an estimate of Δ in the algorithm. Fortunately, the algorithm is not very sensitive to Δ. We can set Δ higher (even 5 times) than the true Δ, and still get 100% TPR and very low FPR. This was demonstrated by our experimentations.

8.3 Payload Encoding

In addition to the mentioned evasion techniques, an adversary may attempt to alter packets' payloads using some form of transformation. Possible transformations include compression, encryption, re-packeting, etc. Collectively, we refer to these transformations as payload encoding.

Payload encoding is effective against any detection algorithm that depends on packets' payloads. Examples of such algorithms include text matching of packets' payloads [32] and characters frequency analysis (thumbprints) [6]. For our algorithm, this evasion is not applicable, because packets' payloads are not used in the analysis.

9 Conclusion

A connection chain refers to a mechanism in which someone recursively logs into a host, then from there logs into another host, and so on. In this paper, we proposed a host-based algorithm to detect connection chains by passively monitoring inbound and outbound packets. From a host perspective, a connection chain appears as a pair of connections through which packets pass back and forth. We took advantage of the fact that the time taken by a packet inside the host has to be bounded for a connection chain to work. We refer to this time as Δ. As such, we employed concepts from association rule mining in the data mining literature. In particular, we proposed efficient algorithm and data structures to discover connection chains among a set of connections. Also, a confidence measure is used to attest the likelihood of a connection chain.

We used public network traces to assess the algorithm in terms of processing time and detection capabilities. For processing time, our experimentations suggest that the algorithm is efficient for real-time operation. It has a constant and low average processing time per packet. In terms of detection capabilities, our experimentations suggest that the algorithm is effective in detecting true connection chains. In particular, the algorithm has a very low false positive rate (FPR). In our experimentation, the FPR does not exceed 0.1% for most of the cases. We also found that the setting of Δ seems to play an important role. We found that it is always safe to set Δ to a higher value than the true value. This ensures the detection of all true connection chains, while the increase in false connection chains is not significant. Also, our experimentations showed some desirable features of the algorithm. In particular, the majority of false connection chains have confidences close to zero, while few have larger confidences. This means that the majority of false connection chains can be eliminated using a low confidence threshold. Also, we found that the confidences of true and false connection chains are clearly separated when Δ is set around or above (even 5 times) the true value. This also gives greater flexibility in setting a confidence threshold to reduce (or eliminate) false connection chains.

References

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Rec.*, vol. 22, no. 2, pp. 207-216, 1993.

[2] A. Almulhem, and I. Traore, *Connection-chains: A Review and Taxonomy*, ECE Department, University of Victoria, Techical Report ECE-05.4, Dec. 2005.

[3] A. Blum, D. Song, and S. Venkataraman, "Detection of interactive stepping stones: Algorithms and confidence bounds," LNCS 3224, pp. 258-277, Springer-Verlag, Jan. 2004.

[4] F. Buchholz, and C. Shields, "Providing process origin information to aid in network traceback," *Proceedings of the 2002 USENIX Annual Technical Conference*, pp. 261-274, 2002.

[5] B. Carrier, and C. Shields, "The session token protocol for forensics and traceback," *ACM Transactions on Information System Security*, vol. 7, no. 3, pp. 333-362, 2004.

[6] S. S. Chen, and L. T. Heberlein, "Holding intruders accountable on the internet," *Proceedings of IEEE Symposium on Security and Privacy*, pp. 39-49, May 1995.

[7] D. L. Donoho, A. G. Flesia, U. Shankar, V. Paxson, J. Coit, and S. Staniford, "Multiscale stepping-stone detection: Detecting pairs of jittered interactive streams by exploiting maximum tolerable delay," *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection*, pp. 17-35, Oct. 2002.

[8] K. Fujii, *Jpcap: Java Package for Packet Capture*, 2008. (http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html)

[9] T. He, and L. Tong, "A signal processing prospective to stepping-stone detection," *Conference on Information Sciences and Systems (CISS' 06)*, pp. 687-692, Princeton, NJ, 2006.

[10] V. Jacobson, C. Leres, and S. McCanne, *Packet Capture Library*, 2008. (http://www.tcpdump.org/)

[11] H. T. Jung, H. L. Kim, Y. M. Seo, G. Choe, S. Min, and C. S. Kim, "Caller identification system in the internet environment," *Proceedings of UNIX Security Symposium IV*, pp. 69-78, Santa Clara, California, Oct. 1993.

[12] H. W. Kang, S. J. Hong, and D. H. Lee, "Matching connection pairs," LNCS 3320, pp. 642-649, Springer-Verlag, Jan. 2004.
[13] B. Kantor, *BSD rlogin*, RFC 1282, Dec. 1991.
[14] C. Lonvick, *SSH Protocol Architecture*, Cisco Systems, Inc., Dec. 2004.
[15] R. Pang, and V. Paxson, *Lbnl-ftp-pkt*, 2008. (http://www-nrg.ee.lbl.gov/anonymized-traces.html)
[16] V. Paxson, and S. Floyd, "Wide area traffic: the failure of poisson modeling," *IEEE/ACM Transactions on Network*, vol. 3, no. 3, pp. 226-244, 1995.
[17] P. Peng, P. Ning, and D. S. Reeves, "On the secrecy of timing-based active watermarking trace-back techniques," *IEEE Symposium on Security and Privacy*, pp. 334-349, 2006.
[18] J. Postel, *Transmission Control Protocol*, RFC 793, Sep. 1981.
[19] J. Postel, and J. Reynolds, *Telnet Protocol Specification*, RFC 854, May 1983.
[20] R. L. Rivest, *Chaffing and Winnowing: Confidentiality without Encryption*, MIT Lab for Computer Science, Mar. 18, 1998. (http://theory.lcs.mit.edu/~rivest/ chaffing.txt)
[21] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. lin Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur, "DIDS (Distributed Intrusion Detection System) - motivation, architecture, and an early prototype," *Proceedings of the 14th National Computer Security Conference*, pp. 167-176, Washington, DC, 1991. (http://citeseer.ist.psu.edu/snapp91dids.html)
[22] L. Spitzner, *The Honeynet Project*, 2008. (http://www.honeynet.org)
[23] SUN Inc., *The J2SE Development Kit*, Last visited: May 26, 2007. (http://java.sun.com)
[24] P. N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*, Addison-Wesley, 2005.
[25] X. Wang, D. S. Reeves, S. F. Wu, and J. Yuill, "Sleepy watermark tracing: an active network-based intrusion response framework," *Proceedings of the 16th international conference on Information security: Trusted information*, pp. 369-384, 2001.
[26] X. Wang, D. S. Reeves, and S. F. Wu, "Inter-packet delay based correlation for tracing encrypted connections through stepping stones," *Proceedings of the 7th European Symposium on Research in Computer Security*, pp. 244-263, Springer-Verlag, London, UK, 2002.
[27] X. Wang, and D. S. Reeves, "Robust correlation of encrypted attack traffic through stepping stones by manipulation of interpacket delays," *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pp. 20-29, ACM Press, New York, NY, USA, 2003.
[28] J. Yang, and S. H. S. Huang, "A real-time algorithm to detect long connection chains of interactive terminal sessions," *Proceedings of the 3rd international conference on Information Security*, pp. 198-203, ACM Press, New York, NY, USA, 2004.
[29] J. Yang, and S. H. Huang, "Matching tcp packets and its application to the detection of long connection chains on the internet," *19th International Conference on Advanced Information Networking and Applications*, pp. 1005-1010, Mar. 2005.
[30] K. Yoda, and H. Etoh, "Finding a connection chain for tracing intruders," *Proceedings of the 6th European Symposium on Research in Computer Security*, pp. 191-205, Springer-Verlag, London, UK, 2000.
[31] K. H. Yung, "Detecting long connection chains of interactive terminal sessions," *RAID 2002*, LNCS 2516, pp. 1-16, Springer-Verlag, Jan. 2002.
[32] Y. Zhang, and V. Paxson, "Detecting stepping stones," *9th USENIX Security Symposium*, pp. 171-184, Aug. 2000.

Ahmad Almulhem is a researcher in the area of network security with special interest in network forensics. He received his Ph.D. in Nov. 2007, from the department of electrical and computer engineering at the University of Victoria, Canada. He is currently an assistant professor in the department of computer engineering at King Fahd University of Petroleum and Minerals, Saudi Arabia.

Issa Traore received an Aircraft Engineer degree from Ecole de l'Air in Salon de Provence (France) in 1990, and successively two Master degrees in Aeronautics and Space Techniques in 1994, and in Automatics and Computer Engineering in 1995 from Ecole Nationale Superieure de l'Aeronautique et de l'Espace (E.N.S.A.E), Toulouse, France. In 1998, Traore received a Ph.D. in Software Engineering from Institute Nationale Polytechnique (INPT)-LAAS/CNRS, Tou-louse, France. From June - Oct. 1998, he held a post-doc position at LAAS-CNRS, Toulouse, France, and Research Associate (Nov. 1998 - May 1999), and Senior Lecturer (June-Oct. 1999) at the University of Oslo. Since Nov. 1999, he has joined the faculty of the Department of ECE, University of Victoria, Canada. He is currently an Associate Professor and held from Oct. 2003 to Dec. 2007 the position of Computer Engineering Programme Director. His research interests include Behavioral biometrics systems, intrusion detection systems, software security metrics, and software quality engineering. He is the founder and coordinator of the Information Security and object Technology (ISOT) Lab (http://www.isot.ece.uvic.ca).