Detecting Distributed Signature-based Intrusion: The Case of Multi-Path Routing Attacks

Paper #1732

***Abstract*—Signature-based network intrusion detection systems (S-IDS) have become an important security tool in the protection of an organization's infrastructure against external intruders. By analyzing network traffic, S-IDS' detect network intrusions. An organization may deploy one or multiple S-IDS', and each of them works independently with the assumption that it can monitor all packets of a given flow to detect intrusion signatures. However, emerging technologies (e.g., Multi-Path TCP) violate this assumption, as traffic can be concurrently sent across all available paths (e.g., WiFi, Cellular, etc.) to boost the end-users' network performances. Attackers may exploit this capability and split malicious payloads across multiple paths to evade traditional signature-based network intrusion detection systems. Although multiple monitors may be deployed, none of them has the full coverage of the network traffic to detect the intrusion signature. In this paper, we formalise this distributed signature-based intrusion detection problem as an asynchronous online exact string matching problem, and propose an algorithm for it. To demonstrate the effectiveness of our proposal we have implemented our algorithm and conducted comprehensive experiments. Our experimental results show that the behaviour of our algorithm depends only on the packet arrival rate: delay in detecting the signature grows linearly with respect to the packet arrival rate and with small overhead for monitor communications.**

I. Introduction

MPTCP is a new set of IETF standardized extensions to TCP [1] that allows end points to simultaneously use multiple paths between them to improve network performance. This new capability has sparked a lot of interest from both academia and industry, especially considering that end devices (e.g., smartphones) commonly support multiple access technologies (e.g., WiFi, 4G), and early empirical studies [2], [3] have demonstrated that MPTCP could significantly increase end users' throughput. To date, both Apple iOS and Google Android support MPTCP. This technology not only improves network performance over the access network (i.e., WiFi, cellular network), and facilitates user mobility [4], but is also beneficial to Cloud Service Providers to take advantage of their rich connectivity to the Internet, and increasing large number of peering links. While the benefits of MPTCP are clear, its security is still being analyzed [5], [6]. In particular, the capability of splitting traffic across multiple paths opens new venues for sophisticated attacks that can evade traditional intrusion detection systems.

Signature-based network intrusion detection systems (S-IDS) have become an important security tool in the protection of an organization's infrastructure against external intruders. By analyzing network traffic, S-IDSs detect network intrusions. An organization may deploy one or multiple S-IDS', and each of them works independently with the assumption that it can monitor all packets of a given flow to detect intrusion signatures without communicating with each other. However, attackers may exploit the multiplicity of paths and split malicious payloads across multiple paths to evade traditional signature-based network intrusion detection systems. Although multiple monitors may be deployed, none of them has the full coverage of the network traffic to detect the intrusion signature. We have easily recreated an attack that evades detection by the popular open-source S-IDS Snort [7].

In this paper, we formalise this multipath signature-based intrusion detection problem as an asynchronous online exact string matching problem, and propose an algorithm for it based on the Aho-Corasick matching algorithm [8]. As with Aho-Corasick's the time complexity of our algorithm for scanning the whole input string is linear with respect to the size of the input string. Our proposed solution relies on: (1) an automaton running on each monitor, for each partially observed input string and (2) asynchronous communication among the monitors. The overhead of the communication is small since information exchanged is merely automaton states. To demonstrate the effectiveness of our proposal we have implemented our algorithm, and conducted a comprehensive set of experiments to find signatures in MPTCP traffic. Our experiments show that the behaviour of the proposed algorithm is independent of factors such as size and number of MPTCP connections, number of signatures in the flows or in the monitors database. *It is only the packet arrival rate at the monitors that matters*. Delays in detecting the signature grows linearly with respect to the packet arrival rate. In absolute terms, with our straightforward prototype, for a network throughput of 450Mbps, most delays (i.e., time to detect the signature) are about 200 microseconds, and less than 400 microseconds. A second important component is the amount of communication traffic generated between the monitors. In the unlikely scenario that a monitor needs to communicate states to other monitors for each received packet, the size of the messages in our simple implementation is 52 bytes including the message headers. In practice, the communication ratio (the number of monitor communication packets over the number of data packets) we observed varied from 45% to 15% decreasing as the throughput increases. Hence, a reliable network connection between monitors is needed but with little cost since the throughput required is relatively small.

II. Background

A. Network Intrusion Detection System

Signature-based network intrusion detection systems (S-IDS) analyze network traffic, and compare packets against a database of signatures from known malicious threats. S-IDS are commonly classified into active versus passive S-IDS. Active S-IDS can drop packets and halt an attack in progress. In contrast, passive S-IDS mainly raise alarms, and rely on humans to take subsequent actions. In this paper, we focus on the passive approach, considering that many commercial IDS' are solely passive []. We discuss how extensions for active S-IDS can be made in Section VII.

S-IDSs apply locally configured rules to each packet. For example, Snort [7] rules consist of two main parts: the rule header, and the rule options. The rule header specifies the action (*pass, drop, alert, log*), protocol, IP addresses, and port numbers, whereas the rule option section specifies the alert message, and information about which parts of the packet should be inspected to determine if the rule action should be taken. The following illustrates a Snort rule. The first line consists of the rule header, and the second line specifies the rule options.

```
alert tcp any any -> 192.168.1.0/24 111 \
(content:"|00 01 86 a5|"; msg:"mountd access";)
```

The rule indicates that any TCP packet sent to any destination in the subnet 192.168.1.0/24, and to destination port 111, with the exact string "—00 01 86 a5—" in the packet payload should trigger an alert with the message "mountd access". The Snort signature database currently consist of about 4000 rules.

In addition to the detection engine, network intrusion detection systems support preprocessor modules with the main goal of re-assembling IP fragments, or TCP segments. The preprocessor modules are applied before the detection engine, and address attacks that span multiple IP packets, rely on overlapping data, or exploit TCP anomalies [9].

B. Multi-Path TCP

Multi-Path TCP is a new transport protocol that allows two endpoints to simultaneously use multiple paths available between them. It is in fact defined as a set of extensions to TCP to retain compatibility with and allow traversal of middleboxes such as firewalls, NATs, and performance enhancing proxies. As a notable feature, MPTCP introduces a 64-bit data sequence number (DSN) to number all data sent over the MPTCP connection. This allows the sender to retransmit data on different sub-flows, and for the receiver to still successfully re-order the received bytes over the different paths.

III. New threat

This section describes how through MPTCP, attackers could evade traditional signature-based network intrusion detection systems. First, to illustrate the attack let us assume the network depicted in Figure 1 consisting of a client (victim) and a server (attacker). We further assume that the network where the client resides deploys a signature-based network intrusion detection system at each ingress point to monitor and analyses

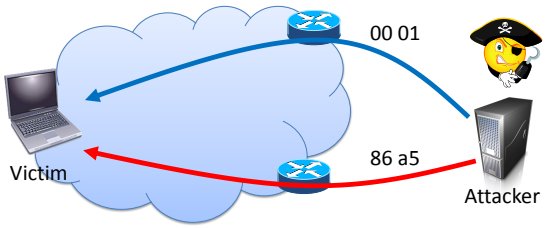

Fig. 1. New threat: With multi-path routing, attackers could evade traditional signature-based network intrusion detection systems by splitting the signature (e.g., "00 01 86 a5") over different paths.

all traffic between its users and the Internet. Continuing the example of Section II-A, we focus on the signature "00 01 86 a5". We assume the client and server have established a MPTCP connection, composed of two flows. Each flow enters the network through a different ingress point, and therefore traverses a different network intrusion detection system.

The attacker can evade detection by splitting the signature into multiple pieces (e.g., "00 01", "86 a5"), and sending them over the different flows. Because each signature-based network intrusion detection system receives only a fraction of the signature (e.g., "00 01"), neither monitor can detect the attack. However, the client receiving all the bytes, gets compromised. We have verified and confirmed the attack, using the open-source S-IDS Snort.

We refer to this threat, as the multi-path signature detection (MPSD) problem, and formulate it as follows:

- A source (i.e., the *attacker*) sends a data stream containing malicious segments to the destination (i.e., the *victim*) via MPTCP;
- Each MPTCP sub-flow is intercepted by one host called *monitor* that runs an IDS;
- The data stream is divided into packets, each of which is associated with a sequence number;
- The source can select, delay and duplicate packets sent to the different paths;
- Monitors are fully connected and have a list of malicious data segment patterns (i.e., signatures).

We abstract the MPSD problem as an *asynchronous online exact string matching* problem with the following definitions. A *string* is a finite sequence of symbols from a given alphabet. An *annotated symbol* is s^k where s is a symbol from the given alphabet and k is a positive integer associated with s. An *annotated string* is a finite sequence of annotated symbols. Let $\tau = s_1 s_2 \cdots s_n$ be a string of length n, then $seqNo(\tau, s_i)$ denotes the sequence number of a symbol s_i in τ, i.e., $seqNo(\tau, s_i) = i$, and $assoc(\tau)$ denotes an annotated string obtained by associating each symbol in τ with its sequence number, i.e., $assoc(\tau) = s_1^{seqNo(\tau,s_1)} s_2^{seqNo(\tau,s_2)} \cdots s_n^{seqNo(\tau,s_n)} = s_1^1 s_2^2 \cdots s_n^n$. We use $symSet(\phi)$ to denote the set of annotated symbols of ϕ, i.e., $symSet(assoc(\tau)) = \{s_1^1, s_2^2, \cdots, s_n^n\}$.

Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be a finite set of strings, which we shall call the *keywords* (i.e., it represents the set of signatures), and x be an arbitrary string, we shall call it the *text* (i.e.,

it represents the data stream). Let $\mathcal{O} = \{o_1, \ldots, o_m\}$ be an arbitrary set of m arbitrary annotated strings, we shall call them the *observed texts*, such that $symSet(assoc(x)) = \bigcup_{o_i \in \mathcal{O}} symSet(o_i)$ (i.e., each annotated symbol represents a packet). For example, let $xyabcz$ be the text, then $x^1y^2a^3b^4c^5z^6$ is an annotated text, and $\{x^1y^2c^5a^3, y^2b^4z^6\}$ is a possible set of two observed texts, in which a^3 is *delayed* after c^5, and y^2 is *duplicated*. Finally, a set of *monitors* $\{\alpha_1, \ldots, \alpha_m\}$ is a set of m network nodes that can cooperatively find all the occurrences of any keyword in $\mathcal{P}$ from x while each α_i has only one observed text o_i.

IV. Proposed Algorithm

A straw man proposal to the MPSD problem could rely on a centralized approach, and have all the monitors select a leader to act as the repository. Non-leader monitors forward all traffic they observe to the leader. Then, the leader which obtains full network traffic information can perform the signature detection locally. However, there are two major limitations: first, the total traffic volume will double due to the inter-monitor communications. Second, the leader monitor can be overloaded.

To address these limitations, we propose a fully distributed solution where each monitor locally scans and processes its monitored traffic. To prevent attacks that may split signatures across multiple paths, we have monitors coordinate their actions, and exchange states. One important objective is to keep the volume of inter-monitor communication low. To achieve it, we have developed a new distributed algorithm, based on the Aho-Corasick [8] automaton-based string matching algorithm. The main idea consists in having all monitors share asynchronously a *global state* of the string matching automaton for each MPTCP connection. Each monitor receives "segments" of the data stream (i.e., locally observed traffic belonging to the same MPTCP connection), scans the received segment locally, and broadcasts to other monitors the latest automaton state as well as the segment's relative position in the data stream. The monitors update their local scans through the received states. As such, the local scans of segments resemble a global scan of the whole data stream. In the remainder of this section, we first briefly introduce the Aho-Corasick algorithm, and then describe our distributed algorithm in detail.

A. Aho-Corasick Algorithm

The Aho-Corasick algorithm [8] is an automaton-based string matching algorithm that has been widely used in network intrusion detection systems such as Snort [7]. The main advantage of the Aho-Corasick algorithm, comparing to other string matching algorithms (e.g. the Boyer-Moore pattern matching algorithm [10]), is that it can scan for multiple signatures at the same time and has time complexity of $O(n)$ where n is the size of the string to be scanned. Let Σ be the alphabet from which the signatures and strings are formed. Given a set of signatures, the Aho-Corasick algorithm computes a single deterministic automaton $\langle S, s_0, O, \Sigma, \delta \rangle$ and the *output function* σ, where $S = \{s_0, s_1, \ldots, s_n\}$ is the set

$\mathbf{state_{cur}}$	0			1					{2, 5}				{3, 7, 9}		
symbol	h	s	.	e	i	h	s	.	r	h	s	.	h	s	.
$\mathbf{state_{new}}$	1	3	0	2	6	1	3	0	8	1	3	0	4	3	0

$\mathbf{state_{cur}}$	4					6			8		
symbol	e	i	h	s	.	s	h	.	s	h	.
$\mathbf{state_{new}}$	5	6	1	3	0	7	1	0	9	1	0

$\mathbf{state_{accepting}}$	2	5	7	9
$\sigma(\mathbf{state_{accepting}})$	$\{he\}$	$\{she, he\}$	$\{his\}$	$\{hers\}$

Fig. 2. Deterministic automaton for signatures $\{he, she, his, hers\}$: (1) 0 is the initial state; (2) "." means any symbol in Σ but not mentioned in the column for a state.

of states, s_0 is the initial state, O is the set of accepting states and δ is the transition function that takes as input a state s_i from S and a symbol c from Σ and gives as output a new state s_{i+1}. The output function σ takes as input an accepting state s from O and gives as output the set of signatures detected. For example, let $\{he, she, his, hers\}$ be a set of signatures, the computed automaton, with the state transitions and the output function are shown in Figure 2.

Given a string, the algorithm scans its characters from the beginning to the end only once, starting with the initial state and using the characters to trigger state transitions. If any accepting state is reached, the output (detected) signatures can be recorded, and the algorithm may continue until the whole string is scanned. For example, let the string be "$ushers$", the sequence of state transitions would be $0 \xrightarrow{u} 0 \xrightarrow{s} 3 \xrightarrow{h} 4 \xrightarrow{e} 5 \xrightarrow{r} 8 \xrightarrow{s} 9$, where signatures $\{she, he\}$ are detected after the 4^{th} character and signature $\{hers\}$ is detected after the last character. As such, the Aho-Corasick algorithm allows one to identify all the matching signatures in a given string.

B. Multi-path Signature Detection Algorithm

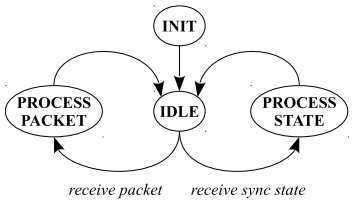

Fig. 3. MPSD Algorithm States

1) Overview: Our proposed algorithm is executed asynchronously at each monitor, and consists of four execution states as illustrated in Figure 3. First, a monitor starts at the INIT state, where it creates a single automaton using the Aho-Corasick algorithm for all the signatures in the database. Then, it moves to the IDLE state, where it waits for two types of information: captured packets (called *data packets*) from the network, and synchronization messages (called *sync states*) sent by other monitors. Every data packet causes the monitor to enter the PROCESS_PACKET state, whose actions are described in Algorithm 1. Similarly, Every synchronization message causes the monitor to enter the PROCESS_STATE state, whose actions are described in Algorithm 2. In both

execution states, the packet payload scanning procedure may be invoked. This procedure is described in Algorithm 3.

To describe the different procedures in details, we introduce three abstractions comprising a *data packet*, a *flow state* and a *sync state*:

- A *data packet*, denoted with <mid, seqno, payload, type>, contains the multipath connection identifier, the sequence number in the connection data flow (not the sub-flow's sequence number), and the content (e.g., sequence of characters). A data packet can be one of three types – START, END and DATA – marking the initiation, the termination and the data transfer of a MPTCP connection.
- A *flow state* is created and maintained by a monitor for each intercepted multipath connection. It is denoted with <mid, cur_seqno, cur_state, fin_seqno, packets>, where cur_seqno is a sequence number indicating the next character in the connection to be processed, cur_state is the latest automaton state recorded locally, fin_seqno is the final sequence number of the connection, and packets is a priority queue that stores packets in ascending order based on the sequence numbers and contains no duplicates.
- A *sync state* is the means for monitors to synchronise their local flow states. Monitors will exchange *sync states* to synchronise. It is denoted with <mid, latest_seqno, latest_state>, where latest_seqno and latest_state are the latest sequence number and automaton state recorded by the sender monitor for the given connection.

Algorithm 1 Process a received packet

```
1:  procedure ProcessPacket(p : Packet)
2:      if p.mid first seen then
3:          fs ← <p.mid, −∞, 0, +∞, ∅>
4:          store fs
5:      else
6:          fetch fs where fs.mid = p.mid
7:                                                  ▷ Placeholder
8:      if p.type = START then
9:          fs.cur_seqno ← p.seqno +1
10:         s ← <fs.mid, fs.cur_seqno, fs.cur_state>
11:         Broadcast(s)                            ▷ Communication
12:     else if p.type = END then
13:         fs.fin_seqno ← p.seqno
14:     if p.seqno ≥ fs.cur_seqno then
15:         enqueue p to fs.packets
16:     else
17:         discard p
18:     ScanIfRequired(fs)
```

2) Handling A Received Packet: In Algorithm 1, lines 2–6 retrieve the corresponding flow state fs given a multipath connection id associated with the packet, or create (and store) a new flow state if none yet exists. When a new flow state is created, its cur_seqno (resp. fin_seqno) is set to $-\infty$ (resp. $+\infty$) indicating that the starting (resp. final) sequence number is unknown, and its cur_state is set to the initial automaton state (i.e., 0). Lines 8–13 record the first and the final sequence numbers of the multipath connection in the flow state. As it is reflected in Algorithm 2 and Algorithm 3, the sequence number cur_seqno recorded in the flow state marks the position of the next character in the whole multipath connection to be scanned (by any monitor). Thus, the incremented sequence number of a START packet will replace $-\infty$ and it will be equal to the sequence number of the first DATA data packet in the connection (line 9). Note that the START packet and the first data packet may be received by different monitors (i.e., sent down different sub-flows). Therefore, the monitor needs to send out the first sync state (lines 10–11) to tell others about the first data sequence number. Similarly, the sequence number of an END packet is memorised (line 13) in the flow state for future termination of the sub-flow scans (see Algorithm 3). Lines 14–17 continue to handle the packet based on its sequence number. If the packet's sequence number is greater than or equal to that recorded in the flow state (which is always the case for the END packet but not the case for the START packet), the packet is enqueued to the buffer for further processing. Otherwise, it must be a packet that has been scanned by at least one of the monitors in the past (i.e., it is a duplicate packet), and hence is simply discarded. Note that during the enqueue operation (line 15) if the buffer has already had a packet with the same sequence number, then the new packet must also be a duplicate and is discarded too. Finally, the ScanIfRequired procedure is called and the packet scanning process may or may not be triggered depending on further comparison of sequence numbers, as described in Algorithm 3.

Algorithm 2 Process a received sync state

```
1:  procedure ProcessState(s : Sync State)
2:      if s.mid first seen then
3:          fs ← <s.mid, −∞, 0, +∞, ∅>
4:          store fs
5:      else
6:          fetch fs where fs.mid = s.mid
7:                                                  ▷ Placeholder
8:      if s.latest_state = ∞ then
9:          remove fs
10:     else
11:         if s.latest_seqno > fs.cur_seqno then
12:             fs.cur_seqno ← s.latest_seqno
13:             fs.cur_state ← s.latest_state
14:             ScanIfRequired(fs)
15:         else
16:             discard s
```

3) Handling A Received Sync State: In Algorithm 2, lines 2–6 obtain the flow state fs of interest given the multipath connection id associated with the received sync state s. If the latest automaton state in s is ∞, which is a signal by the sender monitor announcing that it has scanned the last data packet of the whole multipath connection, then the receiver monitor needs to remove the current flow state in order to free resources. Otherwise, s' sequence number is compared with fs', and there are two cases. If s has a bigger sequence number (line 11), it implies that the sender monitor has performed scan on some packets it received and has contributed to the scanning progress of the whole multipath

connection. In this case, the receiver monitor needs to "catch up", by recording the latest sequence number and automaton state from `s` (lines 12–13). Furthermore, SCANIFREQUIRED is called so that the receiver monitor can try to progress the scanning using its locally buffered packets. In the case where `s` has a smaller or equal sequence number (line 3), `s` is an *out of date* state and can be simply discarded. Such situation may arise after two or more monitors receive and process duplicates of some packet independently and simultaneously.

Algorithm 3 Process the packet buffer of a subflow

```
1: procedure SCANIFREQUIRED(fs : Flow State)
2:     while fs.packets.head.mid < fs.cur_seqno do
3:         dequeue fs.packet_buf
4:     while fs.packets.head.mid = fs.cur_seqno do
5:         p ← dequeue fs.packets
6:         for all each character c in p.payload do
7:             fs.cur_state ← NEXTSTATE(fs.cur_state, c)
8:             fs.cur_seqno ← fs.cur_seqno +1
9:             if fs.cur_state is an accepting state then
10:                record OUTPUT(fs.cur_state)
11:    if fs.cur_seqno = fs.fin_seqno then
12:        fs.cur_state ← ∞
13:    if fs.cur_seqno has changed value or fs.cur_state has
       become ∞ then
14:        s ← <fs.mid, fs.cur_seqno, fs.cur_state>
15:        BROADCAST(s)                          ▷ Communication
16:    if fs.cur_state = ∞ then
17:        remove fs
```

4) Packet Payload Scanning: Algorithm 3 describes the main procedure for packet scanning. Given a flow state `fs`, it first (line 2–3) removes any out-of-date packets in the buffer (i.e., packets with sequence numbers smaller than the current sequence number recorded by `fs`). A buffered packet at a monitor becomes out-of-date if its duplicate is received and scanned by another monitor. In this case, the current monitor must receive a sync state with a bigger sequence number, which triggers the PROCESSSTATE procedure and in turn the current procedure. Next (lines 4–10), if the buffer is not empty, and its head packet's sequence number is equal to the one currently recorded by `fs`, it means that the current monitor has the next data packet in the multipath connection to resume the scanning process. Thus, the head packet is dequeued and scanned, and any detected patterns will be stored (as alerts). This step is repeated until the buffer becomes empty or the head packet has a bigger sequence number than `fs`', i.e., the current monitor tries to advance in the scanning process as much as possible. Finally, if the last data packet of the multipath connection has been scanned, then the latest automaton state in `fs` is replaced with ∞ (lines 11-12), before the `fs` is removed (lines 16–17). In addition, if either the sequence number or the automaton state in `fs` has been modified since the beginning of this procedure, the current monitor needs to announce its latest flow state.

5) Inter-Monitor Communications: Monitors communicate with each other through messages containing sync states. Sending sync states as soon as they are generated (i.e., in Algorithm 1 Line 10 and in Algorithm 3 Line 15) may introduce unnecessary inter-monitor communications. For example, suppose a sequence of consecutive data packets p_1, p_2, p_3 are received by a monitor m in order, and m can scan them immediately (i.e., the current sequence number in the flow state is equal to p_1's), then m will generate three sync states s_1, s_2, s_3 in order. If all these states are sent, then the recipient monitors will perform flow state updates three times but the first two are unnecessary. In order to avoid such situation and to reduce communications, we use an *outgoing sync state buffer* with size of one, and propose the following mechanism: (1) the BROADCAST and FLUSHSTATEBUFFER procedures are defined in Algorithm 4; (2) FLUSHSTATEBUFFER(`mid`) is called at line 7 in both Algorithm 1 and Algorithm 2, where `mid` is from the received packet or sync state; (3) every time after PROCESSPACKET or PROCESSSTATE finishes, if there is no more received data packet or sync state, then FLUSHSTATEBUFFER(`mid`) is called, where `mid` is a fresh id that is not associated with any existing flow state (i.e., this will cause any buffered sync state to be sent out).

Algorithm 4 Buffered Inter-Monitor Communications

```
Require: buff_s : Sync State ▷ A buffered outgoing sync state; NULL if
    none is buffered

1: procedure BROADCAST(s : Sync State)
2:     FLUSHSTATEBUFFER(s.mid)
3:     buff_s ← s

4: procedure FLUSHSTATEBUFFER(mid : a multipath connection id)
5:     if buff_s ≠ NULL and buff_s.mid ≠ mid then
6:         send buff_s to all other monitors
7:         buff_s ← NULL
```

C. Properties of the MPSD Algorithm

We discuss now key properties of our algorithm.

Lemma 1: At the time a data packet is selected to be scanned by a monitor, all the packets before it in the same multipath connection must have been scanned (possibly by different monitors) in order.

Proof 1: We first prove the first part of the lemma stating that at the time a data packet is to be scanned, its previous packet in the connection must have been scanned. Suppose a data packet p is selected to be scanned (i.e., Algorithm 3 Line 5), its sequence number ($p.mid$) must be equal to that ($fs.mid$) recorded in the flow state fs. There are three places where $fs.mid$ can change its value to be $p.mid$: (1) in Algorithm 1 line 9, (2) in Algorithm 2 line 12, and (3) in Algorithm 3 line 8. In case (1), p must be the first data packet in the connection and hence the first part of the lemma holds trivially. In case (3), the data packet before p must be scanned at line 5, and hence the first part of the lemma also holds. In case (2), a sync state s where $s.mid = p.mid$ and $s.latest\_seqno = p.seqno$ must be received. There are only two places where s can be generated: in Algorithm 1 line 10, and in Algorithm 3 line 14. In the former case, the first part of the lemma holds as in case (1). In the latter case, by the "if" statement's condition and the fact that p is a data packet (i.e., $p.mid \neq fs.fin\_seqno$), line 8 must have been executed and hence the first part of the lemma holds as in

case (3). Therefore, the first part of the lemma is proved. The second part of the lemma (i.e. "all the packets before it must have been scanned in order") follows for any p by applying the first part of the lemma inductively for all the data packets before p.

Lemma 2: Assuming that every packet in a multipath connection is captured by at least one monitor, every data packet is eventually scanned by at least one monitor.

Proof 2: This is proved by induction. (Base case) suppose p is the first data packet received by a monitor m. By Algorithm 1 lines 8–11 and Algorithm 2, m must have a flow state fs such that $fs.cur\_seqno = p.seqno$. By Algorithm 3 p must be scanned. (Induction step) suppose p is a data packet that has been scanned by some monitor m, and the next data packet p' in the connection is received by some monitor m', then by Algorithm 3 a sync state s such that $s.latest\_seqno = p'.mid$ must be generated by m and received by m'. By Algorithm 2, m' must have a flow state fs' such that $fs'.cur\_seqno = p'.seqno$. By Algorithm 3 p' must be scanned by m'. Hence, the lemma can be proved using the base case and induction step for any multipath connection.

Theorem 1: Assuming that every packet in a multipath connection is captured by at least one monitor, if there exists a malicious pattern in the connection, MPSD will detect it.

Proof 3: Using Lemma 1 and Lemma 2, we can show that the distributed scanning of a multipath connection by any number of monitors resembles a centralised scanning of the connection using the Aho-Corasick algorithm. The theorem therefore holds.

Proposition 1: Given a multipath connection with n packets (excluding duplicates) and intercepted by m monitors, suppose the sender switches between the paths k ($k \leq n$) times to send the packets, then in the best case there are $k \times (m-1)$ sent sync states and in the worst case there are $n \times (m-1)$ sent sync states.

Proof 4: Every time the sender switches between the paths to send the packets, there are two consecutive packets p_1 and p_2 received by two different monitors (say M_1 and M_2). By Algorithm 3, M_1 must generate and broadcast (i.e., send to $m-1$ other monitors) a sync state after p_1. Therefore, k is the least number of sync states generated for the connection. In the worst case, if the sender sends the packets down different paths in a round-robin fashion, then $k = n$ and hence there are at least n sync states generated (and broadcasted). Also by Algorithm 3, there is at most one sync state generated for each packet. Therefore, there are at most n generated sync states.

Remark 1: In practice, if $k < n$, the number of sync states generated is between k and n, depending on the network speed. If the network speed is fast enough such that every time a packet is scanned, the next packet is already buffered at some monitor, then there will be only k sync states required. But if the network speed is so slow that the monitors always have to wait for the next packet, then there will be n sync states generated. This can be observed in the experiments described in Section V-C.

Remark 2: Duplicate data packets, either sent by the TCP protocol (e.g., due to packet loss) or created intentionally by the attacker, do not increase the communication overhead significantly. The maximum number of sync states an attacker could theoretically cause the algorithm to generate and broadcast is $m \times n$. This is achieved assuming that (i) the attacker duplicates each data packet and sends them down all paths, and (ii) all monitors receive and scan the packet before receiving the corresponding sync state from at least one other monitor. In practice this worst case is virtually impossible to occur due to asynchrony of the communication channels. Further duplicates of the same packet are ignored by the monitors.

V. Evaluation

A. Experimental Setting

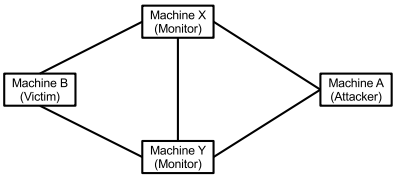

Fig. 4. Experiment Network Setup

We have implemented the multipath signature detection algorithm in C. As packets traverse a monitor, we make a copy of each packet, record the captured time, and extract key fields of the packets including the MPTCP connection token, the MPTCP sequence number, and packet payload. The entire distributed signature scanning process is therefore passive, and does not affect the data transfer between clients and servers.

To evaluate the performance of our proposed solution, we set up a local network as depicted in Figure 4, consisting of four machines, directly connected through gigabit cables. Each machine has an Intel i7-2600 (dual core @ 3.40 Ghz), and is running Ubuntu 12.04 (64-bit). The victim machine, B, is in a network protected by two monitors, machines X and Y, deployed at the ingress points. The attacker machine, A, sends data to the victim B using MPTCP, with two sub-flows (paths A-X-B, and A-Y-B) each going through a different ingress point.

Signatures and data files are randomly generated ASCII texts. A malicious data file is one that has at least one substring matching a signature and that substring is called a (malicious) pattern. Patterns are always artificially injected at a splitting position, so that the pattern spans over two packets. A splitting position in a data file can be calculated based on the fact that most MPTCP data packets (except the last one in the connection) have payloads of size 1428 bytes.

B. Performance metrics

To measure the performance of our proposed solution, we record the following information for each MPTCP connection: the number of packets received by each monitor, the number of sync messages received by each monitor, the total time for each monitor to process a sub-flow, the total download time at the client (B), all the detected patterns with their

detection time. All the events are time stamped using the local system clocks. From this information, we compute the following metrics:

- *Pattern detection delay*: It measures how long it takes for a pattern to be detected by one monitor after the pattern arrives at the victim. In order to avoid errors introduced by network clock synchronization, the delay is calculated as the difference between the time of detection of the pattern and the time of capture of the packet containing the second half of the split pattern, at the same monitor. This computation does not take into account the network delay between the monitors, and client. It therefore represents an upper bound of the time difference between the time an alarm may be raised at a monitor, and the time the victim gets compromised. The actual delay is likely to be smaller as it may take additional time (network delay) for the malicious packet to arrive at the client.
- *Communication ratio*: It measures how much traffic between the monitors is incurred during a MPTCP connection, and is calculated as the number of states received by all the monitors divided by the number of packets passing through the monitors coming from A to B.
- *Download Speed*: we calculate it as the total amount of data sent by the attacker divided by the total download time by the victim for a MPTCP connection.

We have conducted four sets of experiments, each of which is designed to test whether or how certain parameters (e.g., data file size, pattern location, concurrent connections, etc.) may affect the algorithm performance in terms of detection delay. To evaluate the algorithm performance with different network speeds, we also apply rate limiting to control the link capacity from 54Mbps to 450Mbps. Therefore, each experiment has been tested on three configurations for the paths going through X and Y: (1)54Mbps/54Mbps, (2) 54Mbps/450Mbps modeling asymmetric communication channels as is common in mobile devices, and (3) 450Mbps/450Mbps. We will describe each experiment in details and present their results in the next section.

C. Experimental Results

1) Experiment 1 (Pattern Position vs. Performance): The goal of this experiment is to check whether the position of an artificially injected pattern can affect the algorithm performance. Given a randomly generated data file of 2MB, and a randomly generated signature of 20B, there are 1468 (i.e., 2MB / 1428B) splitting positions. 50 data files were obtained by inserting the pattern at the 29th, 58th, ..., 1450th splitting positions. Each data file was sent from A to B 20 times, resulting in 1000 runs (and 1000 MPTCP connections) in total.

The detection delays and the communication ratios under the 54Mbps/54Mbps setting are given in Figure 5, where run IDs $1-19, 20-39, \ldots$ at the x-axis are for the $1^{st}, 2^{nd}, \ldots$ insertion positions. Figure 5 shows that both detection delays and communication ratios are fairly constant across all

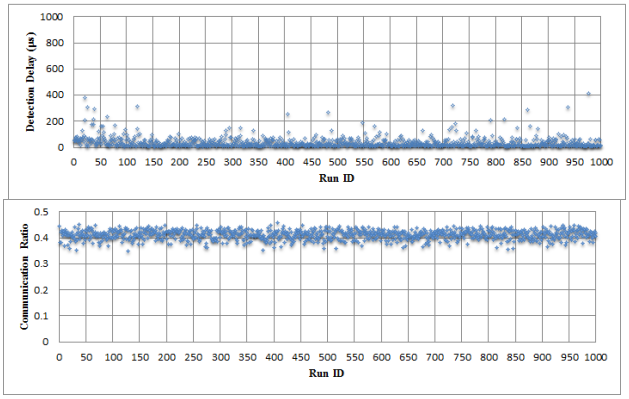

Fig. 5. Exp.1, 54Mbps/54Mbps

runs, with an average delay of 36.51 microseconds and an average communication ratio of 41.17%. Experiments for the 54Mbps/450Mbps and 450Mbps/450Mbps settings (whose plots are omitted due to space limitations) also show constant detection delays and communication ratios.

We include the detection delays and the communication ratios across all three settings in the box-and-whisker plots of Figure 6. In each plot, the dot in the middle of a box corresponds to the median of delay (or communication ratio) values and the edges the first quartile (q_1) and the third quartile q_3. The whiskers extend to the most extreme values not considered outliers, and outliers are plotted individually as red crosses. A value is considered an outlier if it is larger than $q_3 + 1.5 \times (q_3 - q_1)$ or smaller than $q_1 - 1.5 \times (q_3 - q_1)$. For our results, the top whisker is the most relevant since there are a few outliers above it, but they are not significant since the points below the whisker correspond to more than the 95^{th} percentile of all the data (Figure 13 in appendix gives the cumulative distribution function (CDF) plot). The dashed line at the top collapses outliers that are too big to appear in the plot. The box plots show that the bigger the total capacity of all paths, the faster the download speed. And as the download speed increases, the detection delay also increases whereas the communication ratio decreases. The decreasing behaviour of communication ratio conforms to the remark for Proposition 1. To explain the increasing behaviour of detection delay, we conjecture that as the download speed increases, the packet arrival rates at the monitors become larger. As the individual packet scanning speed by any monitor remains constant, there is a larger chance for the malicious packet to stay in the buffer for longer time, and hence increases detection delay. However, the increased rate of detection delay is much smaller than that of download speed. According to the trend of the medians ($y = 0.2385x + 20.7149$), the median detection delay at 1000 Mbps download speed is estimated to be 259 microseconds.

Finally, from the results of Experiment 1 we conclude that detection delay and communication ratio are not affected by the position of the pattern in the multipath connection.

2) Experiment 2 (Number of Patterns vs. Performance): The goal of this experiment is to check whether the total

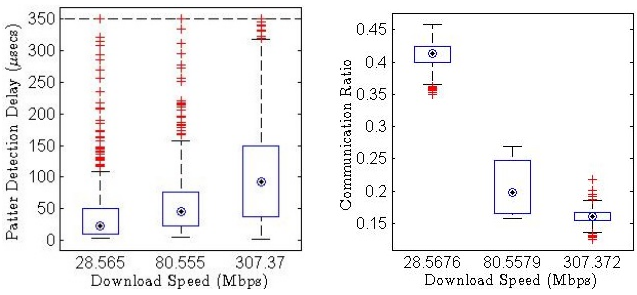

Fig. 6. Exp.1, Box Plots Across Different Settings

number of injected patterns in a data file can affect the algorithm performance. We randomly generated a data file of 2MB, and inserted 1, 2, ..., 50 randomly generated patterns (20B each) at random splitting positions. We repeated this 20 times, resulting in 1000 runs (and 1000 MPTCP connections) in total.

The results for the 54Mbps/54Mbps setting and the box plots across different settings are given in Figure 10 and Figure 11 in appendix, as they are almost identical to those for Experiment 1, not only in terms of the constant behaviour in the detection delays and the communication ratios, but also the distributions of the values (see Figure 13 and Figure 14 in appendix for CDF plots). We can conclude that detection delay and the communication ratio are not affected by the number of patterns in the multiple connection.

3) Experiment 3 (Data Stream Size vs. Performance): The goal of this experiment is to check how the size of data file affects the algorithm performance. We randomly generated data files of size 128KB, 256KB, 512KB, ..., 64MB. For each data file, we inserted a randomly generated pattern of 20B at a random splitting position, and sent it from A to B. We performed this 50 times, i.e., 500 runs (and 500 MPTCP flows) in total. The results under the 54Mbps/54Mbps setting are given in Figures 7

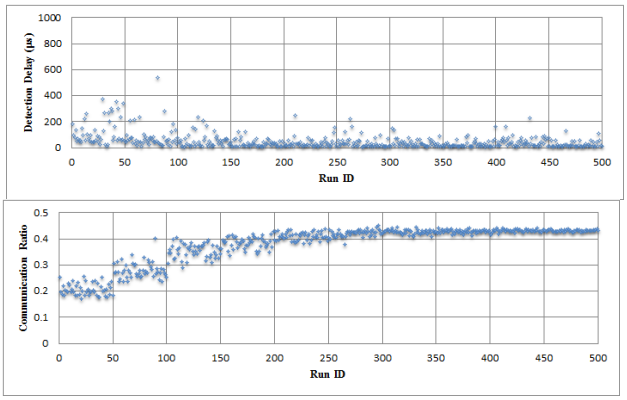

Fig. 7. Exp.3, 54Mbps/54Mbps

The result for detection delay is very similar to the previous two experiments. However, the result for communication ratio differs: it first increases from 20% to just above 40%, and then stays unchanged. Looking at the download speeds of the individual runs, we discovered that the download throughput increases with the file sizes until reaching a maximum value for files of size 2 MB. In addition, the communication ratio for runs 200-249 (i.e., file size of 2MB) is around 41%, which is the same as in the previous two experiments. The box plots across different settings are given in Figure 12 in appendix, as they are very close to those in the previous two experiments.

The results indicate that the data file size does not affect detection delay, but it may affect the download speed which in turn affects the communication ratio.

4) Experiment 4 (Number of Concurrent Connections vs. Performance): The goal of this experiment was to check whether and how concurrent MPTCP flows can affect the algorithm performance. For $N = 2, 4, \ldots 64$, we created N data files of size 1MB, each of which contained a pattern (of size 20B) inserted at a random splitting position. We then sent N files from A to B independently and simultaneously. For each N we did it 20 times, i.e., there were 300 runs and 6300 MPTCP flows in total. The results under the 54Mbps/54Mbps setting are given in Figures 8.

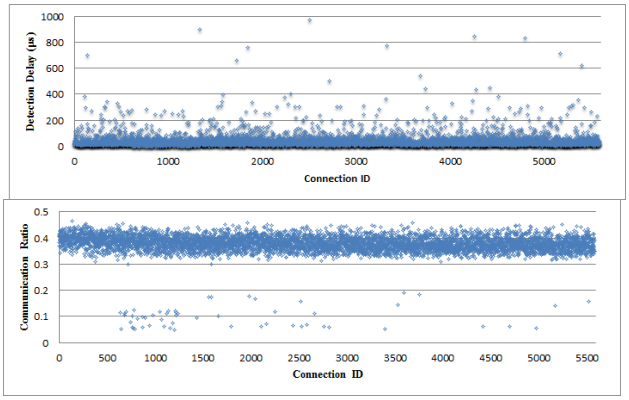

Fig. 8. Exp.4, 54Mbps/54Mbps

The results for this experiment are very similar to the first two experiments, except that a few connections (about 1%) have unexpectedly small communication ratio. We conjecture that this is due to the MPTCP scheduler, which made fewer path switching while sending the packets for those connections.

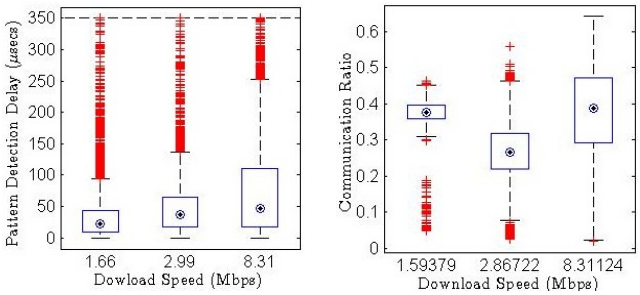

Fig. 9. Exp.4, Box Plots Across Different Settings

The box plots across different settings are given in Figure 9. Differently from all previous experiments, the download

speeds are much smaller. This is because the download speed was calculated for each connection, and there were multiple connections sharing the paths at the same time. The plot for communication ratio across three settings also looks differently from all previous experiments': it does not decrease as the total capacity of all paths increases. We believe this is because all the individual download speeds are relatively slow and are closed to each other, and hence the communication ratio fluctuates around 35%.

Based on the results, we conclude that the number of concurrent multipath connections does not affect the detection delay.

a) Memory Requirement: The execution of the MPSD algorithm requires two main types of memory space: (bootstrap) space for storing the automaton, i.e., the signatures, and (runtime) space for storing data structures during the scanning. Approaches (e.g., [11] have been proposed to reduce the automaton space. The runtime space required is the size needed to store the maximum number of flow states simultaneously maintained by a monitor, which is almost the same as the total number of packets buffered in all the flow states. During our experiments, we observed that the maximum packet buffer size at any monitor at any time was 10 under the 54Mbps/54Mbps setting and 27 under the 450Mbps/450Mbps setting.

b) Inter-Monitor Communication Links: The communication ratio r measures on average how much traffic between the monitors is required based on the traffic on the network, and is calculated as the total number of sync states divided by the total number of packets. Let P_d be the traffic (in Mb) on the whole network per second, let S_d and S_m be the maximum size of a data packet and the maximum size of a packet containing a sync state, respectively, then the throughput of each inter-monitor link P_m can be calculated as $P_m = P_d \times r \times (\frac{S_m}{S_d})$. In our experiment, S_d is 1500B and S_m is 52B. Consider the worst case where $r = 100\%$, and suppose P_d is 1000Mbps, then $P_m = 1000 \times 1 \times (52/1500) = 34.67$ Mbps is the minimum throughput that each inter-monitor link needs to guarantee.

c) Adversary Attacks Using Ambiguity: One of the assumptions of MPSD (and most S-IDS) is that a data packet cannot be modified. However, [12] shows an attack that exploits this assumption. For instance, the attacker first sends a non-malicious packet with small enough TTL to cause it to be dropped between the S-IDS and end-host. Then the attacker sends the packet again with injected malicious data and big enough TTL to reach the end-host. As such, the packet is considered by the S-IDS as duplicate and not scanned. Such attacks can be addressed by MPSD similarly as in the single-path case, for instance by making the monitors topology-aware ([12]). As future work we will investigate whether it is possible to create more complex attacks in the case of multi-paths and develop counter-measures.

VI. Related Work

A number of approaches for network intrusion detection have been proposed in the literature (for comprehensive surveys see [13], [14] and[15]). These can be largely grouped into systems for anomaly detection and systems for signature-based detection. The former (e.g. [16]) identify a priori (e.g. through statistical models) the "normal" traffic of a given network, and detect intrusions by flagging monitored traffic that deviates from the normality. Signature-based intrusion detection systems use signatures of known attacks, expressed as patterns or strings, and detect intrusions by matching them against the network traffic (e.g. packet's payloads). Whereas anomalies detection systems are often considered to be more challenging, due to the difficulty of having to recognize unknown intrusions without causing many false positive alarms, signature-based approaches are more widely used (e.g. [7]). This is due to their high-level of accuracy and ability to support contextual analysis, which in turns enables easier preventive or corrective action [17]. We focus our related work discussion on signature-based network intrusion detection (S-IDS) approaches as our algorithm adopts a similar intrusion detection methodology.

S-IDS approaches provide a simple and effective method for detecting attacks, specified as exact matching strings (i.e. string of ASCII symbols). They use (combination of) high speed and efficient string matching algorithms. For instance, [18] provide a multi-pattern matching algorithm that combines Boyer-Moore [10] regular string-searching technique with Aho-Corasick's algorithm [8], whereas [19] allows for multiple strings to be searched at the same time in order to support a large number of patterns. The Aho-Corasick algorithm is one of the earliest and efficient algorithms in exact multi-pattern string matching. It is today deployed by one of the most popular open source tool for intrusion detection [7]. The algorithm uses a finite automata, built from the string signatures set, that can be either non-deterministic, by including from each node a failures point transition that takes the node to the longest prefix that would lead to a valid string, or deterministic. Proposed enhancements of the Aho-Corasick's algorithm have focused on improving the bootstrap memory required to store the string signature set. For instance, [11] has shown that by using bitmap and path compression it is possible to reduce the memory required for the signature set by up to a factor of 50%. High speed efficient algorithms have also been proposed to convert a deterministic automaton into multiple binary state machines, each with much fewer state transitions, showing that this can dramatically reduce the total space required [20]. Our multi-path signature detection algorithm builds upon the basic Aho-Corasick's algorithm, with deterministic automaton stored by means of basic data structures. This is because our main concern in this paper is the analysis of the execution time of our distributed solution instead of the static memory required for storing the automaton. Existing enhancements of the Aho-Corasick's algorithm could be integrated in our algorithm in order to achieve also improved memory usage. For instance, the use of a nondeterministic automaton would simply require a non-deterministic NextState function in line 7 of algorithm 3.

State-modelling S-IDS are another type of signature-based intrusion detection. They encode intrusions as chain of differ-

ent states that have to be recognised in a given time series order [21]. S-IDS based on expert systems (e.g. [22], [23]) use rules to describe intrusive behaviours. Forward-chaining rule-based tools are used, which combine monitored events entering the systems as facts together with the rules in order to decide whether an intrusion has occurred. Limitation of these approaches is the execution speed due to complexity and generality of the rule-based engine. String matching S-IDS perform simple substring matching of characters in text. They are not very flexible, but on the other hand are simple to understand. Simple rule-based intrusion detection systems (e.g. [24], [23]) are similar to the more powerful expert system approaches and they often lead to speedier execution. Our S-IDS falls in the category of (exact multiple) string matching, but it can be integrated within a rule-based intrusion deduction system such as the Snort system [7].

Somewhat related to our work are proposals of signature-based intrusion detection in wireless ad-hoc networks [25]. Netwook IDS for wireline networks are not appropriate for wireless ad-hoc networks due to the mobility of the network. Fixed check points (monitors) where network traffic can be analysed are not present. Intrusion detection mechanisms are therefore enforced in some or all the nodes in the network, and whenever malicious packets are detected, while in transit between the source (intruder) node and the destination node, the routing protocol is assumed to route the packets via paths that pass through the allocated monitors. In this case, the effectiveness of intrusion detection is therefore strictly dependent on the effectiveness of the routing protocol in supporting this task. However [25] shows that even though some routing protocols for ad-hoc network behave better than others, there is still the fundamental problem that malicious packets may take different routes, and the signature-based attack detection will in this case have an incomplete information of the traffic. This is indeed the problem that our algorithm addresses. Appropriate integration of our distributed string matching approach in routing protocols for wireless ad-hoc network could provide a valuable solution also to the problem of intrusion detection in such class of networks.

Furthermore, researchers have demonstrated how attackers could exploit ambiguities in the traffic stream to evade detection [9]. Solutions such as traffic normalizer [26] have been proposed to address and remove potential ambiguities, and most network intrusion detection systems are now robust to these attacks, by handling IP fragmentation, and supporting TCP flow reconstruction.

More recently, threats analyses have been performed with respect to the MPTCP and potential threats on the protocol itself have been identified [5], [6]. However, these analyses do not describe how attackers could exploit MPTCP to evade network intrusion detection systems, and how to prevent them.

VII. Conclusion and future work

We presented a new network attack that exploits MPTCP and evades existing signature-based intrusion detection mechanisms. To address the problem, we also proposed a distributed signature-based intrusion detection algorithm that defines the S-IDS problem in terms of a distributed exact string matching problem where monitors, located on different paths, share a global state of the string matching automaton for each MPTCP connection. Different sub-flows, with split signatures, may be received by different monitors. The monitors scan each received packet locally and broadcasts its automaton state to all the other monitors. The broadcast enables the monitors to synchronise their local scans. Through comprehensive experimental results we have show that the performance of the algorithm depends only on the network throughput. Delays in detecting the signature grows linearly with respect to the throughput, whereas the communication ratio decreases with the increase of the throughput.

In future work, we will investigate optimizations to further reduce the detection delay, an aspect that is key for active S-IDS. More specifically, in the current implementation, when receiving out-of-order packets, each monitor waits for some sync state before scanning the received packets. Instead, monitors could process those received out-of-order packets and only store the first m bytes of the payload, where m is the maximum size of the signatures. This would allow signatures to be detected more quickly. We will implement this evaluation, and evaluate its performance.

References

[1] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. (2013, Jan.) Tcp extensions for multipath operation with multiple addresses. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6824.txt

[2] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, implementation and evaluation of congestion control for multipath tcp," in *Proceedings of NSDI*, 2011.

[3] Y.-C. Chen, Y.-s. Lim, R. J. Gibbens, E. M. Nahum, R. Khalili, and D. Towsley, "A measurement-based study of multipath tcp performance over wireless networks," in *Proceedings of IMC*, 2013.

[4] L. Deng, D. Liu, and T. Sun. (2014) Mptcp proxy for mobile networks. [Online]. Available: http://www.ietf.org/id/draft-deng-mptcp-mobile-network-proxy-00.txt

[5] M. Bagnulo, "Threat analysis for tcp extensions for multipath operation with multiple addresses," 2011, RFC 6181.

[6] M. Bagnulo, C. Paasch, F. Gont, O. Bonaventure, and C. Raiciu, "Analysis of mptcp residual threats and possible fixes," January 2014, internet-Draft, Internet Engineering.

[7] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *LISA*, vol. 99, 1999, pp. 229–238.

[8] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[9] T. H. Ptacek and T. N. Newsham, "Insertion, evasion, and denial of service: Eluding network intrusion detection," DTIC Document, Tech. Rep., 1998.

[10] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[11] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *INFOCOM*, 2004.

[12] U. Shankar and V. Paxson, "Active mapping: Resisting nids evasion without altering traffic," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, ser. SP '03, 2003, pp. 44–.

[13] H.-J. Liao, C.-H. Richard Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, 2013.

[14] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," Technical report, Tech. Rep., 2000.

[15] R. Srivastava and V. Richhariya, "Survey of current network intrusion detection techniques," *Journal of Information Engineering and Applications*, vol. 3, no. 6, pp. 27–33, 2013.

[16] K. Wang and S. J. Stolfo, "Anomalous payload-based network intrusion detection," in *Recent Advances in Intrusion Detection*. Springer, 2004, pp. 203–222.

[17] H. Debar, M. Dacier, and A. Wespi, "Towards a taxonomy of intrusion-detection systems," *Computer Networks*, 1999.

[18] B. Commentz-Walter, *A string matching algorithm fast on the average*. Springer, 1979.

[19] S. Wu, U. Manber *et al.*, "A fast algorithm for multi-pattern searching," Technical Report TR-94-17, University of Arizona, Tech. Rep., 1994.

[20] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *ACM SIGARCH Computer Architecture News*, vol. 33. IEEE Computer Society, 2005, pp. 112–122.

[21] K. Ilgun, "Ustat: A real-time intrusion detection system for unix," in *Research in Security and Privacy, 1993. Proceedings., 1993 IEEE Computer Society Symposium on*. IEEE, 1993, pp. 16–28.

[22] T. F. Lunt and R. Jagannathan, "A prototype real-time intrusion-detection expert system." in *IEEE Symposium on Security and Privacy*. Oakland, CA, USA, 1988, pp. 59–66.

[23] N. Habra, B. Le Charlier, A. Mounji, and I. Mathieu, "Asax: Software architecture and rule-based language for universal audit trail analysis," in *Computer SecurityESORICS 92*. Springer, 1992, pp. 435–450.

[24] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23, pp. 2435–2463, 1999.

[25] F. Anjum, D. Subhadrabandhu, and S. Sarkar, "Signature based intrusion detection for wireless ad-hoc networks: A comparative study of various routing protocols," in *Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003 IEEE 58th*, vol. 3. IEEE, 2003, pp. 2152–2156.

[26] M. Handley, V. Paxson, and C. Kreibich, "Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics." in *USENIX Security Symposium*, 2001, pp. 115–131.

APPENDIX

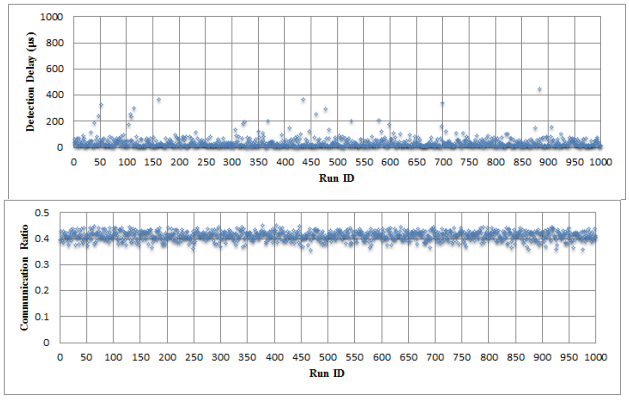

Fig. 10. Exp.2, 54Mbps/54Mbps

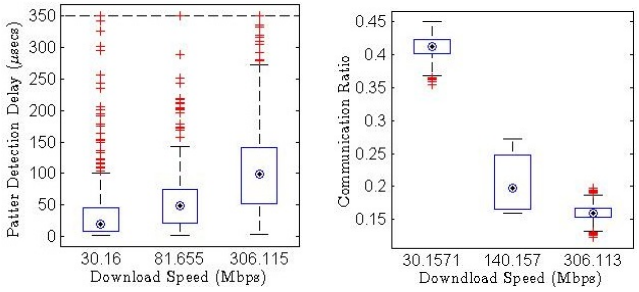

Fig. 11. Exp.2, Box Plots Across Different Settings

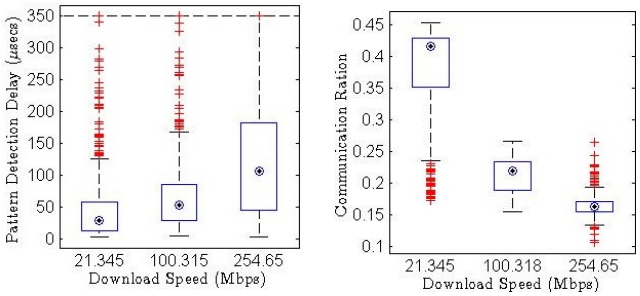

Fig. 12. Exp3, Box Plots Across Different Settings

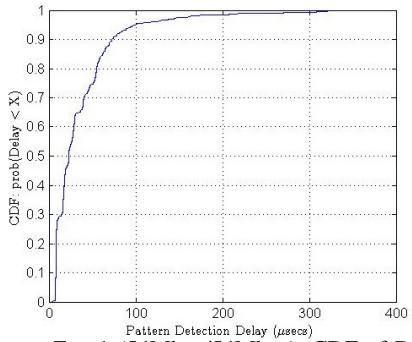

Fig. 13. Exp.1 (54Mbps/54Mbps), CDF of Delay

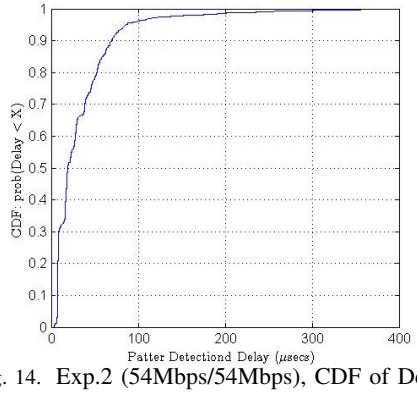

Fig. 14. Exp.2 (54Mbps/54Mbps), CDF of Delay

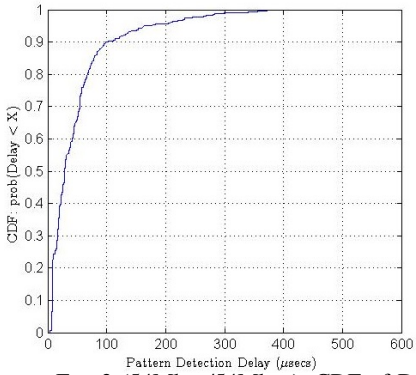

Fig. 15. Exp.3 (54Mbps/54Mbps), CDF of Delay

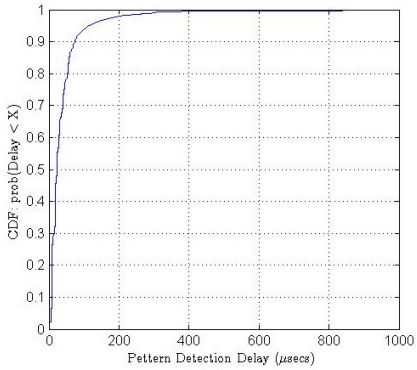

Fig. 16. Exp.4 (54Mbps/54Mbps), CDF of Delay