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

Jiefei Ma*, Franck Le†, Alessandra Russo* Jorge Lobo‡
*Imperial College London, †IBM T. J. Watson, ‡ICREA - Universitat Pompeu Fabra

***Abstract*—Signature-based network intrusion detection systems (S-IDS) have become an important security tool in the protection of an organisation's infrastructure against external intruders. By analysing network traffic, S-IDS' detect network intrusions. An organisation may deploy one or multiple S-IDS', each working 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 different paths (e.g., WiFi, Cellular) to boost network performance. 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 its effectiveness we conducted comprehensive experiments. Our 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 communication overhead.**

I. INTRODUCTION

Multi-path TCP (MPTCP) is a new set of IETF standardised 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 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 analysed [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 organisation's infrastructure against external intruders. An organisation may deploy one or multiple S-IDS', each working independently with the assumption that it can monitor all packets of a given flow to detect intrusion signatures. However, attackers may exploit the multiplicity of paths and split malicious payloads across multiple paths to evade traditional S-IDS'. 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 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 the size and number of MPTCP connections, and the 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 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 implementation is 52 bytes including the message headers. In practice, the communication overhead varied from 1.56% to 0.52%, decreasing as the throughput increases.

II. BACKGROUND

A. Network Intrusion Detection System

S-IDS' analyze network traffic to compare packets against a database of signatures from known malicious threats. S-IDS' are commonly classified into active versus passive. Active S-IDS' can drop packets and halt an attack in progress. In contrast, passive S-IDS' raise alarms, and rely on humans to take subsequent actions. In this paper, we focus on the passive approach, considering that many commercial S-IDS' are solely passive [9]. We discuss how extensions for active S-IDS' can be made in Section VII.

S-IDS' 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, S-IDS' 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 [10].

B. *MPTCP*

MPTCP is a new transport protocol that allows two endpoints to simultaneously use multiple paths between them. It is 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 subflows, and for the receiver to successfully re-order the received bytes over the different paths.

III. NEW THREAT

This section describes how through MPTCP, attackers could evade traditional S-IDS'. 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 S-IDS at each ingress point to monitor and analyses 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 S-IDS.

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 S-IDS 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.

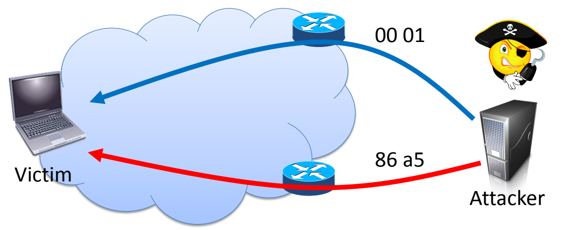

Fig. 1. New threat: With multi-path routing, attackers could evade traditional S-IDS' by splitting the signature (e.g., "00 01 86 a5") over different paths.

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 S-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 $x^1 y^2 a^3 b^4\ c^5 z^6$ is an annotated text, and $\{x^1 y^2 c^5 a^3, y^2 b^4 z^6\}$ is a possible set of two observed texts, in which a^3 is *delayed* after c^5, and y^2 is *duplicated*. Then, an asynchronous online exact string matching problem is defined as a set of m network nodes (*monitors*) $\{\alpha_1, \ldots, \alpha_m\}$ that 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 centralised 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 algorithm [11]), 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 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 detected signatures. For example, given a set of signatures $\{he, she, his, hers\}$, the computed automaton 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 identifies all the matching signatures in a given string.

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

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

$\textbf{state}_{\textbf{accepting}}$	2	5	7	9
$\sigma(\textbf{state}_{\textbf{accepting}})$	$\{he\}$	$\{she, he\}$	$\{his\}$	$\{hers\}$

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.

B. Multi-path Signature Detection Algorithm

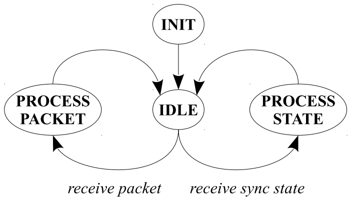

Fig. 3. MPSD Algorithm States

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: received packets (called *data packets*) from the network, and synchronisation 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 synchronisation 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, and is represented as `<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, $-\infty$, 0, $+\infty$, $\emptyset$>
 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 $\geq$ 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 exists yet. 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 replaces $-\infty$ with the expected 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 receiving the `START` packet has to tell others about the first data sequence number (lines 10–11). The sequence number of an `END` packet is stored (line 13) in the flow state for future termination of the sub-flow scans (see Algorithm 3). Lines 14–17 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[1]. Otherwise, the packet must have 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 already has a packet with the same sequence number, then the newly received packet is also a duplicate and therefore discarded. 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, $-\infty$, 0, $+\infty$, $\emptyset$>
 4:         store fs
 5:     else
 6:         fetch fs where fs.mid = s.mid
 7:                                                  ▷ Placeholder
 8:     if s.latest_state = $\infty$ 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 indicates that the sender monitor has scanned the last data packet of the whole multipath connection, then the receiver monitor removes the current flow state to free resources. Otherwise, `s`' sequence number is compared with `fs`'. If `s` has a larger sequence number (line 11), then the sender monitor must have received and scanned some packets of the 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.

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

---

[1] Note that, as in traditional passive S-IDS', packets may get dropped when the buffer is full, potentially impacting the correctness of the algorithm.

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 $\leftarrow$ dequeue fs.packets
6:         for all each character c in p.payload do
7:             fs.cur_state $\leftarrow$ NextState(fs.cur_state, c)
8:             fs.cur_seqno $\leftarrow$ 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 $\leftarrow \infty$
13:    if fs.cur_seqno has changed value or fs.cur_state has become $\infty$ then
14:        s $\leftarrow$ <fs.mid, fs.cur_seqno, fs.cur_state>
15:        Broadcast(s)                              ▷ Communication
16:    if fs.cur_state $= \infty$ then
17:        remove fs
```

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 larger 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`, then the current monitor must have the next data packet in the multipath connection and can resume the scanning process. The head packet is dequeued and scanned, and any detected pattern is stored as alerts. This step is repeated until the buffer becomes empty or the head packet has a larger 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 state `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, then the latest flow state is sent to the other monitors.

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 generates three sync states s_1, s_2, s_3 in order. If all these states are sent, then the recipient monitors performs three flow state updates but the first two are unnecessary. In order to avoid such situation and to reduce communication, we use an *outgoing sync state buffer* with size one, and implement the following enhancements: (1) the Broadcast() and FlushStateBuffer() procedures, defined in Algorithm 4. FlushStateBuffer(`mid`) is called at line 7 in both Algorithm 1 and Algorithm 2, where `mid` is from the received packet or sync state; 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 causes 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

procedure Broadcast(s : Sync State)
    FlushStateBuffer(s.mid)
    buff_s $\leftarrow$ s

procedure FlushStateBuffer(mid : a multipath connection id)
    if buff_s $\neq$ NULL and buff_s.mid $\neq$ mid then
        send buff_s to all other monitors
        buff_s $\leftarrow$ 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 packets before it in the same multipath connection must have been scanned (possibly by different monitors) in order.

Lemma 2: Assuming that every packet in a multipath connection is received by at least one monitor and no buffer overflow occurs, every data packet is eventually scanned by at least one monitor.

The full proofs of Lemma 1 and Lemma 2 are given in an extended version of this paper [12] available online.

Theorem 1: Assuming that every packet in a multipath connection is received by at least one monitor, if there exists a malicious pattern in the connection and no buffer overflow occurs, then MPSD detect the pattern.

Proof 1: 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 2: Every time the sender switches between the paths to send the packets, there are two consecutive packets p_h and p_{h+1} received by two different monitors (say M_i and M_j). By Algorithm 3, M_i must generate and broadcast (i.e., send to $m-1$ other monitors) a sync state after p_h. 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 required sync states. However, 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 asynchronicity of the communication channels. Further duplicates of the same packet are ignored by the monitors.

V. EVALUATION

A. Experimental Setting

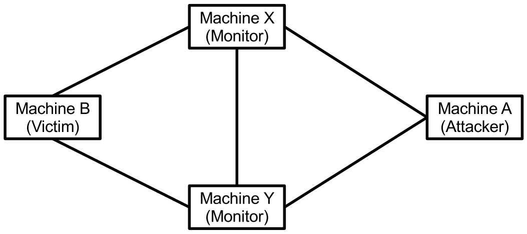

Fig. 4. Experiment Network Setup

We implemented the MPSD 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 runs 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 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 contains at least one substring matching a signature and that substring is called a (malicious) pattern. Patterns are 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) and 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 the time 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 the packet containing the second half of the split pattern is received at the 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 overhead (and ratio)*: It measures the amount of traffic between the monitors during a MPTCP connection, and is calculated as the number of bytes (and states) received by all the monitors divided by the number of bytes (and packets) passing through the monitors coming from the sender to the receiver.
- *Download Speed*: It is the total amount of data sent by the attacker divided by the total download time by the victim for a MPTCP connection.

We conducted four sets of experiments, each 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 rate limited the link speed 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 modelling asymmetric communication channels as is common in mobile devices, and (3) 450Mbps/450Mbps. We 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).

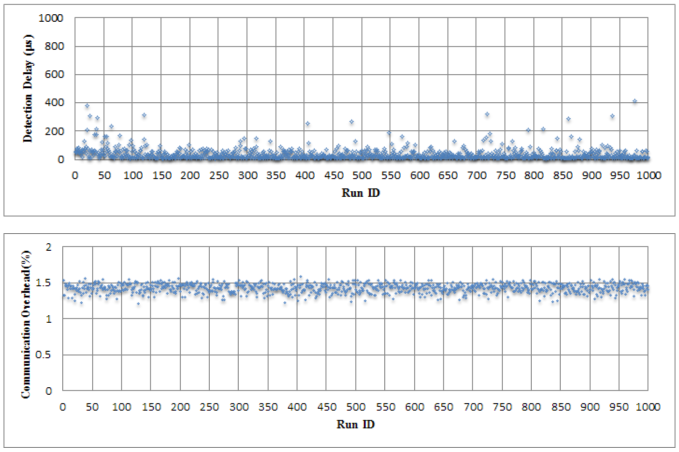

Fig. 5. Exp.1, 54Mbps/54Mbps

The detection delays and the communication overhead 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 overhead are fairly constant across all runs, with an average delay of 36.51 microseconds and an average communication overhead of 1.43%. Experiments for the 54Mbps/450Mbps and 450Mbps/450Mbps settings (whose plots are omitted due to space limitations) also show constant detection delays and communication overheads.

We include the detection delays and the communication overheads across all three settings in the box-and-whisker plots of Figure 6. In each plot, the point in the box represents the median delay (or communication overhead) values and the edges represent 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 the extended paper [12] gives the cumulative distribution function (CDF) plot). The dashed line at the top collapses outliers that are too large to appear in the plot. The box plots show that the larger 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 overhead decreases. The decreasing behaviour of communication overhead 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.

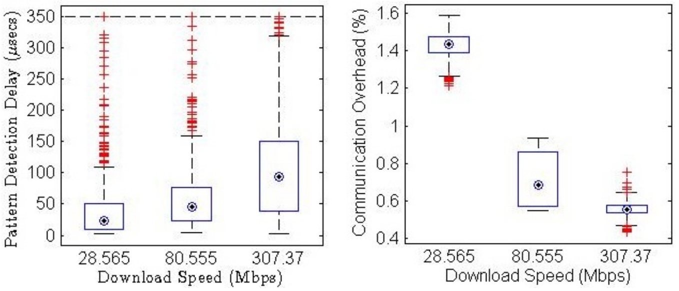

Fig. 6. Exp.1, Box Plots Across Different Settings

Finally, from the results of Experiment 1 we conclude that detection delay and communication overhead 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 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).

The results for the 54Mbps/54Mbps setting and the box plots across different settings are given in Figure 10 and Figure 11 of the extended paper [12], 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 overheads, but also the distributions of the values (see Figure 13 and Figure 14 in the extended paper for CDF plots). We conclude that detection delay and the communication overhead 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). The results under the 54Mbps/54Mbps setting are given in Figures 7

The result for detection delay is very similar to those of the previous two experiments. However, the result for communication overhead differs: it first increases from 0.69% to just above 1.39%, 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 overhead for runs 200-249 (i.e., file size of 2MB) is around 1.43%, which is the same as in the previous

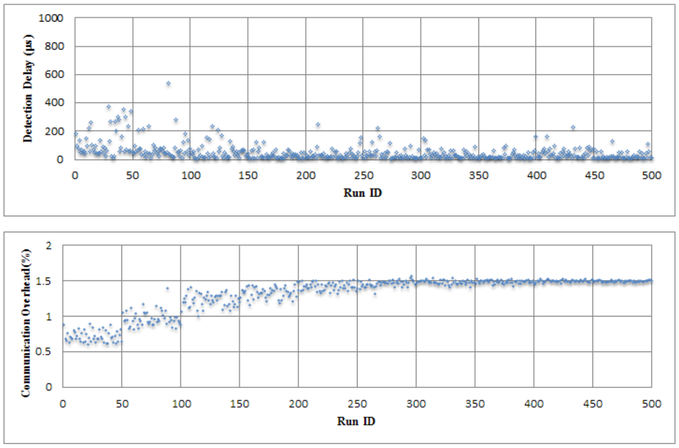

Fig. 7. Exp.3, 54Mbps/54Mbps

two experiments. The box plots across different settings are given in Figure 12 in the extended paper [12], 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 overhead.

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 $N = 2, 4, \dots 64$, 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 Figure 8.

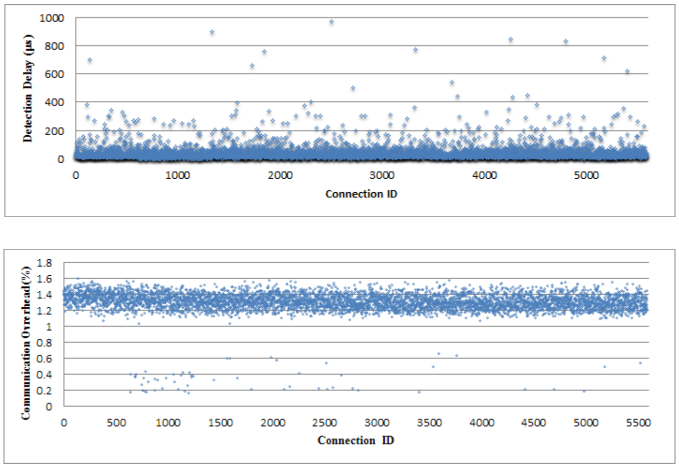

Fig. 8. Exp.4, 54Mbps/54Mbps

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

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 overhead 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 overhead fluctuates around 1.21%.

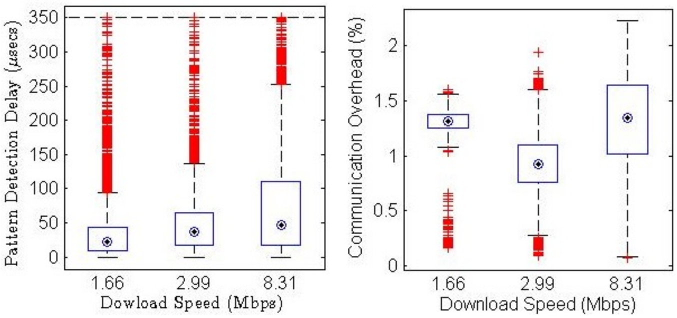

Fig. 9. Exp.4, Box Plots Across Different Settings

Based on these 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., [13]) 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 many messages between the monitors are 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, [14] 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 large 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 ([14]). 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

Because of their importance, a large amount of research has been devoted to network intrusion detection [15], [7], [16], [11], [8], [17], [13], [18], [19], [20], [10], [21]. While several approaches have been developed, exact string matching is among the most widely adopted technique because of its simplicity and accuracy. However, none of the existing work prevents the attack presented in this paper.

More specific to MPTCP, threats analyses have been performed [5], [6]. However, these analyses focus on the protocol (e.g., if one could launch a man-in-the-middle attack), 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 shown 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 overhead 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.

ACKNOWLEDGMENT

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

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] R. Bace and P. Mell, "Nist special publication on intrusion detection systems," DTIC Document, Tech. Rep., 2001.

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

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

[12] "Detecting distributed signature-based intrusion: The case of multi-path routing attacks (extended version)," http://wp.doc.ic.ac.uk/arusso/project/declarative-networking-for-network-and-security-management-of-hybrid-and-dynamic-networks/infocomm15-ext/.

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

[14] 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–.

[15] 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.

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

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

[18] 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.

[19] 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.

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

[21] 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.