Mitigating TCP Protocol Misuse with Programmable Data Planes

Abir Laraba, Jérôme François, *Member, IEEE*, Shihabur Rahman Chowdhury, *Student Member, IEEE*, Isabelle Chrisment, and Raouf Boutaba *Fellow, IEEE*

***Abstract*—This paper proposes a new approach for detecting and mitigating the impact of misbehaving TCP end-hosts, specifically the Optimistic ACK attack, and Explicit Congestion Notification (ECN) abuse. In contrast to the state-of-the-art, we show that it is possible to mitigate such misbehavior leveraging emerging programmable data planes while not requiring any end-host or protocol modifications. A key challenge in doing so is to implement expressive, complex and stateful functions in the data plane within its restricted programming model. In this regard, we propose a security monitoring function that uses Extended Finite State Machine (EFSM) abstraction for monitoring stateful protocols in the data plane. We also design a mechanism for mapping a protocol's EFSM to programmable data plane primitives. Our evaluation results demonstrate that our approach can fully or partially restore the throughput loss caused by misbehaving end-hosts that manipulate TCP congestion control through misinformation.**

***Index Terms*—SDN, P4, Programmable data plane, Security, Monitoring, EFSM, ECN, Optimistic ACK**

I. Introduction

NETWORK protocols can be subject to attacks from non-compliant or misbehaving end-hosts that exploit protocol vulnerabilities. For instance, many amplification Distributed-Denial-of-Service (DDoS) attacks exploit vulnerabilities of protocols such as SNMP, NTP or DNS, among others [1]. At the application layer, protocol verification has been used to reveal potential vulnerabilities such as for the SIP authentication [2] and more recently for 5G networks [3]. Once exposed, protocol vulnerabilities can be patched with some effort. However, patching at scale becomes very challenging when the concerned protocol lies at the core of the Internet and affects primary services. The slow deployment of DNSSEC [4] testifies about the difficulty of patching a critical service at scale.

Transmission Control Protocol (TCP), one of the fundamental building blocks of the Internet, is also prone to attacks [5], [6] unfortunately. TCP vulnerabilities exist because TCP was not designed with strong security considerations. Also, TCP's evolution over time didn't address security issues to maintain backward compatibility. The Optimistic ACK attack is one such example, where a misbehaving TCP receiver acknowledges segments before they are received, in this way manipulating the sender to transmit faster [5]. The impact of the Optimistic ACK attack can be significant in the presence of multiple victims, (*e.g.*, up to 5 GB/s for about 512 victims) [6]. Another example is when a misbehaving TCP receiver does not comply with Explicit Congestion Notification (ECN), and ignores congestion notification to manipulate the sender to keep the same transmission rate during congestion [7]. The research literature discusses several techniques that a misbehaving receiver can use for manipulating TCP congestion control [5], [6], [8]–[10], consequently causing unfair division of shared bandwidth among competing flows.

One mitigation approach is to modify the TCP protocol specification and end-host implementation as new attacks emerge [5], [6], while such change is hard for practical deployment. Moreover, attacks such as ECN abuse are difficult to detect at end-hosts. Typically, network operators deploy hardware or software appliances [11], [12] for monitoring anomalies in layer-4 and above stateful protocols. However, this approach adds cost and operational overhead. In this paper, we take a different approach for detecting and mitigating TCP protocol misuse. We leverage data plane programmability enabled by protocol independent switch architecture (PISA) [13] and P4 programming language [14] for defending against misbehaving TCP end-hosts directly in the switches. Our approach does not require changing the end-hosts and protocol specification. Moreover, as discussed in the literature [15], our approach leveraging data plane programmability has the potential to reduce the capital and operational cost compared to the traditional approaches relying on network security appliances.

We propose to use Extended Finite-State Machine (EFSM) for modeling a TCP protocol behavior to monitor. Then, we design a method to map this EFSM to a P4 program adhering to its restricted computing model. Note that the EFSM abstraction can be adapted to other layer-4 and above protocols, which we leave for future investigation. The rationale for using EFSMs is because they avoid state space explosion by defining variables and actions in addition to state transitions in a finite-state machine. To the best of our knowledge, this is the first endeavor for online detection and mitigation of ECN and Optimistic ACK attacks within the network, without requiring the use of network appliances or requiring end-host modifications. Specifically, we make the following contributions:

- A procedure for mapping an EFSM modeling the behavior of a protocol and its possible misuse detection to a P4 program running on PISA targets [16].

A. Laraba (email: abir.laraba@loria.fr), J. François (email: jerome.francois@loria.fr), and I. Chrisment (email: isabelle.chrisment@loria.fr) are with Université de Lorraine – INRIA, Nancy, 54600, France.

S. R. Chowdhury (email: sr2chowdhury@uwaterloo.ca) and R. Boutaba (email: rboutaba@uwaterloo.ca) are with the David R. Cheriton School of Computer Science, University of Waterloo, ON N2L 3G1, Canada.

- An application of the proposed method for detecting and mitigating two representative TCP misuses, namely the Optimistic ACK attack and ECN abuse.
- Experimental evaluation demonstrating the effectiveness of the proposed method. The results show that our solution successfully mitigates unfair bandwidth sharing caused by both attacks from within the data plane.

This paper extends our initial work in [17].We provide a formal description of how an EFSM abstraction can be mapped to P4 primitives. Previously, we used an EFSM-based technique for detecting and mitigating ECN abuse. In this paper, we extend our application to the Optimistic ACK attack on TCP congestion control. Specifically, we describe the attack in Section IV-A followed by the detailed mechanism of how this can be modeled using an EFSM in Section IV-B. This section also describes how our method of mapping an EFSM model to a P4 program can be used with a new attack model. We give additional evaluation results demonstrating the effectiveness of our proposed approach in detecting and mitigating Optimistic ACK attacks in Section V. Finally, we present a more elaborate discussion of the state-of-the-art works.

The rest of this paper is organized as follows. We present the background and overview of the proposed approach in Section II. Details of our EFSM abstraction and its mapping to a P4 program are presented in Section III. We describe the application of our proposed method to the Optimistic ACK attack in Section IV followed by experimental results in Section V. Then, we explain the ECN protocol mechanism abuse use-case and its detection and mitigation leveraging EFSM in Section VI. Our experimental evaluation of the ECN abuse use-case is presented in Section VII. Then, we discuss practical deployment considerations, including the memory overhead and stateful TCP connection tracking considerations in Section VIII. Then, we discuss and contrast our solution with state-of-the-art works in Section IX. Finally, we conclude with some future research directions in Section X.

II. Background and Solution Overview

A. Data plane programming using P4

P4 is a language for expressing how packets are parsed, processed and deparsed by the data plane elements (switches). P4 allows each hardware vendor to provide a target-specific compiler, making the P4 programs portable across different targets. P4 follows a two-step compilation process. The first step generates an intermediate and hardware-independent representation which is compiled in the second step to a specific hardware target. This intermediate representation is designed to be generic enough to capture behaviors on a large variety of targets, such as NetFPGA [18].

P4 assumes an abstract forwarding model consisting of a programmable parser and a set of match-action tables, divided between an ingress and an egress control pipeline and a programmable deparser as defined in the P4-16 version of the language specification [19]. The parser extracts the headers from the incoming packets. Each match-action table performs a lookup on a subset of header fields and applies the actions within each table. The deparser's goal is to reassemble packets after they have been processed. In our context, dedicated P4 actions will be defined for transitioning state in the EFSM and so require some states to persist in the data plane across packets. P4 provides three types of stateful objects:

- **Match-Actions tables:** consist of the table entries and the possible actions, table entries are typically modified by the control plane.
- **Registers:** stateful memories that maintain state across packets. P4 actions can read or modify register values. It is worth mentioning that in the latest P4 specification (P4-16), registers are supposed to be exposed by extern functions and are thus not anymore hardware-independent as in the previous specification [20]. However, assuming registers as basic components to be provided by a hardware platform is reasonable.
- **Metadata:** per-packet state which may not be derived from packet data. It allows carrying information across multiple P4 processing stages.

B. Solution Overview

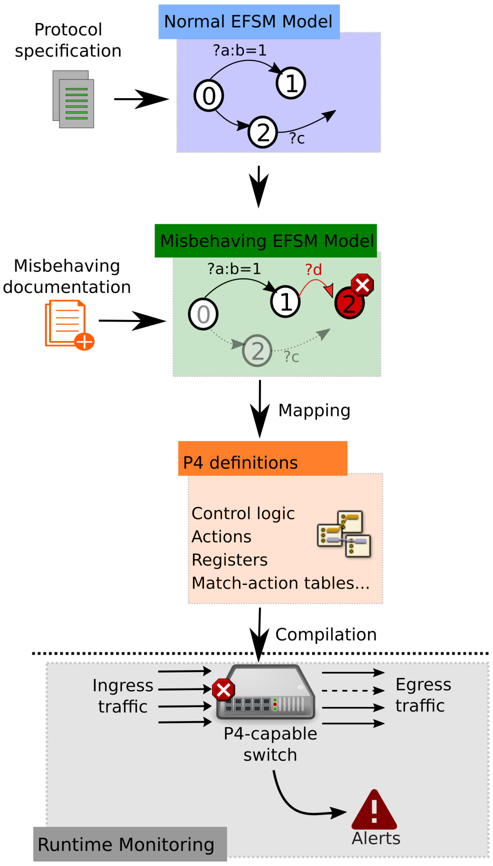


Fig. 1: Approach overview

Our approach for detecting and mitigating protocol misbehavior in the data plane is outlined in Figure 1. The first step is to transform a protocol specification into an Extended Finite State Machine (EFSM). The advantage of using EFSM over regular Finite-State Machine (FSM) is EFSM's ability to store persistent values in variables, thereby, limiting state explosion. For example, maintaining a counter in a FSM would require one state per counter value, whereas the same can be represented in an EFSM with one variable corresponding to the counter. Once the initial protocol specification is modeled as an EFSM, we extend that EFSM with misbehaviors. In this stage, the EFSM can be also compressed, as normal states representing normal behaviors can be merged. We only consider intermediate normal states that are necessary to track before a misbehavior can occur. Once the EFSM of the protocol is defined and extended with misbehaving states, it is mapped to a P4 program using the language primitives. This stage of mapping the EFSM model to a P4 program is essential and is described in Section III-B.

Finally, when the compiled program is installed on a PISA target switch, all flows are tracked online by maintaining the current state of each connection. Network operators can configure the PISA switches to take different actions within the switch capabilities when the EFSM enters a state labeled as misbehavior, including dropping or rerouting the packet, generating an alert and applying corrective actions such as modifying packet field(s), sending a copy of the packets to a remote server for further inspection and setting queue priorities. The proposed approach detects and mitigates protocol abuse in real-time without involving any remote controller.

III. Protocol Behavior Monitoring in the Data Plane

In this section, we describe our approach for monitoring stateful protocol behavior in the data plane. To this end, we leverage EFSM for modeling stateful protocol behavior and map the EFSM to a P4 program that can run on the data plane. We first describe how we formally represent an EFSM in Section III-A. Then in Section III-B, we present a mechanism for mapping an EFSM to a P4 program that can run on a programmable PISA target. Note that our proposed mapping procedure is generic and is not specific to any protocol. Finally, we briefly discuss about the checksum recalculation issue in Section III-C that may arise from modifying the packets while applying the EFSM actions.

A. EFSM model

Notation	Definition
S	States (stage of the behavioral process)
$I \subset S$	Initial states
E	Events (pattern matching applied to packet data)
$e[p]$	Value of parameter p extracted from event e
V	Persistent context variables
A	Actions, $a \in A, a = \{modify, drop, forward, write(v)\}$ with $v \in V$
C	Conditions, $c \in A$ is a function combining numerical or logical condition operator applied to persistent variables ($\in V$) and event parameter values
T	Transitions, $t \in T, t = s_1, c \rightarrow a, s_2$ ($s_1 \in S$, $s_2 \in S$, $c \in C$, and $a \in A$)

TABLE I: Key notations used in the EFSM model

We adopt the EFSM abstraction for modeling protocol behavior. While there are different ways to formally represent an EFSM [21], [22], in this paper we represent an EFSM by a 7-tuple (S, E, A, I, V, C, T) (summarized in table I), where:

- S is the set of possible states. In our case, S consists of all the protocol states that need to be tracked for detecting misbehavior.
- I is the set of initial states.
- E is a finite set of events. An event is triggered by the content of the monitored packets such as the presence of a TCP flag in a TCP packet. Events can be parametric [23] to store the values observed in events. We represent the value of parameter p of the event e by $e[p]$. For example, if $TCP.ACK$ is an event that matches a TCP packet with an ACK flag, $TCP.ACK[AckNo]$ represents its corresponding acknowledgement number.
- V is the set of context variables that are persistent and accessible by all states using read and write operations. One of the main advantages of EFSM is that the context variables can be leveraged to avoid creating numerous states. For example, tracking sequence numbers with a regular FSM would require one state for each of the 2^{32} possible values compared to using just one context variable in an EFSM.
- A is a finite set of actions to be performed. A includes modifications of context variables and other actions that can be performed within PISA switch capabilities such as modify, drop or forward a packet, or send an alert. Actions are split into two categories: packet forwarding actions, and arithmetic and logic operations to update the variables in V or the packet headers.
- C is the set of conditions. Each condition $c \in C$ is defined from event parameters and context variables. This is another major advantage of EFSM compared to regular FSM where transitions are defined only on symbols. With EFSM, more complex conditions can be described. More precisely, each $c \in C$ is a conditional expression composed of operands (events and context variables) to express logical or numerical calculations (*e.g.*, addition, logical AND, OR) and comparisons (*e.g.*, less-than and equal).
- T is the set of transitions, where $t \in T$ is defined as $s_1, c \rightarrow a, s_2$ denotes a transition from state s_1 to s_2 when condition c is satisfied and that results in the action a to be performed.

B. Mapping an EFSM to a P4 Program

For each connection or flow (we assume there are n flows or connections being tracked in the data plane), an EFSM is instantiated in the data plane using the P4 core primitives described in Section II-A. In what follows, we describe how the different components of an EFSM described in Section III-A are mapped to a P4 program. We also summarize and present the mapping process in Figure 2.

1) Maintaining persistent information: For each connection or flow, we need to maintain persistent information in the data plane, including the context variables of the EFSM and the state that the EFSM is in (*i.e.*, the current state). There is a

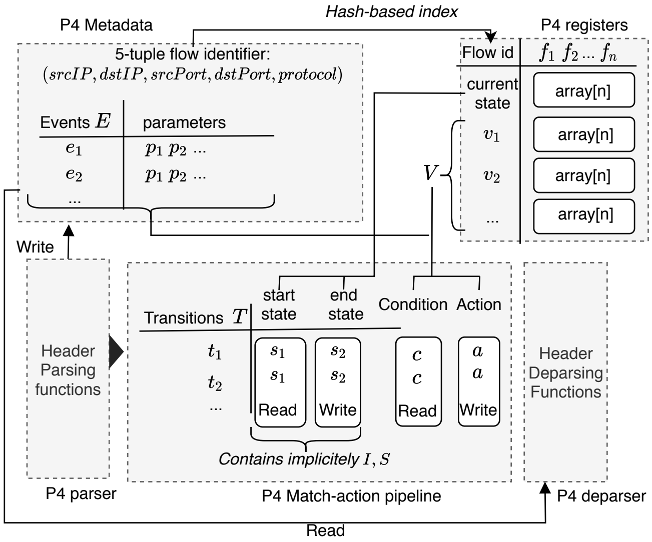


Fig. 2: EFSM mapping to P4 elements and associated I/O operations

one-to-one mapping between the number of executed EFSMs and the number of monitored flows. We formally define a flow as a set of packets $f = \{p_0, p_1, \ldots, p_n\}$ sharing the same set of attributes. Without loss of generality, we define a flow by the 5-tuple: $f = (srcIp, dstIp, srcPort, dstPort, protocol)$, where $srcIp, dstIp, srcPort, dstPort, protocol$ represent the source IP address, the destination IP address, the source port, the destination port and the transport protocol, respectively. This 5-tuple also forms the *flow key*, which we will later use for computing an index in a hash table. As shown in Fig. 2, the header fields in the 5-tuple flow key are defined as P4 metadata. Metadata variables are associated with the processing of each packet and are used for carrying information across the pipeline stages. They are deleted when the processing is completed. Metadata fields along with packet headers are declared in P4 and are populated by the parsing function.

Once the flow key is extracted from a packet, we compute a hash function $hash(key)$ to retrieve the persistent information corresponding to that flow. The hash function $hash(key)$, takes as input the flow key and outputs a flow identifier. The current EFSM state of each flow as well as the context variables from V are stored as entries in register arrays as shown in Figure 2. The flow identifier returned by the hash function acts as the indices of these arrays for retrieving the corresponding information (*i.e.*, the current state of the EFSM or the context variables of an EFSM). In total we need $|V|+1$ register arrays, each with n entries for keeping track of the persistent information of the EFSMs of n flows.

In many cases, we need to monitor behavior for both directions of a flow. For example, monitoring the 3-way TCP handshake requires monitoring both directions of a TCP flow. One solution to this problem proposed in the literature is to form the 5-tuple flow key in the order (dstIp, srcIp, dstPort, srcPort, protocol) if (dstIP > srcIP); otherwise form the 5-tuple flow key in the order (srcIp, dstIp, srcPort, dstPort, protocol) [24]. In this way, the source and destination IP address pair in the 5-tuple always remains the same for both directions of a flow. The same applies to the source and destination ports. However, for the sake of brevity in this section, we keep the regular 5-tuple flow definition. For applications where the forward and return paths may differ, the solution has to be deployed at the network edge to see both flow directions.

2) Event extraction: Events in E are retrieved by matching ingress packets against the patterns defining the events (*e.g.*, the presence of the ACK flag in the TCP header). Additionally, ingress packets are checked against the conditions defined on event parameters for event extraction (*e.g.*, an event is detected if a condition on TCP sequence number is satisfied). The result of aforementioned pattern matching and condition evaluation on packet header fields are captured in metadata fields that we define for tracking the events. These metadata fields are populated during packet parsing and can be read when the parsed packet is deparsed at the end of processing. The protocol header fields that must be extracted for performing such pattern matching for event detection are identified based on the events $e \in E$ and *event parameters* $e[p]$.

Algorithm 1 Packet Processing

Definitions:

- n: number of monitored flows
- $Current[n]$: register array of the current states
- $\forall v \in V, Val[n]$: register array of each context variable

Input: packet pkt

```
1: p ← parse(pkt)
2: id ← hash(p.srcIp, p.dstIp, p.srcPort, p.dstPort,protocol)
3: current ← Current[id]
4: find $t \in T, t = s_1, c \rightarrow a, s_2$ such that $s_1 = current$ and $c.check(p, V) = TRUE$
5: if t ≠ None then
6:     Current[id] ← $s2$
7:     a.apply(V,p)
8: end if
9: deparse(p)
```

3) Mapping of the state transition system: When a switch receives a new packet, the packet is processed according to Algorithm 1. While processing the packet, Algorithm 1 may trigger a state transition within the EFSM corresponding to the flow that the packet belongs to. In Algorithm 1, we assume that the current state and the context variable of the EFSMs are maintained in register arrays as described in Section III-B1. Except for events derived from ingress packets and persistent variables defined a priori, actions, transitions and conditions are implicitly defined within P4 actions.

The control logic of the P4 program starts at line 1 of Algorithm 1 with the parsing of the newly arrived packet. Then, the hash-based flow identifier is computed from the flow key to retrieve the current state of the corresponding EFSM (lines 2-3). The algorithm then searches for a possible transition fulfilling two conditions (line 4). First, the transition origin must be the current state. Second, the condition of the transition computed using the function $c.check(p, V)$ must evaluate to true. If a suitable transition is found in T, the transition is applied. The current state, the context variables and the packet metadata are modified according to the actions of the applied transition.

The conditions and the action logic (C and A) are mapped to the match-action pipeline in P4 in Figure 2. Read operations are used for checking if a transition can occur and the write operations apply modifications when a transition is applied. It is worth mentioning that all states, S, including the initial states, I, are embedded implicitly in the transitions defined in the P4 processing logic, eliminating the need to explicitly define them. Packet modifications include packet header field changes as well as changes to P4 standard metadata. The actions of the mapped EFSM model can be carried alongside other packet processing actions such as packet forwarding.

C. Checksum recalculation

An action in an EFSM consisting of modifying a packet field can be written as $(packet', y) = mod\_pkt(packet, x)$ that takes as input a packet $packet$ and outputs a new packet $packet'$ where $packet$ has one field with the value x changed to value y in $packet's$. If $x \neq y$, recomputing a checksum may be necessary depending on the protocol. As we will be focusing on TCP, the TCP checksum will need to be recomputed. Instead of performing an expensive full checksum recalculation, we incrementally update the checksum using the following simple arithmetic function [25]:

$$HC' = HC - \sim m - m'$$

Where :

- HC: old checksum in the packet header,
- HC': new checksum in the packet header,
- m: old value field including the former packet value field,
- m': new value of field including the modified packet value field, $\sim x$: the one's complement of x.

IV. Application to Optimistic ACK

In this section, we present the first use-case demonstrating the effectiveness of our proposed EFSM-based security monitoring mechanism: the detection and mitigation of Optimistic ACK attack on TCP congestion control [5].

A. Optimisitic ACK attack description

The Optimistic ACK attack was introduced in [5] as a way to mislead end-hosts to increase their TCP congestion window. This is accomplished by malicious receivers who acknowledge not yet received TCP segments to the senders. In the context of TCP, the congestion window limits the number of TCP segments that can be sent by an end-host without receiving an acknowledgement (ACK). TCP congestion control mechanisms (*e.g.*, BIC, CUBIC) typically start with a relatively small congestion window and keep increasing the window based on received ACKs during the slow start and congestion avoidance phases. Since receiving ACKs of in-flight segments is considered a sign of good network condition, the malicious receivers in the Optimistic ACK attack will mislead the sender and trigger an increase in congestion window. This in turn will cause the sender to converge to a sending rate that will have a detrimental impact on legitimate TCP connections sharing the same network. The authors in [5] proposed a solution to Optimistic ACK attack that requires changing the TCP implementation at the end-hosts. In contrast, we propose to leverage data plane programmability for detecting and mitigating the attack without any end-host modification.

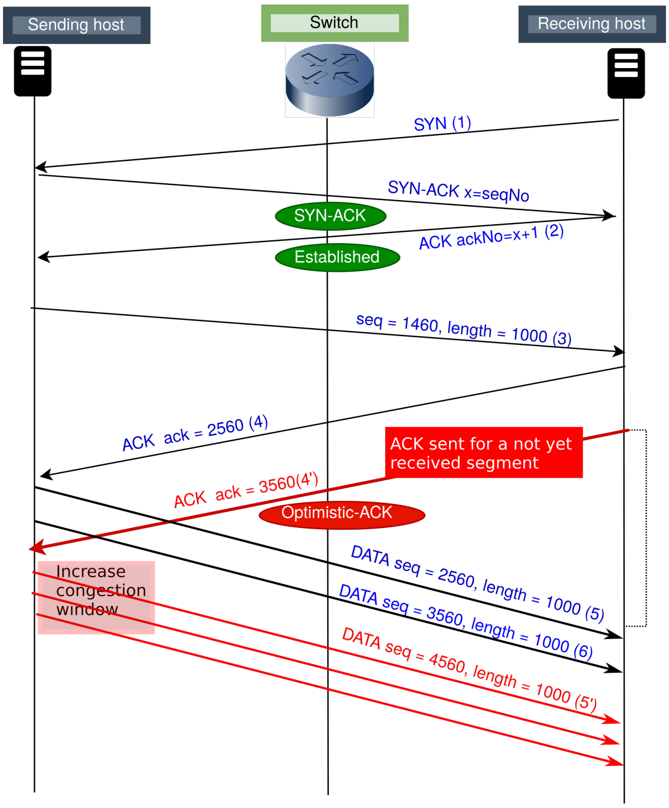

Sending host
Switch
Receiving host
SYN (1)
SYN-ACK x=seqNo
SYN-ACK
ACK ackNo=x+1 (2)
Established
seq = 1460, length = 1000 (3)
ACK ack = 2560 (4)
ACK sent for a not yet received segment
ACK ack = 3560(4')
Optimistic-ACK
Increase congestion window
DATA seq = 2560, length = 1000 (5)
DATA seq = 3560, length = 1000 (6)
DATA seq = 4560, length = 1000 (5')

Fig. 3: Optimistic ACK attack

We illustrate the Optimistic ACK attack through a sequence diagram in Figure 3. Here, a receiving host (*i.e.*, the attacker) is starting a normal TCP connection before abusing it. During normal operations at the beginning (in blue), the receiving host (the attacker) initializes the connection. After the 3-way handshake (step (2)), data transfer begins as usual. In this example, the sender sends a segment of 1000 bytes (step (3)) and the receiving host sends an ACK upon receiving the segment (step (4)). Then, the sender transmits two more segments in a row (step(5)-(6)) assuming a slow start.

The Optimistic ACK attack phase is highlighted in red in Figure 3). During the attack, the receiving host (the attacker) anticipates the reception of segments by sending ACK for segments that are not yet received (step (4')). The receiver (the attacker) sends these Optimistic ACKs in a way that they appear to be corresponding to "in-flight" segments. As a consequence, the sender increases its congestion window and sends the subsequent segments earlier than it should (step (5')). In this example, the sender sends three segments in a row after receiving the Optimistic ACK from the receiver. As a consequence of this attack, the sender believes that network conditions are better than the reality, resulting in a transmission rate faster than it should be. This way, the malicious TCP connection can exhaust bandwidth on some or all of the intermediary links on the path between the sender and the receiver, causing a possible denial of service attack. Furthermore, Optimistic ACKs can cause other legitimate TCP connections sharing links with the malicious connection to get unfair share of bandwidth.

Optimistic ACK can be considered an amplification attack because the attacker only needs to send many small ACKs while delegating the flooding of larger segments to a legitimate host, the victim. Note that the Optimistic ACKs may be received by the sender before the corresponding TCP segments are sent. However, in most of implementations, these anticipated ACKs will be ignored [5]. There is indeed no verification of the matching between received ACKs and the sent segments as it is not necessary for normal operations and adds overhead.

B. *EFSM model of Optimisitic ACK*

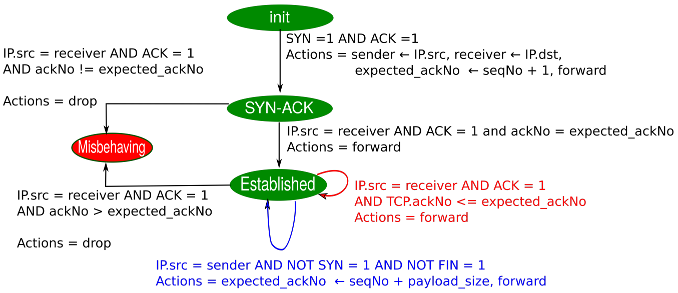

init
SYN =1 AND ACK =1
Actions = sender ← IP.src, receiver ← IP.dst,
expected_ackNo ← seqNo + 1, forward
IP.src = receiver AND ACK = 1
AND ackNo != expected_ackNo
Actions = drop
SYN-ACK
Misbehaving
IP.src = receiver AND ACK = 1 and ackNo = expected_ackNo
Actions = forward
Established
IP.src = receiver AND ACK = 1
AND TCP.ackNo <= expected_ackNo
Actions = forward
IP.src = receiver AND ACK = 1
AND ackNo > expected_ackNo
Actions = drop
IP.src = sender AND NOT SYN = 1 AND NOT FIN = 1
Actions = expected_ackNo ← seqNo + payload_size, forward

Fig. 4: EFSM for Optimisitic ACK detection and reaction

We model the normal TCP behavior and the possible Optimistic ACK attack in a single EFSM. We use the states represented in Figure 3 as the states in the EFSM in Figure 4 with an additional initial state called $init$. We define the events from the relevant TCP flags shown in Figure 3. Once the TCP connection is established, we use the conditions $SYN = 1$ and $FIN = 1$ to identify and ignore a possibly re-transmitted SYN and FIN (end of connection) segments for the tracked connection. The detection of the attack requires observing the TCP segments in both *sender-to-receiver* and *receiver-to-sender* directions. Therefore, we monitor bi-directional flows and determine the direction of a flow by checking if the source IP address of a packet, $IP.src$, is of the receiver (attacker) or of the sender. The $SYN - ACK$ state allows us to monitor the beginning of the connection/flow, initiate its monitoring and identify the receiver and sender in the bi-directional flow. The identified receiver and sender IP addresses are saved into context variables, $receiver$ and $sender$, since they are required for monitoring the connection.

To detect the attack, we have to observe a deviation between the maximum acceptable acknowledgement number (*i.e.*, $expected\_ackNo$) and the acknowledgement number sent by the potential attacker (receiver). We compute $expected\_ackNo$ from the last seen sequence number $TCP.SeqNo$ and the payload size $payload\_size$. The $payload\_size$ in turn is calculated from IP header length ($IP.ihl$), IP total length ($IP.len$) and TCP header length ($TCP.dataOffset$) fields extracted during parsing:

$$payload\_size = IP.len - ((IP.ihl + TCP.dataOffset) \times 4)$$

All these per-packet data are stored as packet metadata in P4 while $expected\_ackNo$ is tracked using the context variable since this needs to persist across the TCP segments of a connection (Section III-A). For this attack, we drop the packet when the attack is detected.

Therefore, the Optimistic ACK attack EFSM model can be represented as follows:

- $S = \{init, SYN - ACK, Established, Misbehaving\}$;
- $I = \{init\}$;
- $E = \{SYN = 1, ACK = 1, FIN = 1\}$ with the following parameters: $TCP.ackNo$ and $TCP.SeqNo$ are the acknowledgement and sequence number of the current packet and $IP.src$ and $IP.dst$ are respectively the source and destination IP address;
- $V = \{expected\_ackNo, sender, receiver\}$;
- Annotations of arrows in Figure 4 represent A, C and T

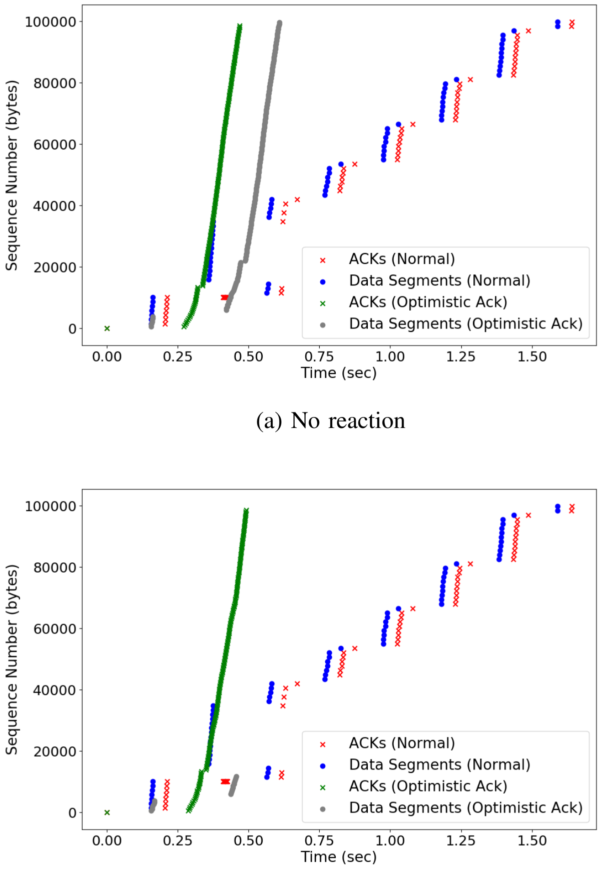
It is worth mentioning that the EFSM in Figure 4 does not represent future actions after the attack is detected. This is dependent on the operator policy and is orthogonal to the EFSM model for detecting misbehavior. One possible action can be to drop all subsequent opportunistic ACKs for the malicious TCP connection.

V. Optimistic ACK Evaluation

We have implemented the data plane program for Optimistic ACK detection and mitigation using P4-16 [19] and compiled it for the bmv2 software switch target [26]. We have used mininet [27] and p4app docker image [28] for conducting the experiments. Mininet and bmv2 switches are indeed not suitable for evaluating performance in a realistic setting, thus a baseline scenario for comparison is used in different experiments. However, they provide us a way to perform functional validation of the solutions.

1) Attack Mitigation: The goal of our first experiment is to evaluate the effectiveness of our approach in mitigating the Optimistic ACK attack from the data plane. To accomplish this we perform an experiment similar to that done in [5]. We consider a simple topology with two hosts (a sender and a receiver) connected to the same switch. We perform the experiment both with and without detection and mitigation in the switch program. We generate the attack using the tool available in [29].

Figure 5 (a) illustrates how the attack unfolds comparing sequence numbers of data segments and acknowledgment numbers assuming a normal client or an attacker. After the first few data segments, the attacker starts its malicious behavior at 0.25s by sending a stream of Optimistic acknowledgements (plotted in green). As a consequence, the sender keeps increasing its congestion window and increases the speed of sending new segments (plotted in grey). As we can see, the server completes its transmission (plotted in grey) significantly earlier than it would normally do without the presence of Optimistic ACK attack (plotted in blue).



(a) No reaction

(b) Reaction

Fig. 5: Optimistic ACK in action

Figure 5 (b) demonstrates how our monitoring strategy effectively detects the Optimistic ACK attack and mitigates its impact. When the switch detects Optimistic ACKs, the EFSM corresponding to that TCP connection transitions to the misbehaving state. Consequently, the anticipated ACKs are dropped at the switch and no longer reach the server. Therefore, the server's congestion window does not incorrectly grow, hence, it does not significantly increase the data transmission rate.

2) Throughput evaluation: In this experiment, we evaluate the impact of the Optimistic ACK attack flow on the achieved throuhgput of a normal flow sharing the same path. The topology for this experiment is composed of two switches and seven hosts (Figure 6). Here, host h1 is the sender (the attacked server). We vary the number of attackers between 1 and 4 (hosts h4 to h7). We generate a normal flow between host pair (h2, h3) using iperf and measure the impact of the attack on its throughput. The link between the two switches is shared by all flows and is thus the bottleneck. We consider two scenarios: $Reaction$, where the switch applies the mitigation process and $noReaction$, otherwise. The results are presented in Figure 7.

Each box in Figure 7 represents the distribution of results obtained from 10 emulation runs of 60 seconds each. For a first baseline experiment, $iperf + Normal$, we have a non misbehaving flow alongside the iperf flow between h2 and h3. In this case, we observe similar behavior for both the $noReaction$ and $Reaction$ scenarios. The observed throughput variation due to the EFSM-based monitoring is very low.

Once we have a baseline, we incrementally increase the number of attackers. First, with a single attacker ($iperf + 1\ attacker$), the throughput degrades in the $noReaction$ scenario compared to the baseline, $iperf + Normal$. When we activate the reaction process, the throughput improves and gets even better than the $iperf + Normal$ case. This can be explained by the fact that the attacker's Optimistic ACKs are dropped, causing the server to stop transmitting further data segments. Consequently, the bandwidth of the bottleneck link becomes fully available for the iperf flow. When the number of attackers increases, the impact of the attack becomes higher as expected. In case of 2 attackers, (*i.e.*, 66% of the hosts being attackers), the reaction process recovers the throughput back to approximately the same level as the $iperf + Normal$ scenario. Starting from three attackers, the throughput is not completely recovered as we can see from the box-plot for the $Reaction$ scenario. In this case, switch s2's ingress queues are flooded with Optimistic ACKs (on the link to the attackers). Although the Optimistic ACKs are dropped and their impact is mitigated on the shared link s1 – s2, the ACKs still need to enter switch s2 for processing. This consumes the switch resources, impacting the throughput recovery. Note that 3 attackers correspond to having 75% of the hosts being attackers. This is a very aggressive scenario and it becomes even worse with four attackers.

3) Switch processing time evaluation: In this experiment, we evaluate the overhead of running our mitigation solution in the data plane by measuring the increase in packet processing time in the switch. The results of this experiment are presented in Figure 8. The two groups of box-plot represent when forwarding is performed with monitoring disabled ($forward$) or enabled ($reaction$).

We deployed a topology with a single switch and two hosts, a sending and a receiving host. We generated 1000, 5000 and 10000 TCP flows and measured the per packet processing time in the switch. As we can observe from Figure 8, the median value is 2.9 ms for the $forward$ configuration and 3.3 ms for the $reaction$ configuration. The switch processing time overhead of our reaction process is 0.4 ms (≈14% additional processing time compared to $forward$). As shown also in Figure 8, our solution has no significant change in processing time overhead with respect to the number of flows to monitor. Note that both the processing time and the overhead will be significantly improved for a hardware target, which we plan to explore as a future extension.

VI. Application to ECN

In this section, we apply our method to model, detect and mitigate misbehaving end-hosts abusing the ECN protocol mechanism.

A. ECN background and attack description

TCP end-hosts typically use end-to-end congestion signals such as packet loss or round-trip-time for adjusting their

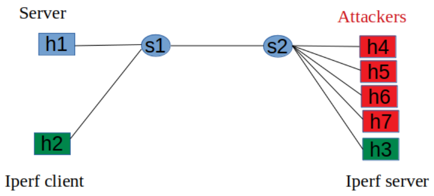


Fig. 6: Topology

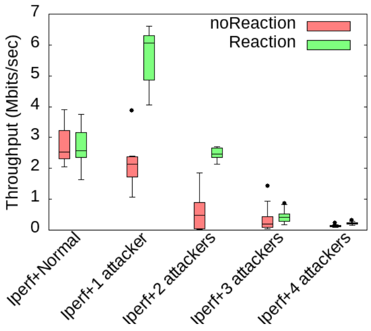


Fig. 7: Optimistic Ack impact on the normal flow

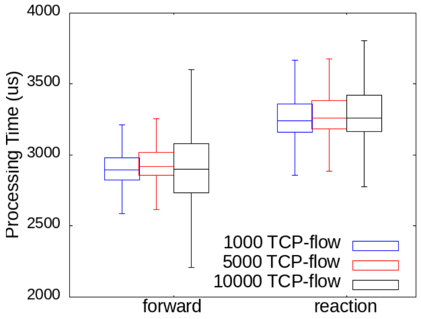


Fig. 8: Switch processing Time

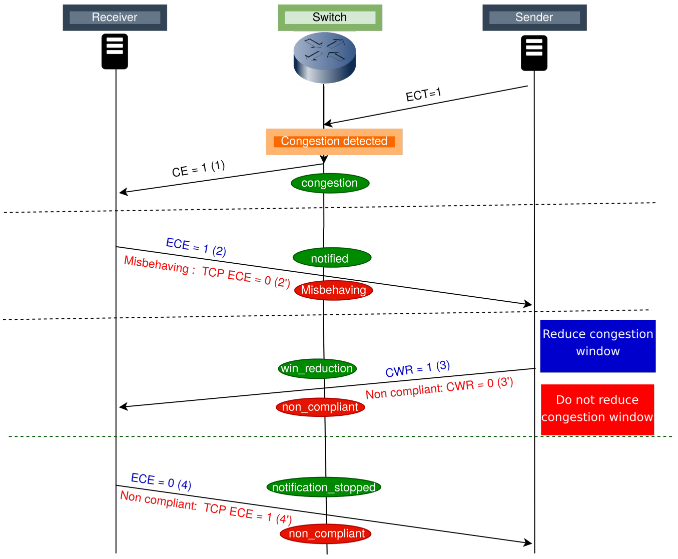


Fig. 9: ECN (normal behavior in blue, misbehavior in red, ellipses represent EFSM state updates)

congestion windows. ECN was proposed as a mechanism for network devices experiencing congestion to send congestion signals to end-hosts [30]. The ECN RFC [7] defines the following codepoints and fields for both IP and TCP protocols:

- Congestion Experienced (CE) and ECN-capable Transport (ECT) codepoints for the DSCP field of IP header.
- ECN-Echo (ECE) and Congestion Window Reduced (CWR) flags in the TCP header.

However, the design of ECN introduces the possibility of having misbehaving end-hosts in the network, *i.e.*, end-hosts that do not fully conform to the protocol specification. We illustrate such misbehavior (messages in red) along with the expected normal behavior (messages in blue) of ECN enabled TCP end-host in Figure 9. During the TCP three-way handshake phase (not shown in the Figure), TCP end-hosts negotiate the use of ECN. Following the TCP three-way handshake, the ECN protocol behaves as follows:

- A congested switch detects an ECN-capable TCP connection (ECT set in IP header) and marks the corresponding packet with CE ((1) in Figure 9);
- After receiving a packet with CE, the receiver becomes aware of the congestion and informs the sender by setting the ECE flag in the TCP header ((2) in Figure 9);
- Once the sender receives a packet with the ECE flag set, it reacts by reducing its congestion window. Then, the sender sets the CWR flag to inform the receiver ((3) in Figure 1), which in turn stops sending congestion notification by unsetting ECE ((4) in Figure 9).

ECN RFC defines a possible misbehavior of an end-host announcing itself as ECN-capable but ignoring congestion notification from the switch [7]. As shown in Figure 9, once a switch notifies about congestion, the receiving host can misbehave by not echoing back the congestion information to the sender, *i.e.*, set the ECE flag to 0 ((2') in Figure 9). As a result, the sender does not reduce the congestion window ((3') in Figure 9). Misbehaving flows can degrade network performance and create a denial service for the benign flows at the expense of their own loss in throughput. This is why the RFC recommends that such flows must be identified and handled.

For the sake of completeness with respect to the original ECN procedure, Figure 9 introduces two non compliant behaviors. First, the receiver can keep sending packets with ECE set ((4') in Figure 9) even though the sender has already reduced its congestion window, forcing the sender to further reduce its congestion window. Second, the sender can ignore the notification sent back from the receiver ((2) in Figure 9) by not reducing the congestion window ((3') in Figure 9). Similarly the sender can reduce its congestion window without notifying through setting the CWR bit. In both cases, the switch cannot deduce if the congestion window has been really reduced. That is why these behaviors are qualified as non compliant rather than misbehaving. These non-compliant behaviors will allow us to evaluate our method in a more complex scenario.

B. EFSM model of ECN

We model the expected and unexpected (misbehaving and non compliant) behaviors of ECN capable end-hosts in a single EFSM. Each time a packet is received, the state of the EFSM illustrated by an ellipse in Figure 9 is updated. Normal states are colored in green while all unexpected states are colored in red. *init* represents the initial state of the EFSM, $I = \{init\}$. The EFSM model is instantiated into the switch and the transitions are triggered by events which are derived from ingress packets as explained in section III-B2, notably ECN related flags in these scenarios. Based on these definitions, we illustrate the EFSM corresponding to Figure 9 in Figure 10. It is defined as follows:

- $S = \{init, congestion, notified, win\_reduction, notification\_stopped, misbehaving, noncompliant\}$;
- $I = \{init\}$;
- $E = \{ECT = 1, ECE = 1, ECE = 0, CWR = 1, CWR = 0\}$ with the following parameters: $IP.src$ and $IP.dst$ are respectively the source and destination IP address;
- $V = \{dstCE\}$;
- Annotations of arrows in Figure 4 represent A, C and T

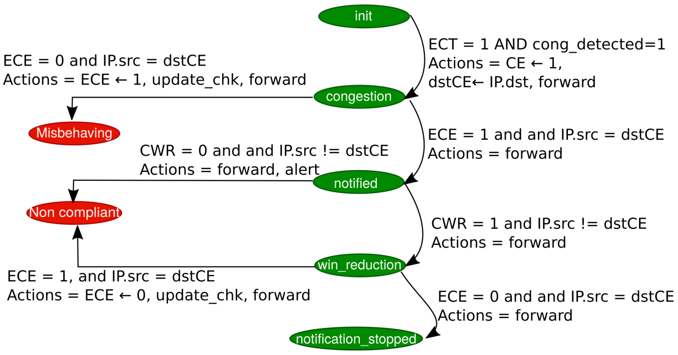

init
ECT = 1 AND cong_detected=1
Actions = CE ← 1,
dstCE← IP.dst, forward
ECE = 0 and IP.src = dstCE
Actions = ECE ← 1, update_chk, forward
congestion
Misbehaving
ECE = 1 and and IP.src = dstCE
Actions = forward
CWR = 0 and and IP.src != dstCE
Actions = forward, alert
notified
Non compliant
CWR = 1 and IP.src != dstCE
Actions = forward
ECE = 1, and IP.src = dstCE
Actions = ECE ← 0, update_chk, forward
win_reduction
ECE = 0 and and IP.src = dstCE
Actions = forward
notification_stopped

Fig. 10: EFSM abstraction for ECN

Unlike the Optimistic ACK scenario, no additional information must be maintained alongside the current state since the states themselves are self-contained in terms of necessary information. However, recognizing the direction of packets is still necessary. Therefore, we use a single context variable $dstCE$ for identifying the host that is supposed to relay the congestion notified by the switch (*i.e.*, the receiver in Figure 9). To properly set the $dstCE$ variable, source ($IP.src$) and destination ($IP.dst$) IP addresses are needed, which are anyway extracted to create the 5-tuple flow key. Once a misbehavior or non compliant behavior is detected, we take corrective actions when possible.

For the misbehaving and the second non compliant cases, the receiver must react according to the congestion signal (with either $CE = 1$ or $CWR = 1$). Therefore, the behavior can be corrected by setting or unsetting ECE accordingly. Doing so will also require TCP checksum to be updated (*action* $update\_chk$ as introduced in section III-C). Correcting the second non compliant case is also important because a receiver with an incorrect ECN implementation would penalize itself by forcing the sender to continue reducing its congestion window. Note that our approach can also be employed to detect incorrect protocol implementation at end-hosts aside from being used for the security use cases.

For the first non compliant case, the switch cannot effectively verify that the sender has reduced its congestion window. As a result, it is impossible to deduce from the switch if the CWR flag must be really set to one in the first non compliant case. Therefore, we resort to sending an $alert$ and $forward$ the packet. The exact definition of the $alert$ depends on the switch capabilities and can be of different kinds such as creating entries in the log files of the switch, sending a packet to the controller, or generating a postcard [31], [32]. The exact details of such action is out of the scope of this paper. In our prototype implementation, we let the packet pass through and change the EFSM state to non compliant.

It is worth mentioning that the congestion is detected by the switch itself enabling so the transition from $init$ to $congestion$. It corresponds to the condition $cong\_detected = 1$ in Fig. 10. In practice, this variable is derived from the occupancy level of the switch queues (intrinsic metadata). When the occupancy reaches a marking threshold, the packets are marked and $cong\_detected$ is set to 1.

VII. ECN Evaluation

All experiments were performed using the same setup as for Optimistic ACK, *i.e.* bmv2 software switch [26] with mininet [27], p4app docker container [28] and P4-16 [19].

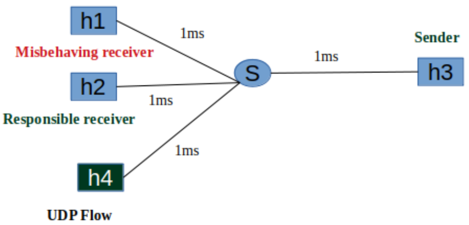


Fig. 11: Experimental topology used in mininet

A. Bandwidth share and throughput evaluation

To assess the impact of the use or misuse of ECN, we evaluate if the bandwidth is correctly shared during congestion assuming the topology presented in Fig. 11. It is composed of one switch and four hosts (h1, h2, h3, h4). By default, all TCP hosts are ECN-capable and the link latency is set to 1ms. To create congestion, we limit the link capacity to 2000 packets/sec and a UDP flow between h4 and h3 is generated with 8 Mbits/sec to flood the network. The size of UDP packets is set to 400B to increase the number of packets in the network and in the switch queue.

We generate TCP flows using iperf between h1 and h3, h2 and h3. h1 acts as a misbehaving host by not echoing the congestion notification ((2') in Figure 9), (*i.e.* not setting the TCP ECE flag). h1-h3 and h2-h3 flows are thus qualified as misbehaving and normal, respectively. The switch queue capacity is set according to the Bandwidth Delay Product rule (BDP) [33]: $queue\_capacity = RTT * Network\_bottleneck\_capacity = 8$ packets, in our setup since the lowest RTT between two hosts is 4ms and the bottleneck capacity is limited by the queue rate of 2000 packets/sec.

We evaluate the misbehaving ECN use-case under different settings for the ECN marking threshold, *i.e.*, the number of packets in the switch queue for triggering the sending of congestion notification. First, we consider recommendations from the data center networking literature. Second, we consider

ECN with an active queue management mechanism, namely, Random Early Detection (RED) [34], which represents a more generic scenario.

In the case of data center networks, the research literature proposes several ECN marking schemes, the majority of them being deterministic, *i.e.*, switches mark packets when switch queue length is higher than a pre-determined threshold. There are different recommendations for the ECN threshold in DCN. DCTCP [30] recommends an ECN marking threshold greater than $queue\_capacity/7$. Another work on ECN presented in [35] recommends a marking threshold of: $queue\_capacity/\sqrt{n}$, where n is the number of flows. In the first case, the threshold should be greater than 1.14 in our setting. In the second case, the marking threshold should be 4.61. However, we decided to vary the marking threshold from 2 to 7 to consider different values including those calculated from the recommendations (2 and 5 obtained by rounding the numbers to the next integer).

In Figure 12, we present the throughput of the normal flow (h2-h3) and the misbehaving flow (h1-h3) side-by-side from the following scenarios:

- noMisbehaving-forward: the baseline scenario where all hosts follow ECN specification and the switch forwards packets without modification (neither detection nor reaction activated). Even the flow h1-h3 behaves as expected.
- Misbehaving-noReaction: h1 misbehaves but the switch continues to forward packets (neither detection nor reaction activated).
- Misbehaving-ECEreaction: similar to the previous case but the switch implements ECN misbehavior detection and applies corrective measures. The switch partially implements the EFSM of Figure 10 with $S = \{init, congestion, misbehaving\}$.

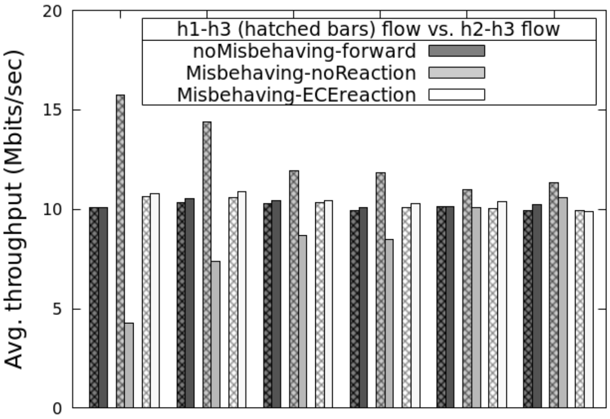


Fig. 12: Misbehaving and normal flow throughput depending on the marking threshold

The bars in Figure 12 represent the average throughput of 10 emulation runs of 120 seconds each. In the case of no misbehaving flow (noMisbehaving-forward), a fair bandwidth share is observed regardless of the marking threshold (each flow is getting ≈10 Mbits/sec). However, the misbehaving flow takes the largest share of the available bandwidth for low marking thresholds (2, 3, 4, 5) when there is no detection. In that case, the misbehaving host ignores the congestion notification and increases its sending rate much higher than the benign flow. Indeed, the benign flow reduces its congestion window in response to congestion notification, leaving more queue space to the misbehaving flow. When increasing the marking threshold, congestion notifications are sent later, (*i.e.*, when the queue is more occupied). Therefore, for higher ECN marking thresholds (6, 7) the impact of the misbehaving flow on the normal flow is reduced compared to low marking thresholds. Even with the most aggressive recommendation [35] with a threshold of 5, the misbehaving flow gets 11.8 Mbits/sec while there is only 8.4 Mbits/sec left for the normal flow (59% vs. 42% share). In the presence of the reaction (Misbehaving-ECEreaction), the bandwidth is properly distributed and an equitable bandwidth share is observed between the flows in all cases. This demonstrates the effectiveness of our solution in detecting and reacting to the attack.

RED is an active queue management mechanism that drops packets with a certain probability in anticipation of network congestion [34]. In our case, we employ RED for probabilistic packets marking with congestion notification instead of dropping the packets. When the average queue length in a switch is between two thresholds $minth$ and $maxth$, an incoming packet is marked with a certain probability. The marking probability is a function of queue length and changes linearly between 0 and 1 [34]. However, if the average queue length becomes higher than $maxth$, then all the incoming packets are marked [34]. For setting the $maxth$ and $minth$ parameters of RED, we set $maxth$ to at least twice $minth$ following the recommendation in [34]. Since the queue capacity is 8 in our experiments, we used the following combinations of ($minth$–$maxth$) tuples: (2-4), (2-5), (2-6), (2-7), (3-6), and (3-7) for covering all the possible combinations in our experiments.

We have implemented the RED algorithm in P4 based on the implementation from [36]. We use the instantaneous outgoing queue occupancy made available by bmv2 as intrinsic metadata for reacting faster to traffic bursts [37]. Note that bmv2 provides queue occupancy in terms of the number of packets. Therefore, we slightly modify the original RED algorithm that used the number of bytes to determine queue length. However, we keep the packet sizes the same in our experiments, hence, the number of packets is directly proportional to the number of bytes. Since P4 targets do not support floating-point operations, we pre-calculate the probabilities, scale the values to be between 0 and 255, and store the probabilities in a match-action table in the data plane. For each incoming packet, the queue length is matched against the entries in this table to obtain the probabilities. The obtained probability is then compared against a random number to decide if the packet must be marked [38].

We report the result of our experiments using RED in Figure 13. We again observe that applying our technique enables ensuring a fair bandwidth share when an attack takes place, reinforcing the effectiveness of our solution. Similar to the deterministic threshold case, we observe a higher impact of the attack when the packets are marked at lower queue

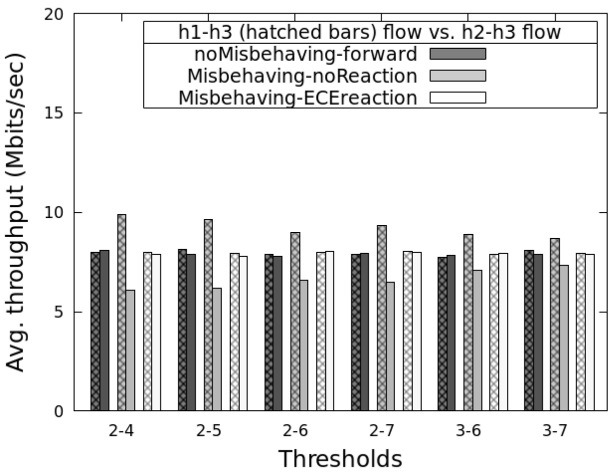


Fig. 13: Misbehaving and normal flow throughput depending on the marking threshold using RED

occupancy. In the case of RED, this scenario occurs for lower $minth$ values, when packets are likely to be marked even when the queue is not substantially occupied. For instance, for the (2-7) combination, the misbehaving flow reaches ≈ 9.31 Mbits/sec, forcing the benign flow to reach only ≈ 6.48 Mbits/sec (58.96% vs. 41.03% share). Finally, we notice each flow achieving ≈ 8 Mbits/sec throughput, which is lower than that reported in Figure 12. This is because applying RED also introduces additional per-packet processing overhead within bmv2, hence, reducing the throughput.

B. *Switch processing time evaluation*

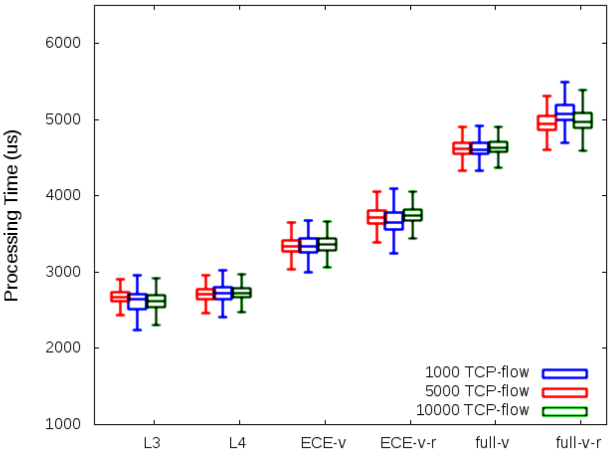


Fig. 14: Switch processing time

In this experiment, our goal is to evaluate the runtime overhead per packet. We consider different configurations:

- L3: parsing is limited up to the IP header.
- L4: L3 + TCP header parsing.
- ECE-v: the state machine is partially implemented to model the normal behavior; states S are restricted to $\{init, congestion, notified\}$ in Figure 10.
- ECE-v-r: the state machine is partially implemented to monitor and react against the misbehavior, ((2') in figure 9); S is restricted to $\{init, congestion, notified, misbehaving\}$.
- full-v: the full state machine is implemented for misbehaving and non compliant flow verification except for the corrective actions and checksum recalculation.
- full-v-r: similar to full-v but including the reaction against the ECN misbehavior ((2') in Figure 9) by setting ECE to 1 and so including checksum recalculation.

In this evaluation, the full-v and full-v-r scenarios are considered to estimate the overhead induced for monitoring the whole ECN state machine in the data plane including misbehaving and non-compliant flows verification. We deploy a simple topology with a single switch and two hosts (server and client). We generate 1000, 5000 and 10000 flows between the hosts and report the processing time per packet in Figure 14. The median value is 2.7 ms, 2.7 ms, 3.5 ms, 3.7 ms, 4.6 ms and 5 ms for L3, L4, ECE-v, ECE-v-r, full-v and full-v-r scenarios, respectively. Parsing TCP does not add a significant cost compared to IP parsing. As expected, the more complex the EFSM, the higher the overhead. It is worth mentioning that corrective actions incur less overhead than monitoring using a more complex EFSM when comparing the increase between ECE-v and ECE-v-r and between full-v and full-v-r. Therefore, we recommend a partial implementation of the EFSM to react only against the misbehavior since the non compliant flows verification induce a substantial overhead. Therefore, before deploying a protocol compliance verification solution, it is necessary to verify its usefulness and its adequacy to network performance conditions. Finally, as shown in Figure 14, our approach is scalable with respect to the number of flows to monitor in parallel similar to the Opt-ACK use case.

These results have been obtained using a deterministic marking threshold. This is because an additional active queue management mechanism for ECN marking is independent of our contribution, and including that for processing time evaluation will not give a real picture. Note that we also evaluated the processing time with RED included and found it to be ≈4.1 ms for the ECE-v-r scenario. This additional processing time explains the decrease in throughput between Figure 12 and Figure 13.

VIII. Discussion

A. *TCP connection tracking*

We track each individual TCP connection in the data plane as a separate flow. For maintaining per-flow persistent data across packets of a flow, we rely on hashing the 5-tuple flow key. TCP connection tracking through flow key hashing can lead to hash collisions that we address in Section VIII-C. After a TCP connection terminates we do not update the associated EFSM anymore. The exact mechanism of how connection termination is detected is out of the scope of our work. A hardware switch can rely on a variety of mechanisms such as detecting FIN and RST flags in TCP segments [39] or using a timeout mechanism for evicting terminated TCP connections when a new flow with the same hash occurs. For the latter, the hardware target must be capable of precise timestamping (available in P4 hardware switches [40]). Furthermore, additional register entries per tracked connection would be required for storing the last seen timestamp as a context variable.

Another issue that may arise for TCP connection tracking in the data plane is sequence number wrap-around, (*i.e.*, when sequence number reaches its upper bound ($2^{32}-1$)), it restarts from 0. This is particularly problematic for use-cases where we need to track the sequence numbers in each flow such as in the optimistic ACK EFSM. The TCP sequence number wrapping issue can be solved by employing techniques such as serial number arithmetic [41] or by enabling the TCP segment timestamp option [42]. Note that the latter will require tracking the last seen timestamp of each TCP connection as in the timeout based flow tracking scenario described above. Both of these techniques can be implemented in the P4 language and programmable hardware target constraints [19].

B. Scalability and Memory overhead

A possible implementation issue is the support of P4 externs such as registers, necessary for maintaining persistent data across the segments of tracked TCP connections. Although P4 leaves the externs to be target-specific, registers are the most widely used externs and are expected to be supported by most P4 targets. For example, in [43], authors propose a P4 compiler for various FPGA hardware including stateful objects. The implementation of our solution on programmable hardware raises the issue of memory constraint. Although metadata variables that carry information between tables has a negligible bit overhead [13], the scalability of our method is limited by the available register memory in the hardware targets. As highlighted in Section III, we need $|V|+1$ registers per flow: one for maintaining the current state of the EFSM and one for each context variable. For n flows, our method requires $n \times (|V|+1)$ register entries. For example, the EFSMs for the ECN and the Optimistic Ack attacks have $|V|=1$ and $|V=3|$, respectively. Assuming 32 bit values in registers, tracking 10,000 TCP connections will lead to allocate less than 64kB and 128kB, respectively, for these two use-cases. In comparison, state-of-the-art programmable hardware provides sufficiently large memory to accommodate state registers capable of holding tens of thousands of active TCP connections. For example, P4FPGA [43] can implement up to a 288 bits key for TCAM or hash-based memory and can fit up to 93K entries. Indeed, memory requirement will be higher with a higher $|V|$. In that case, we can employ optimizations such as lowering the size of register entries below 32 bits. In the current version, the number of states can be up to $2^{32}-1$, sufficient to support any use case.

C. Hash collisions

The limited hardware resources in the data plane can pose challenges for tracking per-flow state [24], [44]. We follow the approach presented in state-of-the-art such as [24], [44] and maintain the hash of the 5-tuple flow key rather than maintaining the flow key itself to reduce the width of flow key used in hardware. This approach creates the risk of multiple flow keys getting mapped to the same hash value, (*i.e.*, hash collision). For a real deployment we can leverage techniques such as those described in [24], [44] for tackling the hash collision issue. Using multiple hash functions in conjunction with multiple hash tables reduces the number of collisions [24]. However, collisions are often unavoidable and therefore must be detected by employing methods such as checking if the sequence number stored against a hash value is correct [44]. Some existing hardware switches include mechanisms for handling hash collisions, however, these mechanisms are hardware-specific and cannot be generalized.

IX. Related work

Traditionally, network monitoring has relied on tools such as SNMP and NetFlow or sFlow. Their stateless nature have forced network operators to use network appliances [11], [12] for monitoring stateful protocols such as TCP at the expense of additional operational overhead. The advent of OpenFlow-based SDN (Software-Defined Networking) and more recently, programmable data planes enabled by P4, have created new opportunities for stateful network monitoring. In the following, we first discuss the research literature on stateful monitoring enabled by the aforementioned technologies. Then, we focus our discussion on the state-of-the-art in detecting and mitigating TCP protocol abuse.

A. Stateful Monitoring in the Data plane

SDN has brought new capabilities for network monitoring. Initially, the OpenFlow-based SDN [45] centralized controller has the capability of dynamically controlling parameters such as the monitoring frequency and the granularity of flow rules to monitor. This motivated a significant body of research [46] with a particular emphasis on addressing the trade-off between monitoring accuracy and overhead in terms of data collection bandwidth [47], [48] and flow table entries [49], [50].

Despite providing additional flexibility over traditional network monitoring, OpenFlow does not enable stateful monitoring in the data plane. To address this limitation, several research works proposed extensions to OpenFlow. For instance, OpenState [51] proposes an OpenFlow extension that enables stateful monitoring using regular FSMs. However, as discussed earlier, regular FSMs may suffer from state explosion since they cannot compress states into variables storing persistent values as in EFSMs. Another OpenFlow extension proposed in the literature is SDPA [52]. The authors in [52] introduce a new component to manage state machines in the switch, however, their proposal is hardware specific (*i.e.*, for FPGA in [52]). Another OpenFlow data plane modification has been proposed by OFX [53], which uses an external agent running on OpenFlow switches to handle stateful monitoring. In contrast to the purely data plane based solutions, the authors in [54] propose a hybrid control and data plane approach for maintaining state machines. In the same vein, Oko [55] proposes to extend OpenFlow capabilities with extended Berkley Packet Filter (eBPF) [56] for stateful processing. However, Oko is exclusively limited to Linux-based software switches.

The recently emerging PISA architecture and the accompanying P4 programming language has inspired many research works leveraging stateful dataplane processing such as load-balancing [44], [57], application acceleration [58], [59] or DDoS detection [15], [60], [61], among others. A substantial body of recent research has taken advantage of the flexible packet parsing, cross-packet state retention, and the limited computational capability offered by PISA switches for advanced network monitoring. For instance, FlowRadar [62] and TurboFlow [63] proposed mechanisms for collecting NetFlow like records for all the flows passing through a switch as opposed to sampling like in NetFlow. The authors extend their approach proposed in [63] for supporting dynamic and concurrent flow queries in [64]. Another line of research focuses on computing approximate summary of network traffic directly in the data plane using data stream sketches [65], [66]. Besides these general directions, many works have focused on performing specific monitoring tasks leveraging data plane programmability such as heavy-hitter detection [67]–[70], congestion monitoring [71]–[73], network connectivity monitoring [74], and machine learning based packet classification [75].

Our approach of leveraging EFSM for modeling stateful protocol behavior in the data plane shares basic principles with XTRA [76] and FlowBlaze [77]. XTRA [76] provides a domain specific language and an EFSM abstraction for deploying transport layer functions in the data plane. XTRA also proposes a timer management mechanism in the data plane and demonstrates its viability by implementing timer-based applications such as SYN-proxy. The key difference between XTRA and our approach is that the former is target specific (*i.e.*, NetFPGA and software switch), whereas we build on a higher level of abstraction (*i.e.*, the P4 language) to support a wide range of hardware and software targets. FlowBlaze [77] augments the Reconfigurable Match Tables (RMT) model of PISA switches with dedicated tables for realizing EFSMs and a language for utilizing those tables. Indeed, a switch architecture similar to that of FlowBlaze will ease our implementation effort. However, in this work, we rely on the RMT model for its generality and higher adoption in hardware P4 targets [16], [78].

B. TCP Protocol Abuse: Detection & Mitigation

TCP protocol was not originally designed with security considerations in mind. As a consequence, many exploits have taken advantage of protocol corner cases for launching different attacks. These attacks are mainly targeted towards manipulating the way congestion window converges, in this way directly impacting the transmission rate along the TCP connection with the goal of flooding the network. Savage *et al.* introduced the Optimistic ACK attack addressed in our work along with two other attacks, namely, ACK division (dividing one ACK into multiple smaller ACKs) and spoofed duplicate ACK attacks (sending multiple forged duplicate ACKs for the last received segment) [5]. The ECN RFC raised the concern that a misbehaving host can unjustly manipulate the sender congestion window like the other attacks [7] as described in details in section VI. As a result of these attacks, the malicious TCP connection can flood the network or can cause unfair sharing of the network bandwidth.

The solutions proposed to tackle attacks from misbehaving TCP end-hosts require changing the TCP implementation at the end hosts. For instance, in case of Optimistic ACK, the authors in [5] proposed a nonce solution where the TCP sender fills each sent segment with a unique random value. The TCP receiver and sender maintain the cumulative nonce sum of all acknowledged segments. Each time a receiver sends an ACK it echoes the nonce value sent by the sender. In that case, the sender can verify that the ACKs sent by the receivers have been really sent. The authors in [6] proposed another solution to the Optimistic ACK attack that randomly drops segments at the sender side. Therefore, when the sender gets an Optimistic ACK for one of the intentionally dropped segments, it can identify the receiver as a misbehaving one. Besides requiring TCP implementation change, this solution also penalizes the legitimate receivers. Note that solely end-host modification based approaches might not be effective for the ECN attack use-case [7].

Jero *et al.*, proposed an offline mechanism for misbehaving TCP end-host detection. They propose to collect logs from end-hosts and compare the resulting TCP performance from the expected performance obtained from a testing environment [9]. This solution is effective for attack forensics, however, it is ineffective for online detection. Since some of the misbehaviors are attributed to incorrect implementation, Kothari *et al.* proposed to use symbolic execution techniques for verifying the protocol implementations [10]. However, this approach is limited to a specific operating system, implementation and language. In contrast to the state-of-the-art works, we perform real-time detection and mitigation of such attacks in the network without requiring any change to the protocol specification or implementation.

X. Conclusion

This paper introduced an EFSM-based security monitoring function capable of mitigating TCP protocol abuse in the data plane without requiring any modifications to TCP end-hosts or to the protocol. We have demonstrated the effectiveness of our proposal in mitigating two TCP end-host misbehaviors abusing the TCP congestion control mechanism. We believe that our approach has the potential to address other attacks such as ACK division and DupACK spoofing [5]. We conclude that the data plane can be leveraged for such security monitoring at the cost of some additional processing overhead. However, we believe this is a small price to pay for the ability to quickly deploy mitigation solutions to attacks as they are uncovered. This is significantly more scalable and practical than changing protocol implementation on all end-hosts or changing protocol specifications. We have implemented our proposed solution on a software switch target for functional validation. Implementation on a commercial hardware target with further consideration for reducing overhead is one of our future goals. Another future direction we plan to pursue is to adopt the EFSM abstraction for attacks targeting stateful layer-4 and above protocols.

Acknowledgement

This work was partly supported by the FrenchPIA project Lorraine Université d'Excellence, reference ANR-15-IDEX-04-LUE.

References

[1] C. Rossow, "Amplification hell: Revisiting network protocols for ddos abuse," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[2] H. J. Abdelnur, T. Avanesov, M. Rusinowitch, and R. State, "Abusing SIP authentication," in *International Conference on Information Assurance and Security (IAS)*, 2008.

[3] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A Formal Analysis of 5G Authentication," in *ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada: ACM , 2018.

[4] W. Lian, E. Rescorla, H. Shacham, and S. Savage, "Measuring the practical impact of DNSSEC deployment," in *Security Symposium*. USENIX Association, 2013.

[5] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "Tcp congestion control with a misbehaving receiver," *ACM SIGCOMM Comput. Commun. Rev.*, 1999.

[6] R. Sherwood, B. Bhattacharjee, and R. Braud, "Misbehaving tcp receivers can cause internet-wide congestion collapse," in *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[7] K. Ramakrishnan, S. Floyd, and D. Black, "The addition of explicit congestion notification (ecn) to ip," *RFC 3168*, 2001.

[8] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson, "Robust congestion signaling," in *International Conference on Network Protocols (ICNP 2001)*, 2001.

[9] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, "Finding protocol manipulation attacks," in *ACM SIGCOMM Conference*, 2011.

[10] S. Jero, E. Hoque, D. Choffnes, A. Mislove, and C. Nita-Rotaru, "Automated attack discovery in tcp congestion control using a model-guided approach," in *Proceedings of the Applied Networking Research Workshop*. ACM, 2018.

[11] "Fortigate 7000e series ips," Fortinet, accessed: May 29, 2020. [Online]. Available: https://www.fortinet.com/content/dam/fortinet/assets/data-sheets/FortiGate_7000_Series_Bundle.pdf

[12] "Zeek: An open source network security monitoring tool," accessed: May 29, 2020. [Online]. Available: https://zeek.org/

[13] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proceedings of ACM SIGCOMM*, 2013.

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, 2014.

[15] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *Proceedings of NDSS*, 2020.

[16] "Barefoot tofino," accessed: May 29, 2020. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/

[17] A. Laraba, J. François, I. Chrisment, S. R. Chowdhury, and R. Boutaba, "Defeating protocol abuse with p4: Application to explicit congestion notification," in *IFIP Networking 2020*, 2020.

[18] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The p4->netfpga workflow for line-rate packet processing," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019.

[19] *P4 Language Consortium. P4-16 Language Specification*, 2018. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html

[20] *P4 Language Consortium. P4-14 Language Specification*, 2017. [Online]. Available: https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf

[21] V. S. Alagar and K. Periyasamy, *Specification of Software Systems*. Springer Publishing Company, Incorporated, 2011.

[22] Kwang-Ting Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *30th ACM/IEEE Design Automation Conference*, 1993.

[23] K. El-Fakih, N. Yevtushenko, M. Bozga, and S. Bensalem, "Distinguishing extended finite state machine configurations using predicate abstraction," *Journal of Software Engineering Research and Development*, vol. 4, pp. 1–26, 2016.

[24] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of tcp," in *ACM Symposium on SDN Research (SOSR)*, 2017.

[25] T. Mallory and A. Kullberg, "Incremental updating of the internet checksum," Internet Requests for Comments, RFC, 1990.

[26] *P4 Language Consortium. 2018. Behavioral Model (BMv2)*, 2018. [Online]. Available: https://github.com/p4lang/behavioral-model

[27] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. ACM CoNEXT '12.

[28] "p4app," https://github.com/p4lang/p4app, accessed: 2020-01-09.

[29] V. Ramesh, "misbehaving-receiver," 2016. [Online]. Available: https://github.com/rameshvarun/misbehaving-receiver

[30] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of ACM SIGCOMM*, 2010.

[31] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *USENIX NSDI*, 2014, pp. 71–85.

[32] T. P. A. W. Group, "In-band Network Telemetry (INT) data plane specification," June 2020. [Online]. Available: https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf

[33] A. Dhamdhere and C. Dovrolis, "Open issues in router buffer sizing," *SIGCOMM Comput. Commun. Rev.*, p. 87–92, 2006.

[34] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, pp. 397–413, 1993.

[35] H. Wu, J. Ju, G. Lu, C. Guo, Y. Xiong, and Y. Zhang, "Tuning ecn for data center networks," in *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2012.

[36] *Traffic control*, 2019. [Online]. Available: https://github.com/PIFO-TM/ns3-bmv2/tree/master/traffic-control

[37] W. Bai, K. Chen, L. Chen, C. Kim, and H. Wu, "Enabling ecn over generic packet scheduling," ser. CoNEXT '16, 2016, p. 191–204. [Online]. Available: https://doi.org/10.1145/2999572.2999575

[38] B. B. et al, "Recommendations on queue management and congestion avoidance in the internet," *RFC 2309*, 1998.

[39] C.-H. He, B. Y. Chang, S. Chakraborty, C. Chen, and L. C. Wang, "A zero flow entry expiration timeout p4 switch," in *Symposium on SDN Research (SOSR)*. ACM, 2018.

[40] R. Kundel, F. Siegmund, J. Blendin, A. Rizk, and B. Koldehofe, "P4sta: High performance packet timestamping with programmable packet processors," in *Network Operations and Management Symposium (NOMS)*. IEEE, 2020.

[41] R. Bush and R. Elz, "Serial number arithmetic," *RFC 1982*, 1996.

[42] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger, "Tcp extensions for high performance," *RFC 7323*, 2014.

[43] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4fpga: A rapid prototyping framework for p4," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. Association for Computing Machinery.

[44] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[45] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, 2008.

[46] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in sdn-openflow networks," *Computer Networks*, pp. 1–30, 2014.

[47] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "Payless: A low cost network monitoring framework for software defined networks," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, 2014.

[48] Z. Su, T. Wang, Y. Xia, and M. Hamdi, "Cemon: A cost-effective flow monitoring system in software defined networks," *Computer Networks*, vol. 92, pp. 101–115, 2015.

[49] L. Jose, M. Yu, and J. Rexford, "Online measurement of large traffic aggregates on commodity switches," in *Proceedings of USENIX Hot-ICE*, 2011.

[50] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Dream: dynamic resource allocation for software-defined measurement," in *Proceedings of ACM SIGCOMM Conference*, 2014, pp. 419–430.

[51] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, “Openstate: Programming platform-independent stateful openflow applications inside the switch,” *ACM SIGCOMM Comput. Commun. Rev.*, 2014.

[52] C. Sun, J. Bi, H. Chen, H. Hu, Z. Zheng, S. Zhu, and C. Wu, “Sdpa: Toward a stateful data plane in software-defined networking,” *IEEE/ACM Transactions on Networking*, 2017.

[53] J. Sonchack, J. M. Smith, A. J. Aviv, and E. Keller, “Enabling practical software-defined networking security applications with ofx,” in *NDSS*, 2016.

[54] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, “Flow-level state transition as a new switch primitive for sdn,” in *Proceedings of the third workshop on Hot topics in software defined networking*, 2014, pp. 61–66.

[55] P. Chaignon, K. Lazri, J. François, T. Delmas, and O. Festor, “Oko: Extending open vswitch with stateful filters,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’18, 2018.

[56] “A thorough introduction to ebpf,” accessed: May 29, 2020. [Online]. Available: https://lwn.net/Articles/740157/

[57] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, “Hula: Scalable load balancing using programmable data planes,” in *ACM Symposium on SDN Research (SOSR)*, 2016.

[58] V. Bruschi, M. Faltelli, A. Tulumello, S. Pontarelli, F. Quaglia, and G. Bianchi, “Offloading online mapreduce tasks with stateful programmable data planes,” in *Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. IEEE, 2020.

[59] H. Takruri, I. Kettaneh, A. Alquraan, and S. Al-Kiswany, “Flair: Accelerating reads with consistency-aware network routing,” in *USENIX NSDI*, 2020, pp. 723–737.

[60] A. C. Lapolli, J. Adilson Marques, and L. P. Gaspary, “Offloading real-time ddos attack detection to programmable data planes,” in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019.

[61] M. Dimolianis, A. Pavlidis, and V. Maglaris, “A multi-feature ddos detection schema on p4 network hardware,” in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. IEEE, 2020, pp. 1–6.

[62] Y. Li, R. Miao, C. Kim, and M. Yu, “Flowradar: A better netflow for data centers,” in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 2016, pp. 311–324.

[63] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, “Turboflow: Information rich flow record generation on commodity switches,” in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–16.

[64] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, “Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow,” in *Annual Technical Conference (ATC)*. USENIX, 2018.

[65] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, “Elastic sketch: Adaptive and fast network-wide measurements,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 561–575.

[66] Q. Huang, P. P. Lee, and Y. Bao, “Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 576–590.

[67] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, “Heavy-hitter detection entirely in the data plane,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’17. Association for Computing Machinery, 2017.

[68] D. Ding, M. Savi, G. Antichi, and D. Siracusa, “An incrementally-deployable p4-enabled architecture for network-wide heavy-hitter detection,” *IEEE Transactions on Network and Service Management*, no. 1, pp. 75–88, 2020.

[69] J. Kučera, D. A. Popescu, H. Wang, A. Moore, J. Kořenek, and G. Antichi, “Enabling event-triggered data plane monitoring,” in *Proceedings of the Symposium on SDN Research*, 2020, pp. 14–26.

[70] R. B. Basat, X. Chen, G. Einziger, and O. Rottenstreich, “Designing heavy-hitter detection algorithms for programmable switches,” *IEEE/ACM Transactions on Networking*, 2020.

[71] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo, “Burstradar: Practical real-time microburst monitoring for datacenter networks,” in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, 2018, pp. 1–8.

[72] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, and O. Rottenstreich, “Catching the microburst culprits with snappy,” in *Proceedings of the Afternoon Workshop on Self-Driving Networks*, 2018, pp. 22–28.

[73] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, O. Rottenstreich, S. A. Monetti, and T.-Y. Wang, “Fine-grained queue measurement in the data plane,” in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019, pp. 15–29.

[74] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, “Blink: Fast connectivity recovery entirely in the data plane,” in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 161–176.

[75] Z. Xiong and N. Zilberman, “Do switches dream of machine learning? toward in-network classification,” in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, 2019, pp. 25–33.

[76] G. Bianchi, M. Welzl, A. Tulumello, F. Gringoli, G. Belocchi, M. Faltelli, and S. Pontarelli, “Xtra: Towards portable transport layer functions,” *IEEE Transactions on Network and Service Management*, pp. 1507–1521, 2019.

[77] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano, “Flowblaze: Stateful packet processing in hardware,” in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[78] “Netronome agilio cx smartnic,” accessed: May 29, 2020. [Online]. Available: https://www.netronome.com/m/documents/PB_Agilio-CX-OCP.pdf

Abir Laraba received the M.Sc. degree in Networks and Distributed Systems Engineering from the University of Toulouse III Paul Sabatier, France, in 2018. In late 2018, she joined the RESIST Team, a joint team between Inria and the University of Lorraine as a PhD student. His current research interests involve programmable data planes for SDN, Network Security and Monitoring.

Dr. François Jérôme obtained his Ph.D. in Computer Science from the University of Lorraine, France in December 2009. He was then appointed as research associate at the University of Luxembourg. He is now research scientist at Inria in the RESIST team. His main research areas are focused on the use of data analytics techniques for security and also its coupling with network softwarization. In 2019, he received the IEEE Young Professional award in Network and Service Management.

Shihabur Rahman Chowdhury is a PhD candidate at the David R. Cheriton School of Computer Science, University of Waterloo. He received his B.Sc. degree in computer science and engineering from BUET in 2009. His research interests are in virtualization and softwarization of computer networks. He is a co-recipeint of several best paper awards, including, at IEEE/ACM/IFIP CNSM 2019, IEEE NetSoft 2019, and IEEE/ACM/IFIP CNSM 2017 conferences.

Prof. Isabelle Chrisment received the Ph.D. degree in computer science from the University of Nice-Sophia Antipolis, France, in 1996 and the Habilitation degree from Henri Poincare University, Nancy, in 2005. She is a Professor of computer science with TELECOM Nancy Engineering School, University of Lorraine, France. Since 2014, she has been the Scientific Team Leader of the RESIST Team (formerly, MADYNES Team), a joint team between Inria and University of Lorraine. Her main research area is related to network monitoring and security, and especially, within dynamics and large-scale networks.

Raouf Boutaba received the M.Sc. and Ph.D. degrees in computer science from Sorbonne University in 1990 and 1994, respectively. He is currently a University Chair Professor in the David R. Cheriton School of Computer science at the University of Waterloo (Canada) and holder of an INRIA International Chair in France. He is the founding Editor-in-Chief of the IEEE Transactions on Network and Service Management (2007-2010) and the current Editor-in-Chief of the IEEE Journal on Selected Areas in Communications. He is a fellow of the IEEE, the Engineering Institute of Canada, the Canadian Academy of Engineering, and the Royal Society of Canada.