# Defeating Protocol Abuse with P4: Application to Explicit Congestion Notification

Abir Laraba, Jérôme François, Isabelle Chrisment
Université de Lorraine - Inria, France
firstname.lastname@loria.fr

Shihabur Rahman Chowdhury, Raouf Boutaba
University of Waterloo, Canada
{sr2chowdhury|rboutaba}@uwaterloo.ca

***Abstract*—In recent years, programmable data planes enabled by the protocol independent switch architecture (PISA) allowed the relocation of network functions closer to traffic flows and thereby the ability to react in real-time to network events. However, expressing complex and stateful network monitoring functions using state-of-the-art data plane programming languages such as P4 still remain challenging. In this context, we propose a method for modeling a stateful security monitoring function as an Extended Finite State Machine (EFSM) and express the EFSM using P4 language abstractions. We demonstrate the feasibility and benefit of our proposed approach in detecting and mitigating Explicit Congestion Notification (ECN) protocol abuse without any TCP protocol modification. Our evaluation shows that the proposed security monitoring function can restore 24.67% throughput loss caused by misbehaving TCP end-hosts while ensuring fair share of bandwidth among TCP flows.**

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

## I. Introduction

Software-Defined Networking (SDN) paradigm promotes high flexibility in network programming. SDN initial design aims at centralizing network intelligence in the control plane. This approach lacks persistent state in the data plane. Thus, network monitoring capabilities are rather limited and complex monitoring functions require the help of the remote controller [1].

In recent years, significant effort has been dedicated for realizing a more intelligent data plane [2]–[4]. However, the proposed solutions are either hardware specific or remain limited by the originally stateless nature of the data plane specification. To alleviate these limitations, P4 [5] took a clean slate approach for enabling data plane programmability. P4 enables programmers to define their own packet parsing logic, match-action pipeline for packet processing, and packet deparsing logic for a protocol independent programmable switch architecture (PISA) [6]. In contrast to the first generation of SDN, *i.e.,* based on OpenFlow [7], P4 introduces the necessary abstraction for stateful processing in the dataplane. Stateful dataplane programming enabled by P4 has led to many applications such as traffic measurement [8], [9], load balancing [10], [11] and security enforcement [12]. However, the P4 language and PISA architecture sacrifice some flexibility in terms of supported operations to ensure packet processing programs can execute at line-rate. These limitations have spurred many workarounds and architecture design extensions in the literature [13]. A notable example is P4CEP [14] proposed to ease the use of P4 by describing complex event processing functions using regular Finite-State Machines. In the same vein, we propose to use a general abstraction based on Extended Finite-State Machine (EFSM) for implementing a security monitoring function. We also present a technique to map it to a P4 based data plane. EFSMs achieve scalability by using variables and actions in addition to state transitions in a finite-state machine, in this way enabling complex operations while avoiding state-space explosion.

In order to demonstrate the effectiveness of our security monitoring function in the dataplane, we focus on detecting and reacting to Explicit Congestion Notification (ECN) protocol [15] abuse. ECN lies between IP and TCP layers and is used by the network switches to indicate possible congestion based on switch queue occupancy. A misbehaving TCP end-host can choose not to echo a set ECN bit back to the sender, giving the sender the illusion that there is no possibility of congestion in the network. Because of this misinformation many TCP based applications may suffer from this type of attack [16]–[20]. Furthermore, we also experimentally demonstrate in Section IV that an attacker can obtain two-third share of the available bandwidth through such an attack.

By nature, TCP is an end-to-end protocol, which limits the capability of monitoring and reaction by intermediate nodes such as switches and routers. In [16], the authors propose to change the TCP specification to defend against TCP protocol abuse, which is very difficult from a practical standpoint. In contrast, our in-network ECN abuse detection and reaction approach neither requires change to the TCP protocol nor to the end-hosts. To the best of our knowledge, we are the first to propose an approach to mitigate ECN abuses directly within the network without modifying TCP.

Specifically, we make the following contributions:

- an EFSM abstraction to monitor and react to any protocol behavior. In particular, misbehaviors can be tracked in the data plane thanks to a method that maps an EFSM model to P4;
- an EFSM-based model to detect misbehaving hosts abusing ECN protocol;
- an implementation of our EFSM model with P4 following our proposed mapping method and its evaluation compared to a baseline scenario without ECN protocol

ISBN 978-3-903176-28 

abuse monitoring. The evaluation shows the benefit of our approach in ensuring a fair bandwidth share between TCP flows even in the presence of the attacker.

The remainder of this paper is organized as follows. We introduce the necessary background and details of our EFSM abstraction and its mapping to P4 in Section II. Then, we describe ECN protocol abuse and the EFSM model to detect that misbehavior in Section III. We report our findings from the experimental evaluation in Section IV followed by a summary of related works in Section V. Finally, we conclude the paper with some future research directions in Section VI.

## II. Proposed Approach

### A. Overview

Our approach for detecting and mitigating protocol misbehavior in the data plane is outlined in Figure 1. Our objective is to monitor network flows in real-time to identify protocol abuse. The first step in our approach 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 modelled 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. These first stages of abstracting the model of misbehaviors (*i.e.*, protocol abuses) using EFSM are done manually based on available documentation. 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 II-D.

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 and all the associated variables of the EFSM. When the EFSM model execution enters a state labelled as misbehavior, different actions can be triggered by the program but need to be defined a priori by the network operator such as dropping the packet, rerouting the packet, generating an alert, applying corrective actions, *etc*.

The proposed approach detects and mitigates protocol abuse in real-time without involving any distant controller. In the following sections, we present some background about the P4 data plane language, the state machine model and the process used to map an EFSM to the data plane primitives in detail.

### B. P4 data plane programming

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

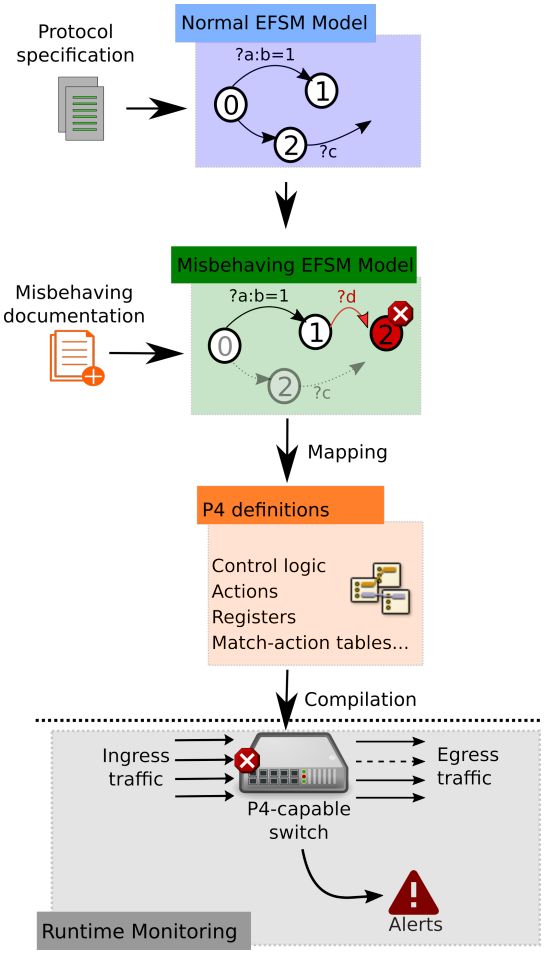


Fig. 1. Approach overview

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 behavior on a large variety of targets, such as NetFPGA [21].

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 [22]. 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: consists of the table entries and the possible actions, table entries are typically modified by the control pipeline.
- Registers: stateful memories that maintain state across

packets. P4 actions can read or modify registers 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 [23]. 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.

### C. EFSM model

We adopt an EFSM model for modeling a target protocol. While there are some variations in the EFSM formalism [24], [25], in this paper we represent an EFSM by a 7-tuple $(S, E, A, I, V, C, T)$ 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 misbehaviors.
- $E$ is a finite set of events. An event is triggered by a particular content in the monitored packets such as the presence of a TCP flag in a TCP packet.
- $I$ is the set of initial states. In this paper, $I$ represents those states of the protocol from where it is necessary to monitor subsequent states to detect misbehaviors.
- $V$ is the set of state variables that are accessible to every state. It is meant to read and write persistent information in an EFSM in addition to the states themselves. This is one of the main advantages of EFSM to avoid creating numerous states for variables. For example, tracking sequence numbers with a regular FSM would require one state for each of the $2^{32}$ possible values.
- $A$ is a finite set of actions to be performed. For example, sending an alert or updating a variable $v \in V$.
- $C$ is the set of conditions. Each condition $c \in C$ is composed of a combination of input events from $E$ and numerical or logical condition on variables from $V$. This is another major advantage compared to regular FSM where transitions are defined on symbols only. With EFSM, more complex conditions can be described.
- T is the set of transitions, where $t \in T$ is defined as $s_1, c \rightarrow a, s_2$ ($s_1 \in S$, $s_2 \in S$, $c \in C$, and $a \in A$). It denotes a transition from state $s_1$ to $s_2$ when condition $c$ is satisfied and results in the action $a$. For instance, when a misbheavior is detected, the transition from a *normal* state to a *misbehavior* state is triggered based on a particular condition (*e.g.*, malformed packet). Besides, to change the system state, we can also send an alert to the administrator or correct the malformed packet as an action.

### D. Mapping an EFSM model to P4 Program

Once a protocol specification is modeled as an EFSM for tracking misbehaviors, the model is mapped to a P4 program and the latter compiled to be executed on the data plane. Therefore, the 7-tuple $(S, E, A, I, V, C, T)$ has to be mapped to the P4 abstraction.

The P4 core primitives mentioned previously (Section II-B) allow us to implement the EFSM in the data plane by maintaining the EFSM states in the switch registers. To monitor network connections behavior in real-time, the data plane tracks and processes the different states and transitions.

*1) Maintaining current states:* An instance of the EFSM model is maintained for each monitored connection or flow. A flow is formally defined as a set of packets $p_i$: $f = \{p_0, p_1, \ldots, p_n\}$ sharing the same set of attributes. A unidirectional TCP/UDP flow is represented as $(srcIp, dstIp, srcPort, dstPort, protocol)$, 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. While this is a regular definition of a flow, the format may be freely defined based on the packet headers extracted by the P4 parser. However, in some cases, packets grouped into flows do not match exactly the same selected headers but share some properties. A notable example is bidirectional TCP flows with inverted IP addresses and TCP ports. Grouping packets of a bidirectional TCP flow is needed to monitor the 3-way handshake initiation. One solution to this problem proposed in the literature is If (dstIP $>$ srcIP), hash the former order (dstIps,srcIp, dstPort, srcPort, protocol) otherwise hash the reverse order (srcIp, dstIp, srcPort, dstPort, protocol) [8]. Thanks to this comparison the value of the IP address in the 5-tuple always remains the same whether it is the source or destination address. The same applies to the TCP ports.

Once flow keys are extracted from a packet, we compute a hash function to associate a packet to a flow. The current EFSM state of each flow is mapped to an entry in a register array $R = \{r0, r1, \ldots, rn\}$, where $n$ is the maximum number of flows to be monitored. These values are accessed and modified by three specific actions:

- *state-lookup* to query the state register corresponding to the hash of a flow 5-tuple and save it in a metadata.
- *state-update* an action to update the state metadata.
- *state-write* to write back the new updated state (metadata) in the register.

These functions are leveraged in the transition system defined in Section II-D3.

*2) Events detection $E$ and mapping variables from $V$:* Events in $E$ are derived directly from the ingress packets. Conditional tests are performed on matching header fields on particular values. Depending on the detected event, variables from $V$ are updated. We distinguish two types of variables according to their nature leading to different representations in P4. First, some variables are persistent across the states of the system and are by definition the EFSM variables, i.e. belonging to $V$. For example, when tracking a TCP connection sequence numbers, we need to maintain the last seen sequence number. They are per-flow (*i.e.*, per EFSM-instance) variables. For this type of variables, we use registers to store and modify the values of the variables during state

transitions. The second type of variables concerns variables which are used only in conditions in $C$ and actions in $A$ of a single transition. As transitions are triggered by event and conditions, this second type of variables is qualified as per-packet metadata. Metadata based variables are associated with the processing of each packet and used temporarily to carry packet related information between the pipeline stages. They are deleted when the processing is completed. For example, they can be used to carry the result of a hash, result of an arithmatic operation, or the value read from a register. For example, the Time-To-Live (TTL) value of a forwarded packet in an L3-switch is solely based on its initial value that is decreased by one. If the forwarding is modelled as a single transition in an EFSM model, the TTL variable does not need to be persistent across states.

*3) Mapping of the transition system:* The transition system expresses the different transitions ($T$) and their components: actions ($A$), conditions ($C$), events ($E$) and variables ($V$). Except for events derived from ingress packets and variables defined a priori, actions, transitions and conditions are implicitly defined within P4 actions. Indeed, the current state of a given EFSM evolves according to conditions on events and variables defined in the ingress control of the P4 processing pipeline. It is important to note the difference between EFSM actions and P4 actions. P4 actions are functional code to perform various operations including conditional tests (modelled as conditions in EFSM). More precisely, each $c \in C$ is a conditional expression composed of operands to express logical or numerical calculations (*e.g.*, addition, AND, OR, *etc.*) and comparisons (*e.g.*, less-than, equal, *etc.*). The set of operations are performed on the stored variables or on the parsed packets header fields. We denote the list of actions with $A = \{a_0, a_1, ..., a_n\}$. Actions are split into two categories. The first category of actions ($A1 \in A = \{forward, drop...\}$) determine packet forwarding behavior. The second category $A2$ represents arithmetic and logic operations to update the variables in $V$ (in P4 metadata or registers) or packet headers. Moreover, to maintain the instance of the EFSM up-to-date, the defined P4 actions rely on particular state maintenance functions described before, *i.e.*, $\{state - lookup, state - update, state - write\}$. They are used to check and update the current state according to the defined transitions and incoming packets. Therefore, from a P4 perspective, EFSM states ($S$) and other variables ($V$) stored in registers are handled similarly.

*4) Hashing and hardware constraints:* The limited hardware resources pose a challenge when offloading some processing to the data plane as highlighted in [11] [8], especially to track per-flow state. Similar to these works, we address this challenge by maintaining the hash of 5-tuple of each flow rather than the 5-tuple to reduce the width of flow keys in hardware. To handle hash collisions a *hash-chaining* method similar to [8], [11], which maintains a linked set of values with different keys in the same hash table index can be used.

P4 is designed to support a wide-range of hardware targets. However, one possible question is how the hardware could support the deployment of the P4 externs such as registers that are necessary to maintain the persistent state. Although P4 externs are target-specific, registers are widely-used and expected to be supported by most hardware or software target platforms. For example, in [26], authors propose a P4 compiler for various FPGA hardware including stateful objects.

## III. Application to ECN

In this section, we demonstrate the viability of our proposed approach to detect and mitigate misbehaving end-hosts abusing the ECN protocol.

### A. ECN background

TCP end-hosts typically use end-to-end congestion signals such as packet loss or round-trip-time for adjusting their congestion windows. ECN was proposed as a mechanism for network devices to send congestion signals to end-hosts [27]. Accordingly, a switch experiencing congestion explicitly notifies an end-host to reduce its congestion window. The ECN RFC [15] 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 2. 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 2);
- 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 2);
- 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 2).

ECN RFC defines a possible misbehavior of an end-host announcing itself as ECN-capable but ignoring congestion notification from the switch [15]. As shown in Figure 2, 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 2). As a result, the sender does not reduce the congestion window ((3') in Figure 2). Another possibility is when the receiver keeps sending packets with ECE set ((4') in Figure 2) even after the sender has already reduced its congestion window.

Both of these misbehaviors are represented in Figure 2. In addition, the sender can ignore the notification sent back from the receiver ((2) in Figure 2) by not reducing the congestion window ((3') in Figure 2).

Misbehaving flows can severely degrade network performance. This is why the RFC recommends that such flows must be identified and handled. We do so in this paper by leveraging data plane programmability to embed in-network ECN misbehavior detection and mitigation.

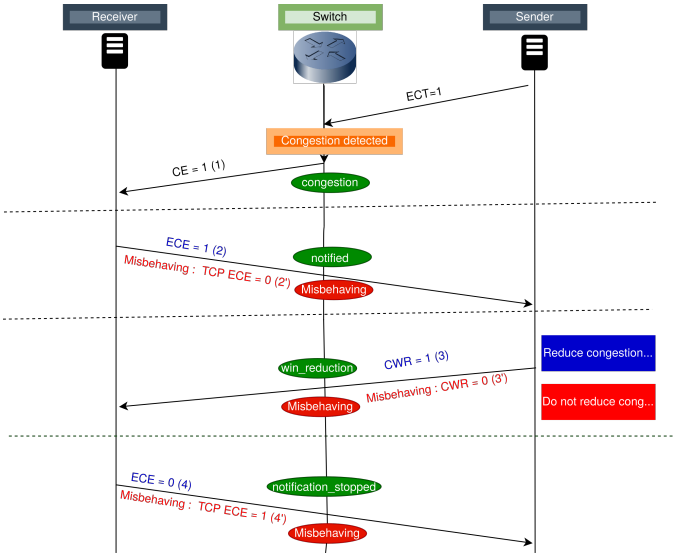


Fig. 2. ECN (normal behavior in blue, misbehavior in red, ellipses represent EFSM state updates)

### B. EFSM model of ECN

We model both the expected and possible misbehavior of ECN capable end-hosts in a single EFSM. First, we define the states of the EFSM. Each time a packet is received, the state of the EFSM illustrated by an ellipse in Figure 2 is updated. Normal states are colored in green while all misbehavior states are merged into a single red colored one. Accordingly, we have the following set of states, $S = \{congestion, notified, win\_reduction, notification\_stopped, misbehaving, init\}$. Here, $init$ is a dummy initial state, $I = \{init\}$. The EFSM model is instantiated into the switch and the transitions are triggered by events which are directly derived from ingress packets as explained in section II-D2. State transition events are defined based on the ECN related flags in the packets as $E = \{ECT = 1, ECE = 1, ECE = 0, CWR = 1, CWR = 0\}$. State transition occurs from $init$ to $congestion$ at the following condition: ECT AND $congestion = 1$ where $congestion \in V$ is an intrinsic metadata defined by the switch architecture indicating the occurrence of the congestion.

Based on these definitions, we illustrate the EFSM corresponding to Figure 2 in Figure 3. For transition between two states, the condition and actions correspond to the labels on the arrows. When a congestion is detected, the switch sets the CE flag to 1 and forwards the packet, *i.e.*, takes two actions: $forward \in A$ and $(CE \leftarrow 1) \in A$. When the receiver acts according to the ECN specification, the switch should receive back a message with ECE set, trigerring the event $ECE = 1 \in E$. At this stage, the EFSM state is updated to $notified$ and the emitter (or sender) reduces its congestion window. The event $CWR = 1$ should then be observed and the EFSM should transition to $win\_reduction$ state. The EFSM is then updated until reaching the final state $notification\_stopped$.

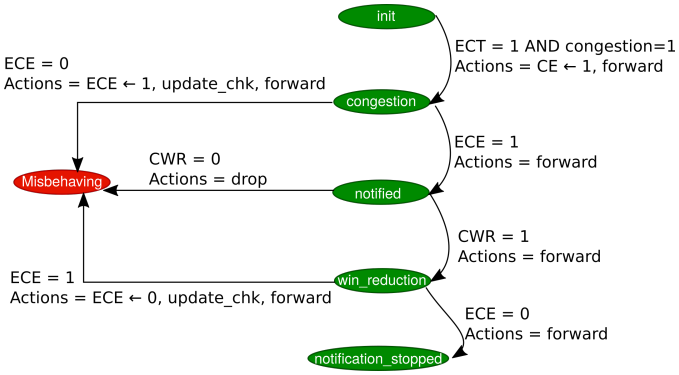


Fig. 3. ECN state machine with misbehavior detection and correction

In Figure 2, there are three possible events that can trigger a transition to the $misbehaving$ state:

1) When the receiver voluntarily ignores the explicit congestion signal from the switch. In that case, the current state of the EFSM is $congestion$ and the event is $ECE = 0$, *i.e.* the receiver does not forward the congestion notification.
2) When the congestion notification has been echoed back to the sender, *i.e.*, state $notified$, but the sender does not reduce its congestion window as observed by the event $CWR = 0$;
3) When the receiver does not stop notifying the congestion (event $ECE = 1$) whereas the sender has already reduced its congestion window (state $win\_reduction$).

Once a misbehavior is detected the EFSM remains in its state. When a TCP connection is terminated, the associated EFSM is not updated anymore. Although it is out of the scope of this paper, detecting the end of a TCP connection may rely on different mechanisms such as through FIN and RST flags.

When a misbehavior is detected, the packet can be simply dropped. However, we also introduce corrective actions in our approach when possible. The switch cannot effectively verify that the sender has reduced its congestion window. As a result, it is not possible to deduce from the switch if the CWR flag must be really set to one in the second misbehaving case, hence, we drop the packet. However, for the first and third cases, the receiver must react according to the congestion signal (with either $CE = 1$ or $CWR = 1$). Therefore, the misbehavior can be corrected by setting or unsetting $ECE$ for the first and third misbehaving cases, respectively, *i.e.* $(ECE \leftarrow 1)$ and $(ECE \leftarrow 0)$ are in the action list $A$.

Since $ECE$ is a TCP flag, altering its value requires updating the checksum. Alternatively, without corrective actions, the packet can be always dropped when a transition to the *misbheaving* state occurs. Therefore, the full list of actions seen in Figure 3 is $A = \{ECE \leftarrow 1, ECE \leftarrow 0, update\_chk, CE \leftarrow 1, drop\}$.

### C. Mapping ECN EFSM to P4

The EFSM model of the ECN protocol is mapped to a P4 program according to the method described in section II-C. We store the current state of a connection in a register and all other variables in P4 metadata since they do not need to persist across packets. Variables other than the current state are as follows:

- $congestion$, as already introduced, indicates if the switch is experiencing congestion. This variable is derived from the occupancy level of the switch queues.
- intermediate variables to process the different headers fields, flags and checksum.

We perform $update\_chk$ action in the egress pipeline. However, instead of performing a costly full calculation of the checksum, we use an incremental procedure [28]. The checksum is incrementally updated using a simple arithmetic function without recalculating the checksum for fields of the packet that have not changed as follows:

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

Where :

- $HC$: old checksum in the TCP header,
- $HC'$: new checksum in the TCP header,
- $m$: old value of a 16-bit field including the former TCP value field ($ECE$ or $CWR$ in our context),
- $m'$: new value of a 16-bit field including the modified TCP value field ($ECE$ or $CWR$ in our context), $\sim x$ : the one's complement of $x$

## IV. Evaluation

All experiments were performed using P4-Bmv2 software switch [29] with mininet [30] and p4app [31]. The P4 programs were implemented using the P4 specification version P4-16 [22]. Our evaluation aims at assessing if ECN misbehaviors are appropriately detected and mitigated, and evaluating the overhead introduced by our P4 implementation. Mininet does not allow us to assess performance in realistic conditions in terms of absolute values of the achieved performance. Therefore, a baseline scenario for comparison purposes is used in the different experiments.

*1) Bandwidth share and throughput evaluation:* To evaluate the impact of the use or misuse of ECN, we measure the throughput of TCP flows to assess if the bandwidth is correctly shared even during congestion. We rely on the topology presented in Figure 4. It is composed of one switch and four hosts (h1, h2, h3, h4). By default, all hosts are ECN-capable and the link latency is set to 1ms. To demonstrate the impact of a misbehaving host during network congestion, we limit the output queue rate (by default to 2000 packets/sec). Previous work on ECN recommends assuming congestion once 65 packets are queued in the switch in case of a 10 Gbits/s network [32]. With our setup using mininet, the maximal throughput observed is around 40 Mbits/s, therefore a 3 packets threshold in the queue is considered as congestion and so will trigger the event $congestion$ to leave the $init$ state of the EFSM. We simulate a misbehaving TCP receiver at h1.

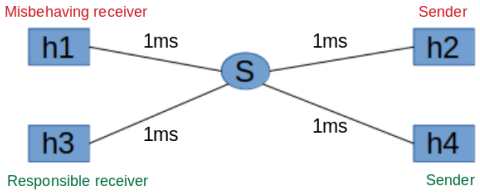


Fig. 4. Experimental topology used in mininet

We generate TCP flows using iperf between h1 and h2, and from h3 and h4 to h1 and h3, respectively (h1 and h3 are iperf servers). We tweaked h1 to act as a misbehaving host by not echoing congestion information ((2') in Figure 2), *i.e.* not setting the TCP ECE flag. h1-h2 and h3-h4 flows are thus qualified as misbehaving and normal, respectively.

In Figure 5, we present the throughput of the normal flow (h3-h4) and the misbehaving flow (h1-h2) 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)
- 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 misbehavior detection and applies corrective measures limited to the first type of misbehavior. The switch implements partially the EFSM of Figure 3 with $S = \{init, congestion, misbehaving\}$.
- Misbehaving-fullReaction: the switch implements the full EFSM we proposed in Figure 3 while h1 continues to be a misbehaving host.

In Figure 5, the boxplots represent 10 simulation runs of 60 seconds each. In the case of no misbehaving hosts (noMisbehaving-forward), a fair bandwidth share is observed, each flow getting around 21 Mbps. In contrast, the misbehaving flow h1-h2 consumes most of the available bandwidth at the expense of the flow h3-h4 when our switch is not programmed to detect and react to misbehaviors (Misbehaving-noReaction). More precisely, the misbehaving flow reaches an average of 22 Mbps while there is only 11.3 Mbps left for the normal flow (66% vs. 34% share). In fact, the normal flow (h3-h4) responds to the congestion signal and reduces its transmission rate, leaving more queue space to the misbehaving flow which gains more share of the bandwidth. However, effective throughput is still limited by the sender

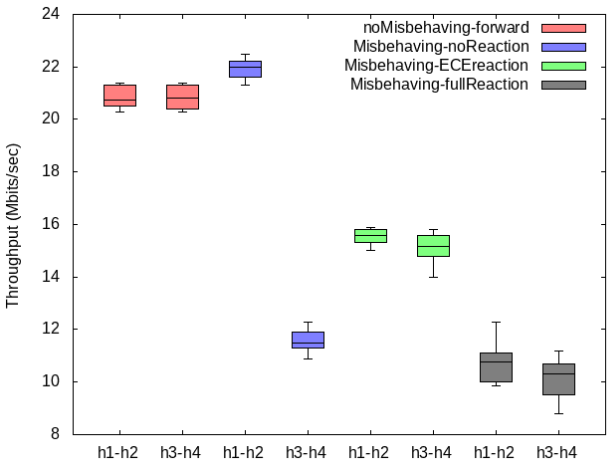


Fig. 5. Bandwidth share for a queue rate of 2000 packets/sec

capacity and thus the increase of the misbehaving flow is lower than the loss of the responsive flow throughput.

When we turn on misbehavior detection and mitigation in the switch (Misbehaving-ECEreaction), Figure 5 shows that our solution properly redistributes the bandwidth and ensures almost equal share between the two flows. In fact, the normal flow regains 24.67% of the throughput loss caused by the misbehaving flow. However, there is a throughput degradation compared to the baseline scenario due to the verification and mitigation process performed by the P4 program. Similarly, when the full EFSM is instantiated (Misbehaving-fullReaction), an equal share is also observed, however, the performance is degraded around 10 Mbits/s per flow. The normal flow's throughput becomes even lower than that in the Misbehaving-noReaction case. Therefore, we recommend a partial implementation of the EFSM against misbehaving ECN hosts limited to when the congestion information is not echoed back ($ECE = 0$). It saves switch resources and avoids wasteful reaction to a bad CWR flag. As can be seen from Figure 3 and described in Section III-B, it is impossible for the switch to apply an appropriate correction for the latter case (only the TCP end-host can effectively reduce its congestion window).

In Figure 6, the normal flow's throughput is measured while varying the output queue rate. For low values below 500 packets/second, the throughput is independent of the type of scenario, *i.e.*, with or without reaction, with or without misbehaving flows. This is because the queue rate is too low to allow the misbehaving flow to really gain some share of the bandwidth. Then, the bandwidth share rapidly becomes imbalanced between flows when the switch only forwards packets without detection and correction. When the recommended corrective actions are applied, regain in throughput is possible for up to a certain queue rate. After this threshold, the queue rate allows naturally to reduce congestion without additional overhead in contrast with our solution, so there is little benefit of our approach. Consequently, before deploying such solution, its adequacy to network conditions must be verified.

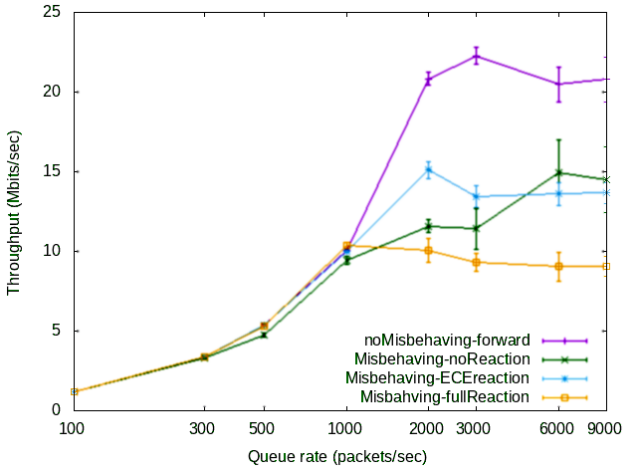


Fig. 6. Responsive h3-h4 flow throughput depending on the output queue rate

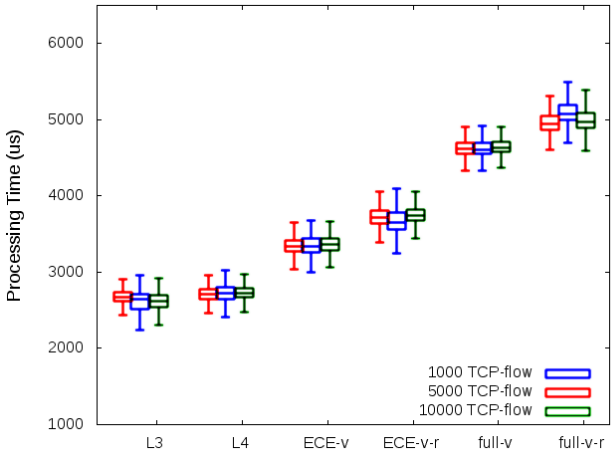


Fig. 7. Switch processing time

*2) Switch processing time evaluation:* In this experiment, our goal is to evaluate the runtime overhead per packet. We consider different switch configurations for this experiment:

- 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 3.
- ECE-v-r: the state machine is partially implemented to monitor and react to the first misbehavior type, ((2') in figure 2); $S$ is restricted to $\{init, congestion, notified, misbehaving\}$.
- full-v: the full state machine is implemented except for the corrective actions and checksum recalculation.
- full-v-r: similar to full-v but including the reaction against the first-type of misbehavior ((2') in Figure 2) by setting

ECE to 1 and including checksum recalculation.

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 7. 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, which is consistent with the results in Figure 5. It is worth noting 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. Finally, as can be seen from Figure 7, our approach is scalable with respect to the number of flows to monitor in parallel.

### A. Discussion on memory overhead

The implementation of our solution on programmable hardware raises the issue of memory constraint. In our approach, the P4 program needs to keep one register entry for each flow's current state, which will require a register array large enough to track all the active TCP connections. Metadata fields that carry information between tables consume a negligible amount of bits [6]. Each entry in the state register has 32 bits for the flow key hash and 32 bits for the value (current state). The state registers can be implemented with TCAM or hash-based memory. State-of-the-art programmable hardware provide sufficiently large memory to accommodate state registers capable of holding 10s of thousands of active TCP connections. For example, P4FPGA [26] can implement up to a 288 bits key for TCAM or hash-based memory and can fit up to 93K entries.

## V. Related works

SDN has brought new capabilities for network monitoring. Initially, control plane-based monitoring enabled by OpenFlow [7] was proposed where switches report monitoring information such as counters to a remote controller. In this paradigm, the controller directly controls the monitoring frequency and the granularity of flow rules to monitor. This created the opportunity for dynamically controlling such parameters based on monitoring needs [33]. For instance, the work in [34] proposes to adapt reporting frequency based on the variability of past observations. Another area explored under SDN monitoring is the use of sketches to capture more sophisticated information from network traffic beyond the already built-in counters [35]. One common trait in all of these solutions is that they are stateless and allow to retrieve general statistics about flows but do not allow to track more complex protocol behaviors. To do so, it is possible to mirror packets to middleboxes that can employ further processing to reconstruct the protocol state machines [36].

In contrast to the above stateless solutions, we propose in-network monitoring of protocol state-machines. There has been some research in the direction of extending OpenFlow with stateful processing capabilities. For example, OFX [3] extends OpenFlow with an external agent running on the switch with stateful monitoring capabilities. Oko [37] proposes to extend OpenFlow capabilities with extended Berkley Packet Filter (eBPF) for stateful processing [37]. However, Oko is exclusively limited to Linux-based software switches. In contrast, our solution can be deployed in any data plane device along the packet processing pipeline.

The recently emerging PISA architecture and the accompanying P4 programming language has inspired many research works in stateful dataplane processing. For instance, stateful load balancing in the data plane has been proposed and implemented using P4 [11]. TCP connection diagnostic in the data plane has been proposed in Dapper [8]. Authors in [9] propose a solution to track queuing delays and react in data plane by rerouting flows. A typical use case explored using data plane programming is DDoS detection such as those presented in [38] and [39]. Authors in [40] made the first step toward mitigating attack targeted to switch programs. They proposed mechanisms to analyze the expected behavior of a P4 program and discover malicious traffic patterns that would cause it to misbehave. Recently, some research effort has been dedicated to performing complex calculations in the data plane such as logarithms, which are not supported by default [41].

To the best of our knowledge, we are the first to address the ECN misbehaving problem without requiring a TCP protocol modification [16]. Furthermore, we provide a general abstraction based on an EFSM and its mapping to P4. A closely related work is OpenState [2], however, it is limited to regular FSM and variables representing stateful information cannot be stored. SDPA [42] is also an OpenFlow extension to support stateful packet processing in the data plane. The authors introduce a new component to manage the state machine at the switch. While they propose a hardware-specific implementation (on FPGA), our solution relies on P4 for supporting different hardware platforms. A hybrid system coupling the data plane and the control plane to maintain a state machine has been proposed in [4].

The abstraction proposed in this paper shares the basic principals with FlowBlaze [43] by modelling EFSM in the data-plane. However, FlowBlaze proposes to extend the RMT models with dedicated tables for EFSM and a language similar to P4 for utilizing those tables. Therefore, FlowBlaze can be considered as a parallel endeavour to P4. In contrast, we rely on P4 for its generality and its higher adoption in commercial switches.

## VI. Conclusion

In this paper, we proposed an EFSM abstraction for monitoring protocol abuse and reacting appropriately when such abuse is identified. We also proposed a method for mapping an EFSM model representing a protocol state-machine to network data plane primitives using P4. We demonstrated the feasibility of our proposal through the ECN protocol use case in which we guarantee an equal bandwidth share even in the presence of misbehaving end-hosts. We conclude that an EFSM model can be partially implemented to mitigate selected misbehaviors without incurring overhead. As future work, we plan to further

explore the attack space and end-host misbehaviors targeting other protocols.

## ACKNOWLEDGEMENT

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

## REFERENCES

[1] N. Feamster and J. Rexford, "Why (and how) networks should run themselves," in *Proceedings of the Applied Networking Research Workshop*, ser. ANRW '18, 2018.

[2] 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.

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

[4] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for sdn," in *ACM SIGCOMM*, 2014.

[5] 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.

[6] 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.

[7] 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.

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

[9] B. Turkovic, F. Kuipers, N. van Adrichem, and K. Langendoen, "Fast network congestion detection and avoidance using p4," in *ACM Workshop on Networking for Emerging Applications and Technologies (NETA)*, 2018.

[10] 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.

[11] 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.

[12] 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.

[13] R. Bifulco and G. Rétvári, "A survey on the programmable data plane: Abstractions, architectures, and open problems," in *IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018.

[14] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, and K. Rothermel, "P4CEP: towards in-network complex event processing," in *Workshop on In-Network Computing NetCompute@SIGCOMM 2018*. ACM, 2018.

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

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

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

[18] 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.

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

[20] 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.

[21] 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*, ser. FPGA '19, 2019.

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

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

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

[25] 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.

[26] 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, 2017.

[27] 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.

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

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

[30] 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, 2012.

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

[32] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ecn in multi-service multi-queue data centers," in *Proceedings of USENIX NSDI*, 2016.

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

[34] 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.

[35] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[36] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and et al., "Packet-level telemetry in large datacenter networks," in *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[37] 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.

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

[39] 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.

[40] Q. Kang, J. Xing, and A. Chen, "Automated attack discovery in data plane systems," in *USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, 2019.

[41] D. Ding, M. Savi, and D. Siracusa, "Estimating logarithmic and exponential functions to track network traffic entropy in p4," in *Proceedings of IEEE/IFIP NOMS*, 2020.

[42] 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.

[43] 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.