This is a preprint copy that has been accepted for publication in IEEE Communications Surveys & Tutorials. Please cite this article as:
T. Dargahi; A. Caponi; M. Ambrosin; G. Bianchi; M. Conti, "A Survey on the Security of Stateful SDN Data Planes," in IEEE Communications Surveys & Tutorials , vol.PP, no.99, pp.1-1
doi: 10.1109/COMST.2017.2689819
Note: This preprint copy is only for personal use.

A Survey on the Security of Stateful SDN Data Planes

Tooska Dargahi, Alberto Caponi, Moreno Ambrosin, *Student Member, IEEE*
Giuseppe Bianchi, and Mauro Conti, *Senior Member, IEEE*

***Abstract*—Software-Defined Networking (SDN) emerged as an attempt to introduce network innovations faster, and to radically simplify and automate the management of large networks. SDN traditionally leverages OpenFlow as device-level abstraction. Since OpenFlow permits the programmer to "just" abstract a static flow-table, any stateful control and processing intelligence is necessarily delegated to the network controller. Motivated by the latency and signaling overhead that comes along with such a *two-tiered* SDN programming model, in the last couple of years several works have proposed innovative switch-level (data plane) programming abstractions capable to deploy *some* smartness directly inside the network switches, e.g., in the form of localized stateful flow processing. Furthermore, the possible inclusion of states and state maintenance primitives inside the switches is currently being debated in the OpenFlow standardization community itself. In this paper, after having provided the reader with a background on such emerging stateful SDN data plane proposals, we focus our attention on the security implications that data plane programmability brings about. Also via the identification of potential attack scenarios, we specifically highlight possible vulnerabilities specific to stateful in-switch processing (including denial of service and saturation attacks), which we believe should be carefully taken into consideration in the ongoing design of current and future proposals for stateful SDN data planes.**

***Index Terms*—Software-Defined Networking (SDN), Stateful SDN Data Planes, Data Plane Programmability, SDN Security, Vulnerability Assessment, OpenFlow, OpenState, P4.**

I. Introduction

Software-Defined Networking (SDN) is an emerging networking paradigm in which the control and management of the network is separated from the traffic forwarding primitives. In SDN, a (logically) centralized *control plane* monitors the whole network and makes decisions on packet forwarding for the switches (i.e., *data plane*) inside the network. In existing SDN frameworks, the main interface to the network switches is *OpenFlow* [1]. OpenFlow provides network administrators with a simple and uniform abstraction to the configuration of different (physical or virtual) multi-vendor's network devices. Specifically, an *SDN controller* inserts and updates forwarding rules for the current *traffic flows* into one or more *flow tables* inside each switch. The resulting decoupling of the control and data plane simplifies network monitoring, fault tolerance, and security policy enforcement, albeit it introduces new security issues that are comprehensively discussed in the literature [2]–[7].

OpenFlow was originally designed with the desire for rapid adoption, opposed to first principles [8], i.e., it emerged as a pragmatic attempt to address the dichotomy between i) flexibility and ability to support a broad range of innovation, and ii) compatibility with commodity hardware and vendors' need for closed platforms [1]. The aftermath is that most of the proposed network programming frameworks [9]–[15] circumvent OpenFlow's limitations by promoting a "two-tiered" [16] programming model: *any* stateful processing intelligence of the network applications is delegated to the network controller, whereas OpenFlow switches limit to install and enforce stateless packet forwarding rules delivered by the remote controller. Centralization of the network applications' intelligence is an advantage whenever changes in the forwarding states do not have strict real time requirements and depend upon global network states. However, it may become a show stopper for applications which rely only on local flow/port states and which need to react at the *packet-level time scale*: as the latency toll imposed by the reliance on an external controller rules away the possibility to enforce software-implemented control plane tasks at wire-speed, i.e., while remaining on the fast path[1].

A. The rise of stateful SDN data planes

In the last couple of years, a new wave of research started to consider the possibility to *offload* some *stateful* traffic processing and control tasks directly inside the network switches, so as to reduce the switch-to-controller's signaling overhead and the latency shortcomings induced by the above discussed "two-tiered" programming model.

The first proposals explicitly considering the handling of *flow states* as a switch's primitive were OpenState [18] and FAST [19]. Their rationale was to permit the programmer not only to formally describe (and hence programmatically configure inside the switch) *stateless* flow tables, but also

T. Dargahi, A. Caponi and G. Bianchi are with CNIT and the Department of Electronic Engineering at the University of Rome Tor Vergata, Rome, Italy (e-mail: name.surname@uniroma2.it).

M. Ambrosin and M. Conti are with the Department of Mathematics, University of Padua, Padua, Italy (e-mail: surname@math.unipd.it).

[1]A 64 bytes packet takes about 5 ns on a 100 Gbps speed, roughly the time needed for a signal to reach a control entity placed one meter away. And the execution of an albeit simple software-implemented control task may take way more time than this. Thus, even the physical, capillary, distribution [4], [17] of control agents (as proxies of the remote SDN controller for low latency tasks) on each network device would hardly meet fast path requirements.

stateful rules, i.e., forwarding rules which dynamically change on the basis of a (programmatically defined!) *state* of a flow, and which further determine when (and how) state transitions should occur.

As surveyed in Section II-C, many other proposals emerged since then, focusing on either innovative hardware architectures [20]–[22] for stateful in-switch processing, or on higher level programming languages and compilers which could exploit such emerging programmable data planes [16]. The need to include stateful processing in the data plane was also more recently promoted by the P4 [23], [24] data plane programming language community[2]. P4 language specification [25] indeed goes in this direction by providing the programmer with an array of *registries* which can store states that persist across multiple packets of a flow (i.e., flow states). And, last but not least, at the time of writing, there is a serious debate in the Open Networking Foundation (ONF), the forum which standardizes OpenFlow, on proposals which foster the explicit inclusion of states and state tables in the next release of the OpenFlow standard, namely version 1.6.

At this stage, the layman reader might perhaps see the *distribution of states inside network switches* as a step backward with respect to the SDN principles, and, perhaps for this reason, proposals such as [26] take a more conservative (hybrid) approach where switches keep the state of flows under supervision of the controller, but state management inside the switch is still performed by the controller. However, we believe that also the most aggressive stateful SDN data plane proposals presented later on *do not violate* the SDN principle of separating control from data plane forwarding, as they still retain a centralized control of (i) which states are offloaded, for performance reasons, to the switch, and (ii) how states shall be managed. In fact, the *decision* of how a switch should react to packet-level events and update the forwarding policies is centrally taken, and formalized into *data plane programs* (e.g., a P4 code [23], or an OpenState Finite State Machine [18], see Section II-C) which are then *enforced* inside the switches so as to reduce the communication needs with the controller and accomplish a dramatically increased (wire-speed!) reactivity of the switch to packet-level events.

B. Security implications of stateful SDN data planes

So far, most of the pioneering work carried out in this emerging stateful SDN data plane field focused on key technical networking aspects to prove the viability and practicality of this new research direction, thus including the identification of new switch architectures [18]–[22], [27], relevant programming abstractions [16], [23], [24], and proof-of-concept use cases and application scenarios [28]–[32].

However, to the best of our knowledge, no literature work so far has yet addressed the *security implications* emerging within stateful SDN data planes. Indeed, stateful SDN data plane switches, while extremely effective in improving performance and programmability, also bring about *novel* security concerns (targeted denial of service and state exhaustion attacks, data plane attacks, and so on) which need to be clearly exposed to the community in these early stages of research in this field, so that they shall be explicitly accounted in ongoing and future research progresses.

Concretely, this survey paper aims at raising awareness about *potential* vulnerabilities of stateful SDN *applications* against a variety of attacks. We use on purpose the term "potential", as *none* among the attacks dissected in this paper seems to be a *fundamental* one, i.e., a "blocking" vulnerability that would bring to the negative conclusion that SDN data planes might be fundamentally insecure[3]. Rather, the take-home message of this paper is that attention to security issues must spread across both system developers as well as developers of applications relying on stateful SDN data plane facilities, so that they can anticipate the possible security concerns likely emerging whenever their designed system or application relies on the following features:

- Store per-flow information, i.e., states, inside switches (including distributed storage of states);
- Ability to change such in-switch states upon data plane packet arrivals or data plane events;
- Ability of the switches to make autonomous decisions based on the local state information.

C. Contributions

This paper has a twofold goal: i) provide the reader with background on the (novel) trend of stateful SDN data planes, and ii) dissect the relevant security issues, also via a concrete analysis of selected use case applications. In the use case analysis we focus on concrete examples: the port knocking application used as motivating example in [18], the UDP flooding mitigation application considered in [16], and the failure recovery proposed in [31]. However, while these concrete use case scenarios permit us to focus on concrete examples, our broader goal is to further provide the reader with guidelines and take-home messages about the conditions that should be taken into account during the implementation of a stateful application in order to avoid falling into security traps. More punctually, the main specific contributions in this paper include:

- A survey of the recently proposed schemes for the stateful SDN data plane (Section II-C);
- An analysis of the vulnerabilities of existing stateful SDN data plane proposals (Section III-A);

[2]See also the keynote talk of Nick McKeown, a key P4 proponent, during last year's P4 workshop in Stanford, https://2ndp4workshop2015.sched.org/event/4e0u/welcome-to-p4org-and-introductions

[3]For an analogy with widely deployed systems, any reader familiar with the security of Ethernet switches knows well that the dynamic operation of a layer 2 switch's forwarding database is prone to several possible attacks [33], including ARP spoofing [34], port stealing [35], resource exhaustion, and so on. However, no reader would conclude that we should *not* use Ethernet switches at all! And in fact tailored defenses to attacks to Ethernet switches do exist (e.g., the well known *port security* feature implemented in most commercial switches).

- A concrete demonstration of potential attacks to applications leveraging stateful SDN data plane switches (Section III-B), focusing on use case examples taken from different literature papers, and chosen so as to highlight different vulnerabilities;
- An implication on the security issues inherited from traditional SDN and security improvements brought by stateful SDN data plane (Section III-C);
- A discussion on possible defense methods and recommendations to design applications which are resilient against the discussed vulnerabilities, as well as possible future research directions (Section IV).

II. From SDN to Stateful SDN Data Planes

In this section, after a brief background on the basic concepts behind Software-Defined Networking (SDN), OpenFlow (Section II-A), and the relevant evolution occurred after the first OpenFlow standardization (Section II-B), we review the state of the art in the area of stateful SDN data planes (Section II-C). Furthermore, we list some relevant use cases enabled by stateful SDN that we will use in this paper (Section II-D). Throughout this section, we assume the reader to be already familiar with SDN and OpenFlow, and thus we limit to provide the basic notions needed for the reader to understand the rationale behind the ongoing evolution from SDN/OpenFlow to stateful SDN data planes[4]. For a more comprehensive overview of *"traditional"* SDN models and frameworks, we refer the reader to the many extensive surveys published, see for instance [17], [36]–[40].

A. SDN and OpenFlow: background

We recall that Software-Defined Networking (SDN) emerged as an attempt to address the problem of how to simplify the control and management of large scale multi-vendor networks. The high level reference SDN architecture promoted by the Open Networking Foundation (ONF) [41] is highlighted in Figure 1. It comprises three main layers: the *infrastructure layer* which supports the data plane operation, the *control layer*, and the *application layer*. The figure further highlights two identified reference application programming interfaces, called (using RFC 7426's terminology) *Northbound API* and *Southbound API*.

The Northbound API is the interface exposed to the SDN applications' developers. It aims to expose simple-to-use high level abstractions devised to hide the complexity inherent in the underlying network topology and network-wide states, as well as to release the network administrator from the need to deal with low level network nodes' configuration details. A huge amount of work has been carried out in this area, see the above mentioned surveys [17], [36]–[40] as well as more specific frameworks such as [9]–[15].

[4]We remark that at the time of writing there is an ongoing discussion in the frame of the Open Networking Foundation about the possibility to *already* include state tables and state transitions as early as in the very next version 1.6 of the OpenFlow standard, thus potentially accelerating such a foreseen evolution.

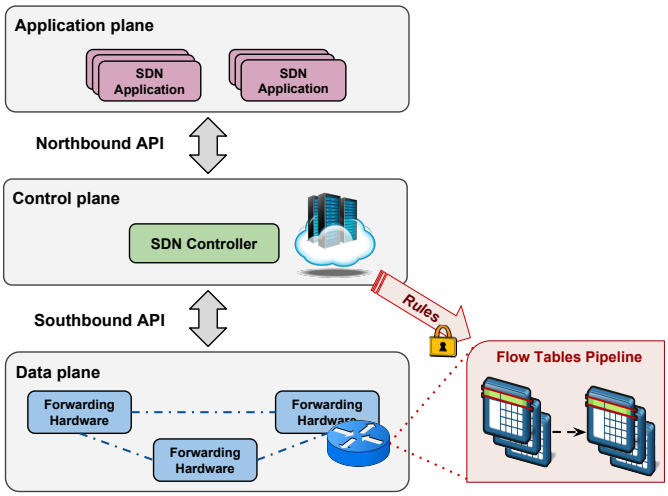

Fig. 1: SDN Architecture

Rather, in this paper we are mostly concerned with the Southbound API, as this is the interface that stateful SDN data plane proposals do explicitly revisit. Until very recently, and despite the fact that the SDN architecture does not restrict to OpenFlow (a *"minor piece in the SDN architecture"*, according to the OpenFlow inventors themselves [42]) as Southbound switch-level interface, OpenFlow was the main (and perhaps the only concrete) candidate for such role. Indeed, starting from the recognition that several different network devices implement somewhat similar flow tables for a broad range of networking functionalities (L2/L3 forwarding, firewall, NAT, etc), the authors of OpenFlow proposed in [1] an *abstract model of a programmable flow table*, usually referred to as "match/action" abstraction, which was (verbatim quoting [1]) "*amenable to high-performance and low-cost implementations; capable of supporting a broad range of research; and consistent with vendors' need for closed platforms*".

The original [1], [43] OpenFlow's match/action abstraction is illustrated in Figure 2. It consists of three main parts: i) a matching rule which permits the programmer to broadly specify a flow by matching any arbitrary combination of selected header fields from layer 2 to layer 4; ii) one or more forwarding/processing "actions" (natively implemented in the device), freely associated by the programmer to the considered matching rule; and iii) a field devised to collect statistics on the flow identified by the relevant matching rule. As thoroughly discussed in [8], such an abstraction emerged as (and owes most of its success in its being) a *pragmatic compromise*: on one side, it was sufficiently innovative to permit a meaningful level of programmability, well beyond what was possible in commercial switches at that time; on the other side it was almost immediately deployable on commodity hardware already present in most merchant-silicon switch chipset (specifically, Ternary Content Addressable Memories–TCAMs).

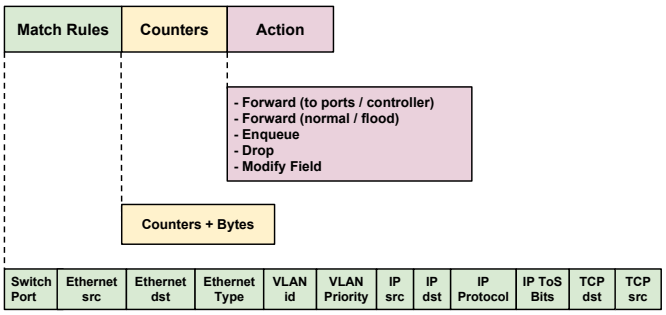

Fig. 2: OpenFlow V1.0.0 Flow Table Architecture

B. The quest for additional flexibility

Soon after the first OpenFlow standardization cycle concertized in version 1.0 [43], it became clear that the original OpenFlow proposal was too restrictive. Subsequent research and standardization work has hence focused on permitting extra flexibility in at least three major directions, as discussed below. The first two directions (i.e., multiple flow tables and improved match/action flexibility) are somewhat independent to the topic of this paper and will be very briefly reviewed, mostly for completeness. More interesting for our purpose is the third direction (i.e., tailored stateful operations), as proposals in this direction anticipate, even if for very special cases, the broader need for stateful processing inside the switches.

1) Multiple Flow Tables

Starting with version 1.1, Openflow was extended to support multiple pipelined flow tables. Indeed, the single table model originally proposed in [1], [43] was considered too restrictive to address many practical networking use cases. On one side, there are several situations in which independent actions are associated to independent matches. The usage of a single flow table would force the programmer to deploy OpenFlow rules which combine the Cartesian product of all the required matches. Obviously, such exponential increase in the deployed rules would lead to an extremely inefficient usage of the flow table capacity, typically implemented via costly and practically constrained TCAM memory. On the other side, many applications naturally benefit of a two-stage processing model [44], where in a first stage packets are first tagged based on some packet characteristics, then more matching and processing occurs. Pipelined flow tables thus were an essential improvement in the earlier OpenFlow versions, and permitted significant savings in terms of TCAM memory and simpler/cleaner configurations. In parallel to the advance in standardization, the research community has further worked on very advanced hardware solutions for more flexible multiple matching. A breakthrough work in this direction is the ***Reconfigurable Match Table (RMT) design*** proposed in 2013 [27], which permits to very flexibly map, over a same physical TCAM array, an arbitrary number of (logical) match tables having different widths and depths, and which even permits to define matching rules using "parameters" computed/extracted from previous matches. It is worth to mention that RMT-type hardware is a key enabling technology for the support of the emerging P4 data plane programming language, discussed in Section II-C2.

2) Improved Match/Action flexibility

Different networking applications at different layers obviously require the ability to associate actions to different types of matches. A greater matching flexibility was immediately recognized as a priority in the OpenFlow evolution. As a matter of fact, the limitation to just 12 matching fields in the original OpenFlow version 1.0 [43] was addressed in subsequent versions. The aftermath is that, today, the latest OpenFlow version considered in conformance products' testing (version 1.3.4 [45]) already supports 40 matching fields, whereas the latest specification produced at the date of writing, namely OpenFlow version 1.5.1 [46], brings this number to 45, and further includes fine-grained fields such as TCP flags. Even more aggressive proposals have been promoted by the research community, starting from the 2013 ***Protocol Oblivious Forwarding - POF*** proposal [47], which supports matches over arbitrary bit patterns extracted from the packet header field via an <*offset, length*> tuple. Similarly, flexibility was significantly enhanced in terms of supported actions, as well as both numbers (more than 60 specified to date) and types (actions, instructions, bundles, etc.). We refer the reader to [39] and other related surveys for details.

3) Tailored stateful OpenFlow Extensions

Since the first version of the standard, several extensions of the OpenFlow protocol has been proposed in order to accommodate specific needs which could not be handled via ordinary flow tables and OpenFlow actions. **Group tables** are probably the most prominent example of OpenFlow extensions which introduce very tailored stateful operations' support. A very concrete and important problem that can be partially solved with group tables is that of link failure. With the original OpenFlow specification, upon a (physical) failure of a link/port, an OpenFlow switch should invoke the intervention of the remote controller to instantiate a new forwarding rule (e.g., forward all traffic originally handled by the failed link/port on a backup link). However, the controller will take some time to react, and in such interval of time all the packets addressed to the failed port will be lost[5]. More recent versions of the OpenFlow standard introduce a group type, called *fast failover*, which specifically addresses this problem. A fast failover group comprises multiple "action buckets" which are evaluated in an order defined by the group, so that the first "live" bucket is selected. Another group type, called *select*, permits to execute one bucket among those specified for the group, and hence support load balancing. Despite solving significant issues, both fast failover and select group types are not only still *optional* in the latest OpenFlow specification, but also loosely specified, with crucial details (such as how to configure a load balancing policy for a "select" group type)

[5]For a layman example, a 100 ms reaction time on a 10 Gbps link would cause up to 30 million minimal-size packets to be lost!

being left to the implementation and/or considered external to OpenFlow. Finally, group tables are of course not the only example of tailored stateful OpenFlow extensions. Other significant examples are **meters** for rate control, **synchronized tables** for supporting learning-type functionalities, and so on - we refer the reader to the latest OpenFlow version 1.5.1 [46] for details. Rather, the point we would like to stress here is that, the evolution of the OpenFlow standard further confirms the need to support *some* stateful operations directly *inside the switch itself* so as to prevent the excessive delay that the explicit involvement of a remote controller would bring.

C. Stateful SDN data plane proposals

As discussed in Section II-B, the OpenFlow community has already recognized the need to bring *some* very specific stateful operations directly in the switch itself, so as to reduce the switch-to-controller's signaling overhead and the latency shortcomings induced by the need to rely on the remote controller *for any possible update in the forwarding state*. The proposals discussed in this section, which we refer to in most generality as "Stateful SDN data planes" [16], [18]–[20], [23] specifically challenge the goal of overcoming OpenFlow's limitations and providing *more expressive programming abstractions* to the data plane, so as to make it possible to configure (as much general as possible) *stateful* operations inside the switch.

Figure 3 provides an overview of a general stateful SDN data plane concept. The basic idea of the stateful SDN can be summarized in two main principles:

- Keeping the state information of the flows inside the switch and allow programmatically formalized packet-level state transition;
- Giving the capability of making forwarding state updates decisions to the switch, based on the local state information of the flows without the need to contact the controller.

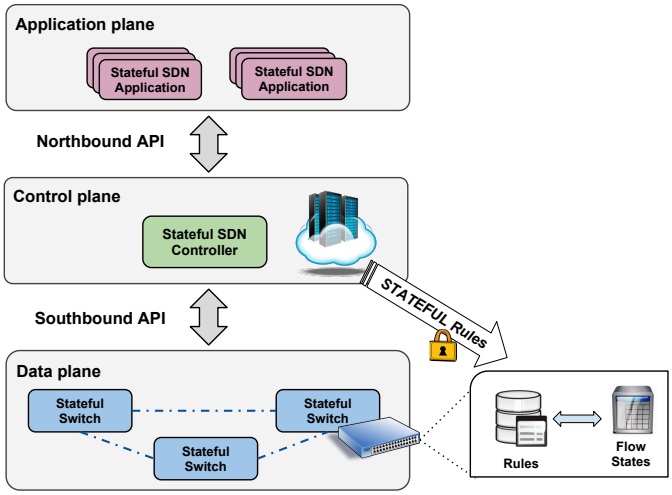

Fig. 3: Stateful SDN Architecture

In particular, depending on the application, a stateful switch may need to store a historical information (i.e., state) of the incoming/outgoing packets. This information can be changed due to receiving/sending new packets of the same flow. Therefore, in such a switch, a packet-level event may lead to the flow's state information transition. Thereafter, the switch takes the forwarding decision for the corresponding flow, based on the current state of the flow. It should be noted that in the stateful data plane, the same as the basic SDN switches, routing decisions are based on a list of pre-defined action rules imposed by the controller. However, the stateful nature of the data plane gives the opportunity to choose an adequate action based on the historical information about the flows, independently.

Although several technical differences of course emerge, and work has focused on different "levels" (core platforms, programming languages and compilers, network-level frameworks), the underling principle of flexible and very efficient forwarding state reconfigurability and data plane programmability is common among the various emerged proposals. In what follows, we provide a survey of the so far proposed approaches. For convenience of the reader we organize the presentation in two subsection: core platforms in Section II-C1, and programming frameworks in Section II-C2. A summary of the features of the schemes discussed in the following is provided in Table I.

1) Platforms and Enabling Technologies

a) **OpenState:** In 2014, researchers proposed *OpenState* [18], which adopts a special class of eXtended Finite State Machines (XFSM), namely a Mealy Machine, in order to provide data plane programmability for SDN. The OpenState work has been very recently further extended in [21], where support for "full" XFSMs (opposed to the original Mealy Machines) has been shown. In the following, we explain the stateful programming model and packet processing procedure in OpenState. The XFSM is modeled by a 4-tuple (S, I, O, T), where S is a finite set of *states* including a starting state S_0 (Default); I is a finite set of inputs (*events*); O is a finite set of outputs (*actions*); and T is a state transition function (*rule*) that maps <*state, event*> pairs into <*state, action*> pairs. In OpenState, each switch stores two distinct tables (see Figure 4): 1) *State Table* that stores the current *state* of the flows based on received packets related to that flow; and 2) *XFSM Table* which is used to define the rules (i.e., state transition) based on the received packet's <*state, event*> information. In order to handle an incoming packet, each OpenState switch performs three steps:

1. In the first step, upon receiving a packet, the switch performs state table lookup to retrieve the packet's current state (i.e., the state of the flow that the packet belongs to); it uses flow ID (e.g., source IP address) as the key for table look-up. If there is no match in the state table, the switch assigns "Default" state to the incoming packet (i.e., flow ID). Then the switch appends the retrieved state label (or Default label) to the packet as a metadata field;

TABLE I: Summary of Stateful SDN data plane schemes features.

Scheme	Category	State Storage	State Removal	Communication with the Controller
OpenState [18]	Platform	Hash table + TCAM for storing state machine	1-Time-out; 2-Application-specific (XFSM transition to the default state)	1-Install rules; 2-XFSM
FAST [19]	Platform	Hash table	N/A	1-Install rules; 2-Controller update per state transition (specific applications)
SDPA [20]	Platform	TCAM + SRAM	Controller request; Time-out	1-Install rules; 2-Initiate state tables for an application; 3-Allocate state table to each flow (first packet); 4-State update
RMT [27]	Enabling Technology	N/A	N/A	Install rules
P4 [23]	Compiler and Programming Language	RAM or TCAM	Application-specific	Compile program on target machine and implement
Domino [22]	Compiler and Programming Language	Banzai (Specific hardware)	Application-specific	Compile program on target machine and implement
SNAP [16]	Framework	Hash table or CAM	Application-specific	1-Install rules; 2-Define new state variables and their placement
Event-driven Programming [48]	Framework	Registers	Application-specific	1-Install rules; 2-Update the controller in case of state transition due to an event

2. The second step is the XFSM table lookup to find the matching rule with <*state, event*>, performing the associated action and updating the state field of the packet based on the pre-defined "next state" field of the XFSM table;
3. Finally, the switch updates its state table based on the retrieved "next state" field from the previous step for the corresponding flow ID.

It is evident that the switch does not need to contact the controller for each received packet. Instead, it makes routing decisions based on the received packet's current state and the pre-defined rules for each <*state, event*> pair by the controller.

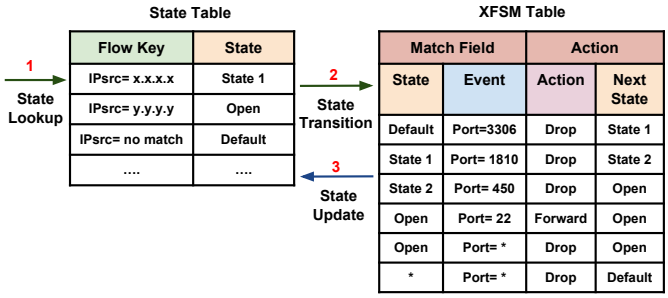

Fig. 4: OpenState tables: State Table, XFSM Table. Entries are related to a port knocking example (explained in Section II-D3).

b) **FAST:** In [19], researchers proposed another stateful data plane abstraction, named *Fast*, which has pre-installed state machines inside each switch similar to OpenState. In FAST design, there could be several instances of state machine inside a switch, each of which dedicated to a special application. The control plane and data plane defined in the FAST platform is different from the original SDN implementation, in the following aspects. The control plane has two main components: (i) a compiler, and (ii) switch agents. The compiler, which is an offline component, translates the state machines into switch agents, while the switch agents, which are online components, manage the state machines inside the switches. The switch agent basically has the following tasks: (1) it takes care of the state machine functionality inside the switch; (2) it can perform some computations locally by receiving updates from the switch, e.g., in case of rate limiting, it receives periodic statistics in order to compute the flow rate; (3) it can manage memory restrictions for confined switches by implementing only a part of the state machine inside the switch, e.g., the switch does not process some "rare events" and just sends these kind of packets to the agent for processing; (4) finally, it handles the communication between the switch and the controller, and updates the controller about the local status of the switch, e.g., if a switch detects heavy hitters, it updates the switch agent which subsequently updates the controller.

In FAST, the data plane consists of four tables (see Figure 5): (i) *State Machine Filter*, (ii) *State Table*, (iii) *State Transition Table*, and (iv) *Action Table*. There is one state machine filter table for all the instances of the state machine inside a switch. However, each state machine has its own state table, state transition table, and action table. The state machine filter table selects the corresponding state machine related

to an incoming packet, while the state table stores the state information of the incoming flows. State table is a hash table which maps each packet header (e.g., source IP, or destination IP) to the corresponding flow's state. Each state is stored inside a variable along with its current value, i.e., considering a number, some high bits used to store the state and other bits represent the corresponding value. State transition table is used to define the next state of the incoming packet based on its current state and packet fields. Finally, the action table specifies an action to be performed for an incoming packet, based on the packet header and its new state.

In [19] the authors discussed some research challenges, such as state machine verification, that are in place when using a state machine inside the switch. Moreover, they explained that the controller might want to have the control of network status and online debugging of the network, and proposed switches to send a copy of packets to the controller per state transition. As another challenge, they mentioned possible state conflict in case of the packets that should pass by more than one state machine. They expressed that the compiler should take care of such a scenario.

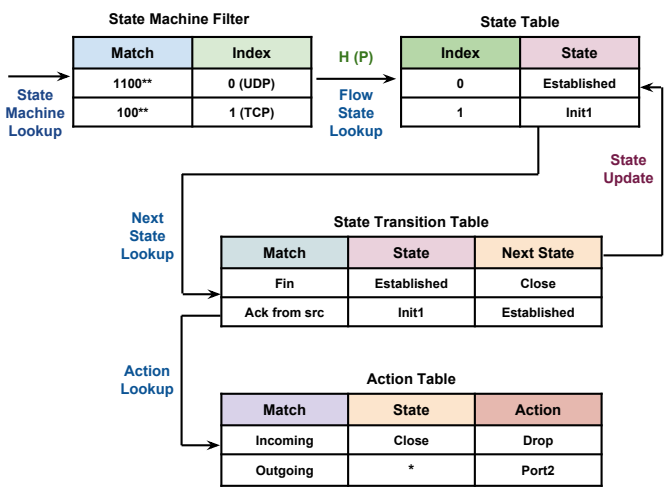

Fig. 5: Fast tables: State Machine Filter, State Table, State Transition Table, and Action Table (for TCP state machine example [19]). H(P) indicates the hashed packet fields

c) **SDPA:** Another recently proposed stateful SDN scheme is *SDPA* [20]. The SDPA platform consists of three tables (see Figure 6): (i) *State Table*, (ii) *State Transition Table*, and (iii) *Action Table*, as well as a state processing module, so-called *Forwarding Processor* (FP). The *match* field of the state table could be any combination of the header fields of the packet. The *state* field is the flow's current state, and the *instruction* field could be a "state operating instruction" or a "packet processing instruction", e.g., GOTO_ FT(m) which means go to flow table with ID "m". The state transition table and the action table define the next state and the corresponding action for the current flow, respectively. In SDPA, each application has its own instances of the three tables. The state table dedicated to an application is initiated by the controller upon receiving the stateful processing request. In such a case, the controller updates the FP about the associated state table to this application and required fields in the table.

In SDPA, contrary to the two previous platforms (i.e., OpenState and FAST), the SDN controller communicates with the FP module and initiates the state tables. Upon receiving the first packet of a flow (e.g., f_i), the switch sends the packet to the controller in order to determine the state table that should store the corresponding flow's state. The other incoming packets from flow f_i will be processed locally inside the switch without the need for communicating with the controller. However, the controller has full control and updated information of the state tables through receiving updates from the FP (i.e., periodic updates or due to a specific event), for which the controller decides when to receive the update (not necessarily per state-transition).

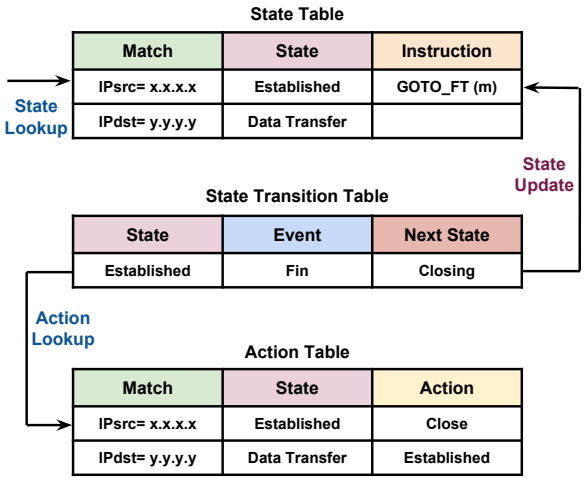

Fig. 6: SDPA tables: State Table, State Transition Table, and Action Table

To the best of our knowledge, by the time of writing, the aforementioned three stateful data plane platforms are the only proposals in the literature.

2) Compilers, Programming Languages, and Frameworks

a) **P4:** In 2014, an abstract forwarding model and a high-level programming language for configuring the switches proposed in [23], in which all the switches are able to process packets statefully. In particular, the authors proposed a target-independent abstract model for packet processing, in which the programmer does not need to know the specifications of the underlying forwarding device in terms of the device type (e.g., switch, router) or technologies (e.g., NPU, FPGA). Therefore, the programs written in the proposed language, i.e., language for Programming Protocol-independent Packet Processors (*P4*), can be mapped to several kinds of devices.

Different from OpenFlow, P4 presents a programmable packet parser (based on the proposal in [49]) which enables switches to parse new header fields. Moreover, P4 allows the match/action selection to be in parallel, in series, or both;

however in OpenFlow it could be only in series. P4 has two main operations: (i) *Configuration*, in which the packet parser is configured, order of the actions is defined, and the way that a switch should process the packets is specified; (ii) *Population*, which defines the policies and match/action table entries. Inside a switch, each packet is first processed by the *parser* in order to extract the header fields, which are in turn sent to the match/action tables. The match/action tables are divided into: (i) *Ingress*, based on which the switch decides what to do with the packet (e.g., forward, drop); (ii) *Egress*, based on which the switch modifies the packet header.

Based on the explained abstract model [23], the authors proposed a compilation process of the packets, which consists of two steps: 1) programmers determine the packet processing procedure using P4; 2) the program written in P4 is translated into a Table Dependency Graph (TDG), which defines dependency between the header fields, corresponding actions and flow process between the tables. The compiler, subsequently maps the TDG to the target switch based on the available hardware resources (e.g., software switches, hardware switches with RAM or TCAM, switches having few physical tables). P4 consists of five main components: (1) *Headers*, which describe the header fields of the packets (e.g., field name, field width); (2) *Parser*, which defines the state machine that should traverse packet header. The state machine is defined as a set of state transitions which is actually a transition from one header to another header. By matching the header field values with match/action table rules, the switch handles the packet; (3) *Tables*, which are used to process packets. By using match/action tables the programmer defines the match conditions (e.g., exact match, wildcard), and corresponding action to each match. Furthermore, based on the table specifications defined by the programmer, the compiler chooses a memory type for the table (e.g., TCAM or SRAM); (4) *Actions*, which define a set of complex actions (e.g., add tag) based on some primitive actions (e.g., add header, set field); (5) *Control programs*, which specify the order based on which the packet should traverse the match/action tables.

It is worth noticing that, the programmable parser [49] that is used in P4 adopts state tables to perform stateful processing. In a recent work [50], researchers proposed HyPer4, a virtualization solution built on top of P4 that allows to (virtually) support different networking contexts.

b) **Domino:** Recently, researchers proposed a data plane programming language, so-called *Domino* [22]. Domino adopts "packet transaction" sequential code block proposed by the authors by which the programmer only concerns about the operations on one single packet. Moreover, the authors proposed a specific hardware, named *Banzai*, as line-rate programmable switch. In Banzai, taking into account the challenges which are in place to perform stateful operations in line-rate switches, the authors provided a programmable instruction set. Moreover, the authors also provided a compiler for Domino. The Banzai hardware model is composed of a vector of processing units, called *atoms*, inside a pipeline of physical stages which model each ingress or egress pipeline of a switch. Atoms enable line-rate atomic stateful processing of the packets in one clock-cycle. In particular, atoms may store internal state variables for the packets, as well as being able to modify the global state variable stored on the switch.

In order to have a stateful data plane algorithm, the programmer provides an application written in Domino, which in turn will be compiled by the Domino compiler. The compiler provides proper configuration of the program for the Banzai machine's atoms through three phases: (i) normalization, (ii) pipelining, and (iii) code generation. In the normalization phase, the compiler simplifies the packet transactions performing the following actions: 1) removing the code branches, 2) restricting state variable operations to simplify handling states during the pipelining, 3) simplifying read and write dependencies by replacing every assignment to a packet field with a new packet field, 4) converting the code into three-address code, in which all instructions are operations on packet fields or state variables. During the pipelining phase, the compiler transforms the sequential code into a pipeline of *codelets* (i.e., a block of three-address code), which in turn will be mapped to the atoms inside the Banzai stages. To this end, the compiler produces a dependency graph and transforms it into a Directed Acyclic Graph (DAG) in order to restrict the operations on "one" state variable to "one" atom. In the code generation phase, the compiler makes a pipeline of atoms for the Banzai machine considering resource and computational limits.

An important feature of Domino is that the state variables are not shared between the atoms. Since each atom performs atomic operations on the packets, Domino protects the data plane algorithms from having state access conflicts. However, Banzai only works well for data plane algorithms having small set of packet headers and stateful operations.

c) **SNAP:** In [16], the authors introduced a programming language and a compiler for stateful data plane, so-called *SNAP*. Using SNAP, a programmer is able to define some global variables and arrays in order to store the state information of the flows. SNAP policies are defined for an abstract network topology, in the sense that the programmer does not need to know how and where to store the state information, since the data plane seems to be a "one-big-switch". The authors also proposed algorithms for compiling the SNAP programs in order to detect policy errors, as well as the state variables placement decision making.

Going more into details, SNAP has a function, so-called *eval*, which takes as input a packet along with a starting state, and outputs a set of packets and the next state. The *state* of the incoming packets are stored inside array-based global variables that are defined by the user. The policies defined by SNAP are distributed through out the network switches by the SNAP compiler automatically. Moreover, the compiler takes care of the atomic update of network variables and states. In fact, before installing the rules inside the switches, the SNAP compiler should know where to store the state variables, and how to route the packets inside the network, ensuring that the packets will pass through the required state variables. To

this end, the authors proposed the use of a Mixed-Integer Linear Program (MILP) in order to decide the packet routing and state placement inside the network, as well as ensuring the correct order of packets passing through different state variables. So, having the decision results from the MILP, the compiler generates the corresponding rules, which are subsequently installed inside the switches by the controller.

We can infer that in SNAP the design of the network topology in terms of state variables, where to be stored, and how to be processed are decided by the controller, and hence the controller has a general view and control of the network status and state variables. However, the incoming packets to each switch are processed independently without the need to contact the controller. This means that, each switch sends the packet to a place that the state of the packet should be analyzed (i.e., another switch that stores the corresponding state variable), and thereafter the packet will be processed based on its current state. It should be noted that, SNAP stores each state variable in only one physical switch, to avoid state synchronization between the switches. However, the authors [16] also mentioned, in case of being interested in distributing the state variables through several switches, it is possible to divide a variable, e.g., s, into several parts, e.g., s_1 to s_k, each of which dedicated to "one" port/IP, and store each s_i in a different switch. In order to implement the state variables in hardware, the authors proposed the use of *hash tables* or *Content Addressable Memory (CAM)* considering a trade-off between efficiency, latency and state consistency.

d) **Event-driven Programming:** In [48] researchers proposed semantics of stateful processing in the network for event-driven updates. The authors defined an *"event-driven consistent update"* to be moving from a configuration to another upon occurrence of an event. The event could be the required changes in the forwarding behavior of a switch upon receiving a packet that triggers the update. Their proposal ensures that, the packets are processed by a consistent configuration throughout the network, and the network updates in all the switches happen at the correct time (i.e., not too early and not too late). To this end, they considered a system named *Event-driven Transition System (ETS)*, which is similar to the state machines adopted by the stateful data plane platforms that we explained in Section II-C1. In order to prevent conflicts in the network, which might happen due to different switches' view point about the network events, the authors proposed a new structure called *Network Event Structure (NES)*. In particular, NES considers two transition constraints: (1) dependency between the network events, and (2) compatibility between the events.

In order to implement a stateful program, NES helps the network designers to program the switches in such a way that switches are able to store the incoming events, route the packets based on the events and announce other switches about the event. In particular, the proposal in [48] works as follows: (1) embeds the events presented as tags inside the packet headers; (2) defines guards for transition between different configurations, which could be triggered due to reception of a packet having specific tag; (3) stores local events in the switch as "state"; (4) changes the switch configuration based on the switch local state and switch's view of the global state. In particular, each packet entering the network carries a global state of the network due to the switches that it traverses. This way, if the packet passes by a switch, whose local state is different from that of the packet, it will be updated to the latest state. In order to provide the state consistency, the controller receives the state transition information from the switches, though the controller can also update the switches periodically based on its global view of the network.

D. Stateful SDN enabled applications and use cases

In recent years, researchers proposed applications running on top of stateful SDN data plane, adopting the proposed platforms (which are introduced in Section II-C1) and programming languages (which are introduced in Section II-C2). In this section, we introduce some example applications enabled by stateful SDN, and use cases introduced in the literature to show the usability of the stateful SDN. We will later implement some of these applications and use cases for our vulnerability analysis (Section III-A), and attack assessment (Section III-B). In particular, we present: (i) *stateful failure recovery*, an application for fast node/link recovery using local states of the switches (Section II-D1); (ii) *HULA*, a data plane load-balancing application (Section II-D2); (iii) *port knocking* use case, an access control technique that allows a user to "pass through" a firewall after "knocking" a specific sequence of ports (Section II-D3); (iv) *DNS tunneling stateful detection*, which detects users who intend to bypass access policies to reach a host using DNS messages (Section II-D4); and (v) *UDP flooding stateful mitigation*, which recognizes users who send an aberrant number of UDP packets (Section II-D5).

1) Stateful Failure Recovery

The work in [31] proposes a controller-independent stateful link/node failure detection and recovery scheme, based on OpenState. This solution tags packets with labels, upon detecting a node or link failure, in order to communicate with previous nodes to use a detour path for subsequent packets.

The protocol works as follows: let us consider a scenario in which a source host $\mathcal{H}$ is sending a packet p_i to a destination host $\mathcal{I}$, through a network of stateful SDN switches. Upon recognizing a link or node failure, the nearest node to the failure tags the same data packet p_i with a label (e.g., an MPLS label) containing information on the failure event (operation (1) in Figure 7), instead of sending a notification of the failure. This tagged packet is routed back on the original path to a convenient reroute node, e.g., node $\mathcal{N}$ (operation (2) in Figure 7). Upon receiving a tagged packet, node $\mathcal{N}$ performs a state transition for the corresponding flow in its state table and reroutes (through a pre-defined alternative route) all the subsequent packets of the flow afterwards (operation (3) in Figure 7). When reaching to a merge node in the route, the tag will be removed from the packet and the packet will be

forwarded to the destination host $\mathcal{I}$ (operation (4) in Figure 7). The stateful processing feature of this scheme allows the switch to re-route the packets without the need to report the link failure to the controller, since each node (i.e., switch) maintains the state of every flow, it can decide the forwarding path (i.e., normal path, or detour) autonomously.

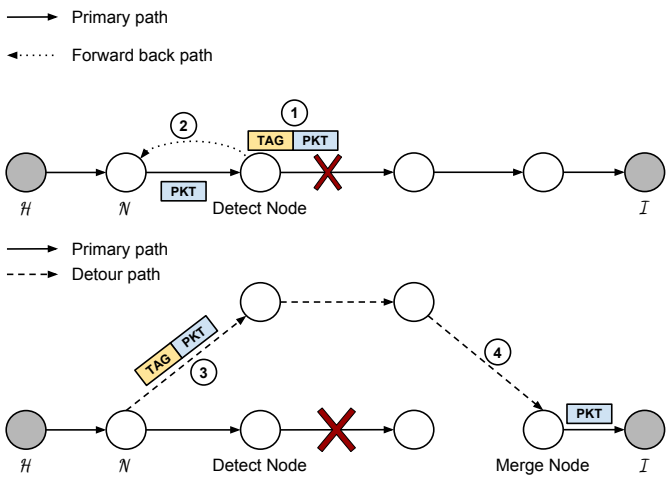

Fig. 7: Failure recovery operations in [31].

2) HULA

Recently, the work in [30] presents HULA, data plane load-balancing technique for data centers, designed to be adopted on top of programmable switches, such as RMT [27], and based on P4 programming language [23]. In HULA, each switch keeps a best path utilization table with the information of the next best hop towards a destination Top of Rack (ToR) switch (i.e., the downstream path). The entries of the utilization table are of the form: <*DestIp, BestHop, PathUtil*>. In order to find the best next hop, each ToR switch periodically sends *probes* throughout the network to gather global link utilization information. Then, based on the received probe, each switch proactively updates its best path utilization table with the best next hop, which balances the load toward a destination throughout the existing paths.

Each probe packet header contains two fields: (i) a 24 bit *torId*, i.e., the identifier ToR switch that originated the probe; and (ii) an eight bit *minUtil*, which carries the utilization of the best path for flowlets (i.e., subcomponents of a flow) traveling in the *opposite direction* with respect to the received probe. Once a switch receives a probe from a port i, it first computes *maxUtil* as the maximum value between *minUtil* and the (local) measured link utilization on port i. Then, the switch computes the minimum value between *maxUtil* and the previous corresponding *PathUtil* value recorded inside its utilization table. If *maxUtil* is the minimum value, the switch updates the fields *PathUtil* and *BestHop* with the values *maxUtil* and *torId*, respectively, in its best path utilization table. Moreover, the switch produces a new probe specifying, inside the header, the latest *torId* and *PathUtil*, which is propagated throughout the network using a simple loop-free strategy. Figure 8 shows the probes propagation strategy of HULA, on a three tier network, starting from a ToR node *ToR1*. Aggregator nodes (*A*1-*A*4) forward the received probe to the downstream (i.e., ToR) nodes, but not to the upstream spines (*S*1-*S*4), unless the probe was received from a ToR node (*ToR1* in Figure 8). When the probe reaches a ToR node, it stops being forwarded.

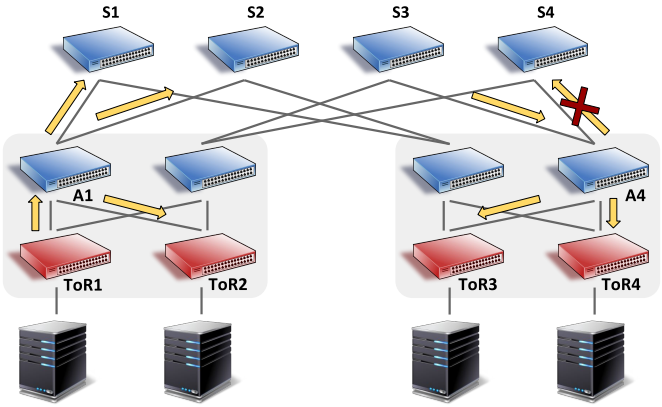

Fig. 8: HULA probe propagation [30].

3) Port Knocking

In [18], the authors considered port knocking as an application to show OpenState architecture and operations. Port knocking allows a stateful firewall to grant access to one or multiple hosts on a specific port (in our case, port 22), only if they are able to successfully conclude a pre-defined (ordered) sequence of connection attempts to different ports. Such ordered sequence represents a "shared secret" between the firewall and the host(s).

Consider a scenario in which a host $\mathcal{H}$ (identified by the IP source address) tries to establish an SSH session (on the default port 22), passing through a firewall $\mathcal{F}$, implemented on an OpenState switch. In our example, let $\{3306, 1810, 450\}$ be the sequence of ports to "knock". If $\mathcal{H}$ sends the exact three ordered requests as connection attempts, $\mathcal{F}$ will authenticate the host and open port 22; otherwise the request will be dropped. Figure 9 illustrates such an example.

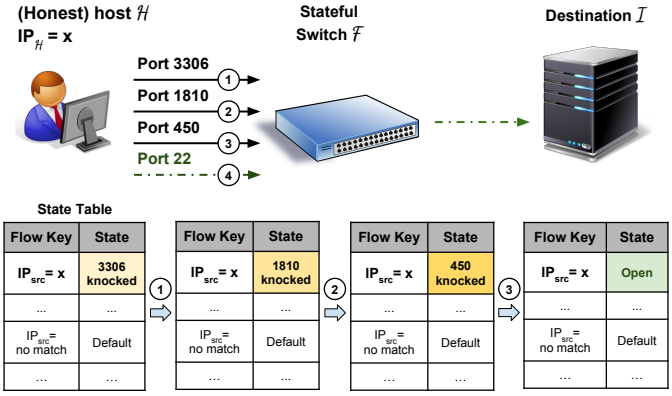

Fig. 9: Port knocking example scenario in [18].

As previously mentioned in Section II-C1, OpenState keeps an XFSM table implementing an extended state machine inside

the switch. In an OpenState port knocking implementation, the XFSM table could look like the one shown in Figure 4, i.e., rules are defined for five events (i.e., packet received on ports 3306, 1810, 450, 22, and any other ports), four states (i.e., *Default*, *State 1*, *State 2*, *Open*), and two actions (i.e., Drop and Forward). Upon receiving a packet from $\mathcal{H}$, the switch performs the state lookup procedure to retrieve the current state of the packet. Starting from the *Default* state, a state transition is triggered if and only if $\mathcal{H}$ knocks the next expected port of $\mathcal{F}$ in the sequence; this leads to a packet drop action, and a state transition to *State 1*. The new state is written back in the state table for the corresponding host ID (IP source). This procedure continues until $\mathcal{H}$ knocks all the expected ports, and then its state will be updated to *Open*. In this state, upon receiving a request for port 22, the firewall opens this port for user $\mathcal{H}$. In any state, if the host sends a request to an incorrect port, its state rolls back to the *Default* state. In OpenState, the authors assumed the XFSM table to be ordered by the priority of the rules. Moreover, they considered a timeout field in the state table, for a state transition to the next state (i.e., next knocked port): if the user does not knock the correct ports within a pre-defined timeout, its state rolls back to the *Default* state.

4) DNS Tunneling Stateful Detection

In this section, we describe the DNS tunneling detection application from [16], which the authors considered as a running example to explain the SNAP stateful framework. In a DNS tunneling attack scenario, a malicious user tries to abuse the DNS messages to bypass the access policies in order to send data. As described in [16], the following steps should be performed in order to detect a DNS tunneling attempt (see Figure 10):

1) Assign a counter $c_{\mathcal{H}}$ to each client $\mathcal{H}$, in order to keep track of all the resolved IP addresses for $\mathcal{H}$;
2) Increase $c_{\mathcal{H}}$ when the client receives a DNS response, and decrease it per used resolved IP address, i.e., when the client sends a packet to the corresponding IP address;
3) Consider a threshold for $c_{\mathcal{H}}$, and report as malicious the client whose counter value is higher than the threshold.

In order to perform the second step, SNAP maintains all the resolved IP addresses destined to client $\mathcal{H}$ in an array-based variable. In particular, the SNAP switch maintains a variable, *orphan*, which maps each pair of source and destination IP addresses, <*src, dst*>, to a boolean value. If a client $\mathcal{H}$ receives a resolved response from the DNS for a destination IP address $IP_{\mathcal{I}}$, the value of the variable at $<IP_{\mathcal{H}}, IP_{\mathcal{I}}>$ will be set to `True` and the value of $c_{\mathcal{H}}$ will be incremented (operations (1) and (2) in Figure 10). When $\mathcal{H}$ sends a packet to $\mathcal{I}$, the switch checks the state associated to $<IP_{\mathcal{H}}, IP_{\mathcal{I}}>$ and, if `True`, it sets this value to `False` and decrements the value of the counter $c_{\mathcal{H}}$, meaning that $\mathcal{H}$ actually "used" the information contained in the received DNS record (operation (3) in Figure 10).

5) UDP Flooding Stateful Mitigation

Here, we describe UDP flooding mitigation, another use case example for the stateful data plane which is presented

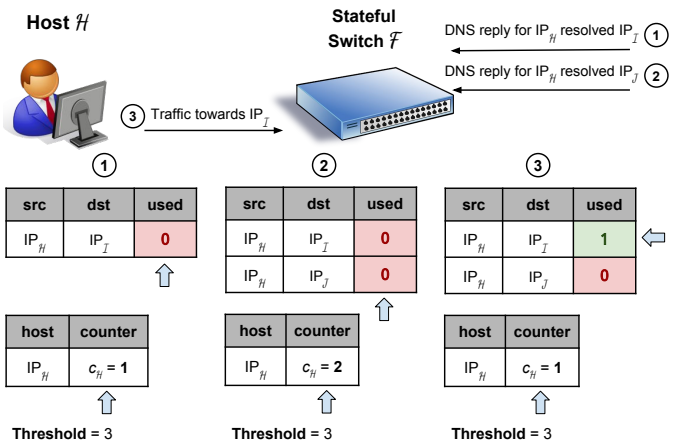

Fig. 10: DNS tunneling example scenario in [16].

in [51] and implemented using SNAP [16]. In [16] the authors provided an algorithm to identify UDP flooding attack by distinguishing the users who send an unexpected number of UDP packets. To this end, the authors considered several variables: (i) a counter $c_{UDP_{\mathcal{H}}}$ for the number of UDP packets originated by host $\mathcal{H}$ (i.e., the source IP of each packet); (ii) a variable *udp-flooder*[$\mathcal{H}$] to keep the state of client $\mathcal{H}$: `True` if $\mathcal{H}$ is UDP flooder and `False` otherwise; (iii) a threshold $\mathcal{T}$ for the legitimate number of UDP packets that could be sourced from an IP address.

When user $\mathcal{H}$ sends a UDP packet, the SNAP policy first investigates the state of $\mathcal{H}$ to check if $\mathcal{H}$ has been already recognized as a malicious user: if *udp-flooder*[$\mathcal{H}$] is `True` the algorithm will drop the packet. Otherwise, if *udp-flooder*[$\mathcal{H}$] is `False`, the algorithm increments the value of the counter $c_{UDP_{\mathcal{H}}}$, and checks if $c_{UDP_{\mathcal{H}}} = \mathcal{T}$, it drops the packet and sets the *udp-flooder*[$\mathcal{H}$] to `True`.

III. Stateful SDN Data Plane Security Issues

In Section II, we surveyed the existing proposals for stateful SDN data plane in the literature. At a very high level, all the stateful SDN schemes we explained in this work share the same design principle: they push down the flow states to the data plane. This allows each switch to take local autonomous and intelligent decisions, with a consequent improvement in the performance of the network (thanks to the reduced communication traffic between the data plane, and the control plane). However, while the benefits of having stateful data plane are evident, we believe that its implications in terms of security are currently underestimated.

In order to focus the community's attention to this important issue, in Section III-A we highlight the main vulnerabilities that are in place due to the intrinsic features of stateful SDN data plane. Furthermore, in Section III-B we describe potential attack scenarios that could occur exploiting the identified vulnerabilities of the stateful SDN data plane. To this end, we consider three use case examples (which are explained in Section II-D) and show the feasibility of the considered

attacks. Finally, it remains a fact that stateful SDN proposals are not meant to solve security issues emerging in traditional SDN, and hence in the last Section III-C we briefly discuss traditional SDN vulnerabilities, and how they apply and/or change when cast in the novel context of stateful SDN.

A. Vulnerabilities

In our analysis of the existing stateful SDN data plane proposals and stateful SDN enabled applications, we identified four main types of vulnerabilities. In this section, we provide a high-level description of these four vulnerabilities: *unbounded flow state memory allocation* (Section III-A1), *triggerable CPU intensive operations* (Section III-A2), *lack of authentication mechanisms* (Section III-A3), and *lack of a central state management* (Section III-A4). Later on, in Section III-B, we provide detailed explanation of the attacks that could exploit these vulnerabilities, considering three practical attack examples taken from the recent literature on stateful SDN data plane.

1) Unbounded flow state memory allocation

In order for the data plane to be programmable (i.e., to have stateful switches), each stateful SDN solution must allocate memory space to keep track of the state transitions generated by incoming flows. Depending on the specific stateful SDN proposal, the data structure used to maintain state information is referred to as "state table" [18]–[20] or "array-based variables" [16]. For simplicity, in the rest of the paper we will use the term "state table" to indicate a generic data structure used for state storage.

In order to attack a switch, an attacker may take advantage of the potentially large in-memory space that each switch requires in order to store flow state information, to exhaust the memory of the switch.

2) Triggerable CPU intensive operations

The second attack vector derives from the possibility for an attacker to "force" the execution of potentially CPU intensive operations on a switch. For example, this is the case of situations where the (virtually) centralized controller must have full control of the network to verify the network functionality [19]. In such a scenario, the control plane requires to receive updates about *every* event in the network, i.e., at every state transition.

In this case, an attacker can perform a DoS attack by forcing the switch to continuously update the control plane, consuming computation resources that the switch will not use for packet processing.[6] It should be noted that, in case of resource exhaustion, some attacks could be related to structural aspects, while others could come from implementation inappropriateness, or programmer-defined policies.

[6]The update packet generation may involve cryptographic operations, as typically, the communication between data plane and control plane is secured, e.g., with SSL/TLS [45].

3) Lack of authentication mechanisms in the data plane

Some recent stateful SDN proposals, e.g., the ones we introduced in Section II-D1 and Section II-D2, are moving towards the use of control plane independent functionalities, such as load balancing and network management. This translates into the use of probe messages between switches in the network [30], or information passing between switches and "piggybacking" inside regular traffic packets [29], [31]. These approaches represent a major shift from a more classic control plane to data plane (only) communication model, and allow for more advanced and performant networking troubleshooting and management. Unfortunately, existing proposals dedicate scarce attention to the security implications of this shift, and in particular, there is a general lack of authentication/encryption mechanism between switches in the data plane.

Leveraging this, an attacker may impersonate an honest switch and inject fake event/packet to the network, or she can manipulate the content of the information passed between the switches. This way, the attacker is able to change a specific flow state inside the switch, thus making the switch take decision based on the spoofed events/packets. This vulnerability could be exploited to perform several types of DoS attacks, such as faking a link failure in the network to degrade the overall network performance.

4) Lack of a central data plane state management

State inconsistency is actually a concern that already exists in traditional SDN, specially when it comes to physically distributed control plane [52]–[54]. However, state distribution in the data plane level in stateful SDN becomes more important and challenging, as no central entity is: (i) involved in the processing of some data plane events; and (ii) responsible for synchronization of states inside the switches. In particular, as state transition is triggered by the received packets, an attacker has the power to influence and force state transitions, to drive the network into an inconsistent state.

We believe that being vulnerable against state inconsistency does not relate to how states evolve inside a switch. Rather, the fact that there is not any controller to synchronize the states might lead to state inconsistency in the network. In order to tackle this issue, one may propose to keep the controller updated about the current state of the state machines that are implemented inside the switch [19]. However, it should be noted that the controller update process should be performed very carefully in order to avoid introducing new potential bottlenecks, and therefore attack points. The state inconsistency could be due to a fake packet injection attack, or might be due to an inappropriate implementation. Nevertheless, it might lead to security or performance issues inside the network.

B. Attack examples

In this section, we provide attack examples to the existing stateful SDN schemes, exploiting the vulnerabilities

highlighted in Section III-A. We consider three of the use case applications introduced in Section II-D, i.e., *port knocking* (Section II-D3), *UDP flooding stateful mitigation* (Section II-D5), and *stateful failure recovery* (Section II-D1), and show potential attacks. For each attack, we list the exploited vulnerabilities. We chose these three use case examples for their simple design, and ease of implementation, which let us have more control on our experiments. We implemented all the use cases on top of the OpenState platform [18] to provide experimental results showing the feasibility of the attacks (refer to Appendix A for implementation details). Table II shows the mapping between the considered attacks, and the exploited vulnerabilities.

In particular, in Section III-B1 we provide two different use case examples that exploit the *unbounded flow state memory allocation* vulnerability (explained in Section III-A1), which leads to a *memory saturation attack*. In Section III-B2 we introduce a *re-routing attack* on the use case application we introduced in Section II-D1, which exploits the *lack of authentication mechanisms* vulnerability (explained in Section III-A3) to impose *state inconsistency*. Finally, in Section III-B3 we discuss how a *CPU exhaustion attack*, which exploits the *Triggerable CPU intensive operations* vulnerability (explained in Section III-A2), may lead to distributed state inconsistency. In each section we provide the necessary preliminaries, and attack scenario. The reader may refer to Appendix A for detailed information about the implementation and experimental assessment of the attacks that we present in this section.

1) Switch Memory Saturation Attack

A first simple and yet effective attack on stateful SDN is flooding switch's state table in scenarios where such a table is continuously extended and updated. Indeed, while in general there is no need to keep the state of all the flows traversing a switch, an adversary could smartly "force" this behavior in specific applications, and exhaust the memory of the switch, leading to a *memory saturation attack*. This is, for example, the case of *port knocking* (Section III-B1a), and of *UDP flooding stateful mitigation* (Section III-B1b).

a) Port Knocking Attack:

We now present an attack on the port knocking application that we introduced in Section II-D3.

Attacker Model: Our attack on the port knocking application, considers an attacker that can be of two types:

- *Informed attacker*. This is an attacker that knows the correct sequence of ports to knock. This information could be gained through several ways such as sniffing the traffic [55];
- *Oblivious attacker*. This is an attacker that does not know the exact port sequence to knock.

Considering these two attacker types, we now describe the possible memory saturation attack.

Informed Attacker Scenario. Assume $\mathcal{A}$ is an informed attacker. As shown in Figure 11, $\mathcal{A}$ sends a large amount of connection attempts (i.e., packets) to the first port in the pre-defined sequence (i.e., 3306) from a set of spoofed IPs, towards a switch $\mathcal{F}$. Upon receiving a packet from a source IP, say 1.2.3.4, $\mathcal{F}$ checks its state table to retrieve the state of the incoming flow. If there is no record for this IP address, $\mathcal{F}$ assigns the *Default* state to the corresponding IP and performs the XFSM table lookup. Now, since all the incoming packets are destined to the correct port, the state of all the incoming packets (i.e., the corresponding IPs) will be updated to the next state (i.e., *State 1*: "3306 knocked"). Therefore, we will have a large number of entries inside the state table of the switch. This way, a conscious attacker is able to force the generation of thousands of records in the state table, and thus exhaust the switch memory.

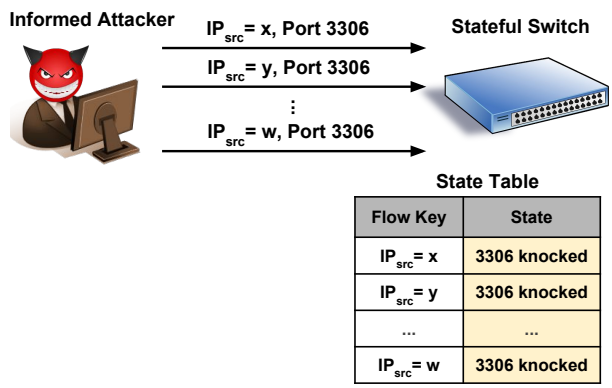

Fig. 11: Informed attacker scenario, i.e., the attacker knows the correct sequence of the ports that should be knocked. This attacker can exhaust the memory by knocking the correct port from several different IP addresses.

Oblivious Attacker Scenario. Consider the case in which $\mathcal{A}$ is an oblivious attacker. In this case, $\mathcal{A}$ can perform the attack by generating a large number of packets from a set of spoofed IPs (e.g., $IP \in D_1 = [128.0.0.1 - 128.0.255.254]$) towards all the ports of $\mathcal{F}$, i.e., 65535 packets from 65535 spoofed IPs (see Figure 12). Assuming that none of the selected IP addresses are used before, $\mathcal{F}$ assigns the *Default* state to all the corresponding IPs and performs an XFSM table lookup. Note that, if the programmer, due to an inappropriate implementation, stores one *Default* record per flow (i.e., for each of the incoming flows, allocates one record in the table), then the result will be 65535 records in the state table. If the attacker continues flooding all the ports of $\mathcal{F}$, it will successfully complete the memory saturation attack. However, in a "correct" implementation, there should be only "one" record for the *Default* state. Now, two conditions might happen for the incoming packet from IP 1.2.3.4:

- Accidentally, $\mathcal{A}$ has knocked the correct port (i.e., 3306). In this case, the state of the IP 1.2.3.4 will be updated to the next state (i.e., *State 1*);
- $\mathcal{A}$ did not knock the correct port, and its state will remain as *Default*. In this case, there is no need to keep the state of IP 1.2.3.4.

Now, since $\mathcal{A}$ can easily perform the packet flooding

TABLE II: Mapping between the potential attacks described in Section III-B and the corresponding vulnerabilities explained in Section III-A.

Potential Attacks	Exploited Vulnerabilities	Attack Goal
Switch memory saturation attack (Section III-B1)	Unbounded flow state memory allocation (Section III-A1)	DoS on the switch
State inconsistency attack (Section III-B2)	Lack of authentication mechanisms (Section III-A3)	Impose state inconsistency, degrade network performance, impose delay
CPU Exhaustion Attack (Section III-B3)	Triggerable CPU intensive operations (Section III-A2)	Impose state inconsistency, DoS on the switch

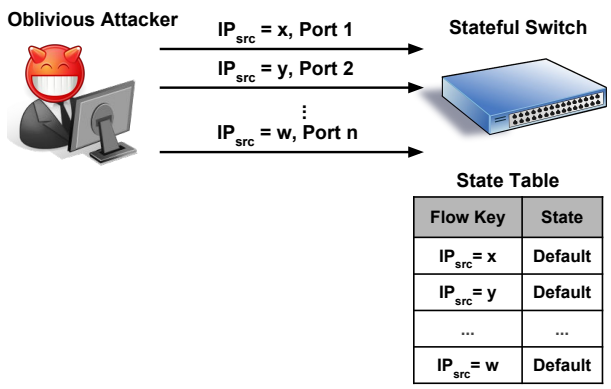

Fig. 12: Oblivious attacker scenario, i.e., the attacker does not know the correct sequence of the ports that should be knocked. This attacker can exhaust the memory in case of sloppy implementation, by knocking all the ports from several different IP addresses.

periodically, she might choose the IP addresses as follows:

- Select an $IP \notin D_1$: this case, she will have the chance to knock the correct port and will have one new record for the correct knocked port in the state table with one of the IPs (e.g., IP 5.6.7.8);
- Select an $IP \in D_1$: if she knocks a wrong port with IP 1.2.3.4, based on what has been explained in [18] the state of this IP in the state table should be changed to *Default*. Note that, again if we have a non-carefully designed implementation, instead of overwriting the existing record in the state table (and actually removing it from the table), the switch may just set a new *Default* state for IP 1.2.3.4.

Once the switch memory is exploited by the adversary, it is not able to process the new incoming packets which could be issued by a legitimate user. Now, the switch could decide to perform one of the following actions:

- Send the packet to the controller: in this case, its behavior is the same as traditional SDN;
- Drop the packet: this leads to DoS to a legitimate user;
- Overwrite on one of the existing records in the state table: the challenge is how to choose the record to overwrite.

To tackle the random port scanning attack, authors in [28] proposed to consider a new state, so-called *attack state*. The authors considered a metric M_1 to count the SYN rate used by a host. If the switch detects a port scan by a flow, it updates the status of the corresponding flow to "attack state", blocks the host for two minutes, and triggers an alert. However, this solution could not mitigate our proposed memory saturation attack.

b) Attack the UDP Flooding Mitigation:
Here, we show the vulnerability of the stateful UDP flooding mitigation application introduced in Section II-D5 against a memory saturation attack.

Attack scenario: As explained earlier in Section II-D5, SNAP stores a counter for the UDP packets originated by each client, as well as the current state of the client, i.e., if the client is a UDP flooder or not. Therefore, if an attacker sends UDP packets from a large number of different IP addresses, SNAP will allocate a state variable and a counter to each of the source IP addresses. This way, the attacker is able to easily exhaust the dedicated memory to these two variables.

In order to mitigate such an attack, one may think of considering a time-out for the counter and the state variable. Meaning that, if the switch does not receive any UDP packet from a client $\mathcal{H}$ during a pre-defined amount of time (i.e., the time-out value), it can free the memory space allocated to the client $\mathcal{H}$. However, a "smart" attacker can send UDP packets periodically in order to force the switch to keep the state of the corresponding host.

2) State Inconsistency Attack

Another possible attack to the stateful SDN is *state inconsistency*, which comes from the *lack of authentication mechanisms* vulnerability and leads to *event spoofing* and *re-routing attack*. As we explained earlier, in stateful SDN, each switch makes the routing decision for the incoming packet based on the <*state, event*> information of the corresponding flow. On the other hand, a packet-level event in the network leads to state transition for a specific flow inside the switch. Therefore, an attacker can easily perform a fake packet injection and impose state transition for a specific flow. This leads to inconsistency between different instances of the stored states, inside different switches, for the same flow. Such state inconsistency, consequently, enforces the switch to make attacker-desired routing decisions.

In what follows, we elaborate this attack more clearly through the *stateful failure recovery* use case example (explained in Section II-D1). In particular, we show how we

can mount a re-routing attack by inducing inconsistent state into the stateful switch. It is worth noting that, such kind of attack can also be applied to any stateful load balancing application that makes use of in-band signaling between switches, such as the one in [30]. In a load balancing scenario, an attacker may redirect the network load to an inappropriate link, leading to link failure.

a) Re-routing Attack:

We now consider the detour planning scheme proposed in [31] (and introduced in Section II-D1), and we use it to illustrate how an attacker can perform re-routing attack.

Attack scenario: In this scenario, an attacker can inject fake packets, which are tagged with "broken link", and spoof a link failure event as follows: the attacker eavesdrops the exchanged traffic between two nodes (e.g., node $\mathcal{N}$ and "Detect Node" in Figure 7), captures a packet, tags the packet by appending a label saying the forwarding route is broken, and sends back the tagged packet to $\mathcal{N}$. Upon receiving a counterfeit tagged packet, $\mathcal{N}$ performs a state transition for the corresponding flow and detours all the pursuant packets.

Based on what explained above, it would be easy for an attacker to perform a re-routing attack, and impose inconsistency and latency to the network, as well as decreased performance. This attack is actually possible due to the fact that there is no authentication and/or encryption mechanism for the exchanged traffic between two switches inside the stateful SDN architecture. Note that, this attack does not apply in cases where all the switches inside the network can be considered as honest.

3) CPU Exhaustion Attack

Another possible attack to the stateful SDN, is application specific. As explained in Section III-A2, in some networking applications, such as network monitoring and DDoS detection, the controller needs to have an updated complete overview of the network. To this end, one might design the data plane in such a way that the switch updates the controller per state transition (as proposed in [19]). In such a scenario, an attacker can try to impose a large number of state transitions inside the switch in two ways:

- Sending a large number of new packets leading to insertion of a new entry inside the state table;
- Sending several packets related to an ongoing flow which already has a record inside the state table. This results in per packet state transition for the corresponding flow.

In both cases, the switch should update the controller about the new state of the flows, resulting in a large amount of packet processing and packet exchange between the switch and the controller. This will lead to the exhaustion of the CPU processing power of the switch. Since the CPU is only able to process a certain number of packets per second, due to the explained attack, stateful processing of the packets will fail.

Another consequence of such an attack is state inconsistency between the switch and the controller. Let us consider a scenario in which switch s_1 is an edge switch and runs an intrusion detection system. In particular, s_1 is responsible for detecting malicious IPs and updating the controller. The controller should consequently install new rules for the detected malicious IPs inside the switches. Now, consider an attacker $\mathcal{A}$ that wants to access the network resources. Having knowledge about the s_1, the attacker $\mathcal{A}$ (or some colluding attackers) could perform the explained CPU exhaustion attack. In this case, since s_1 is under attack, it will not be able to process any packets or update the controller. Hence, $\mathcal{A}$ will be able to bypass s_1 and access the network.

C. Security issues inherited from traditional SDN

In the previous subsections, we specifically discussed vulnerabilities and attacks which are *newly* emerging in the context of stateful SDN. We here remark that stateful SDN proposals modify the traditional SDN design by introducing a stateful data plane. For this reason, stateful SDN naturally inherits most of its vulnerabilities, in particular the ones regarding the application plane, and its interaction with the control plane. In what follows, we (very briefly) summarize existing traditional SDN vulnerabilities, and discuss: (i) which of them still affect stateful SDN; (ii) when applicable, the (implicit) security improvements brought by stateful SDN; and (iii) the new challenges (w.r.t. traditional SDN) that stateful SDN introduces. In this section we restrict our analysis to SDN control plane and data plane vulnerabilities (as they are more related to the topic of this paper) that have a non-obvious mapping to stateful SDN. The reader interested in a broader discussion may refer to the extensive literature that has specifically addressed security in (traditional) SDN, see e.g., [2]–[7].

Table III provides a summary of the main vulnerabilities of traditional SDN, extracted from previous surveys [2]–[7], plus the vulnerabilities identified in this work. For each traditional SDN vulnerability, Table III indicates which one affects also stateful SDN, and vice versa. Note that, while existing work in the literature mainly focus on "attacks" or "security issues" in the traditional SDN, this section discusses (and focuses on) *vulnerabilities*, by design, of both traditional SDN and stateful SDN. Thus, the list of vulnerabilities reported in Table III have been derived from the attacks reported in [2]–[7] (since one vulnerability can lead to more than one attack/issue, there is no 1:1 mapping with the literature). We divided the vulnerabilities listed in Table III into three categories (i.e., control plane, data/control plane, and data plane), which we will explain in the following.

1) Control plane vulnerabilities

Stateful SDN leaves the SDN control plane practically unchanged, and therefore inherits most of its vulnerabilities. Indeed, even if stateful SDN moves some states "down" to the data plane, the core logic is maintained by the (logically centralized) control plane; as such, it suffers from potential issues generated by an inconsistent global view of the network, e.g., in a distributed controller setting as a consequence of a controller crash or disconnection

Affected Plane	Vulnerability	Traditional SDN	Stateful SDN
Control plane	Inconsistent global state (e.g., controller crash in a distributed control plane)	☹	☹
Data/Control plane	Poor scalability for switch-to-controller communication	☹	☺
Data plane	Data plane side channels (e.g., switch configuration leakage)	☹	☺
	Exhaustible TCAM (flow table) memory	☹	☺
	Unbounded flow state memory allocation (Section III-A1)	N/A	☹
	Triggerable CPU intensive operations (Section III-A2)	☹	☹
	Lack of authentication mechanisms in the data plane (Section III-A3)	N/A	☹
	Lack of a central data plane state management (Section III-A4)	N/A	☹

TABLE III: Main vulnerabilities "by design" in traditional SDN according to [2]–[7] and in stateful SDN according to Section III (☹ indicates the presence of the vulnerability, ☺ indicates a reduced exposure to the vulnerability, while N/A indicates that the vulnerability does not apply).

(see [52]–[54]). Moving states to the data plane, however, may eliminate the vulnerability for very special applications: as an example, consider the port knocking application we presented in Section II-D3. In this case, the operations performed by the switch are completely local, and independent from the global potentially inconsistent state maintained by the control plane.

2) *Cross data/control plane vulnerabilities*

The major improvement in terms of security of stateful SDN with respect to the traditional SDN is related to the reduced required communication between the data and control plane. Thanks to this, stateful SDN by design addresses scalability-related issues that caused by SDN centralization of control logic [4]. Here, a prominent attack example is the *control plane saturation attack*. This attack aims at incapacitating the control plane, which is mainly a consequence of the high degree of interaction required between the data plane and the control plane. An attacker can launch a control plane saturation attack by generating a huge amount of different flows (with spoofed IP addresses); this will cause data plane switches to forward many `packetIn` requests to the controller, possibly saturating either the communication link, or controller's internal resources. While researchers proposed [56]–[58] data plane solutions to tackle this problem, mitigating control plane saturation remains an open challenge in legacy SDN environments. By design, stateful SDN is more resilient to this threat, as its stateful nature limits the required communication between the data plane and the control plane.

3) *Data plane vulnerabilities*

At the data plane level, stateful SDN naturally mitigates two main traditional SDN vulnerabilities: (i) flow information leakage, via timing side channels [59]; and (ii) in many cases, the exhaustible TCAM dedicated to the flow tables. Vulnerability (i) might be used by an attacker to learn the configuration of the network. This would be possible due to the time overhead introduced by the control plane in order to install a rule, e.g., when a new incoming flow cannot be matched to any rule in switch's flow table [60]. However, in stateful SDN, this may not be the case: the control plane can program the stateful switch to take autonomous decisions on how to handle incoming flows, without the need to contact the controller. This intuitively reduces the attack surface from an attacker that rely on timing information. For the same reason, those attacks that leverage vulnerability (ii) are implicitly mitigated by the way stateful SDN switches are programmed, and hence behave; indeed, the use of stateful data plane functionalities can reduce the required changes to the flow table. Note that, while vulnerability (ii) is in nature similar to the one we identified and described in Section III-A1 (i.e., unbounded flow state memory allocation), the latter refers *specifically* to the data structure used by a stateful SDN switch to maintain state at the data plane, which may be different from the flow table used in traditional SDN. Finally, we remark that the vulnerability in Section III-A2 can also affect the traditional SDN. As an example, consider again the control plane saturation attack, where an attacker has a way to force the communication between the SDN switch, and the control plane. The attack has an impact not only on the control plane, but also on the switch itself: the switch must send all its `packetIn` messages through a TLS encrypted channel, and, as a consequence, needs to heavily use cryptographic operations. The use of cryptography adds a non negligible processing overhead on the attacked switch, making it vulnerable to DoS attacks.

IV. Discussion and Future Work Directions

In Section III, we discussed major vulnerabilities and possible attacks to the stateful SDN data plane. Table IV summarizes the described existing security issues classified based on different categories of stateful schemes introduced in Section II-C. The "*Vulnerabilities*" column contains the vulnerabilities (introduced in Section III-A) identified per each stateful SDN data plane category. The "*Vulnerability Motivations*" column describes the reason behind each emerging vulnerability. The "*Potential Attacks*" column shows which of the attacks explained in Section III-B applies to the corresponding category. Finally, the "*Remarks*" column refers to one or more of the guidelines or critical points

TABLE IV: Security comparison of the Stateful SDN schemes

Stateful SDN Schemes Categories	Vulnerability	Vulnerability Motivations	Potential Attack	Remarks
Platforms (Section II-C1)	Unbounded flow state memory allocation (Section III-A1)	- Applications allocate a large amount of memory (to hold data structures for application-specific states). This may be due to inappropriate implementation of the state tables, bogus memory management (unallocated memory), or by the nature of the stateful application itself.	Switch Memory Saturation (Section III-B1)	Programmable monitoring (Section IV-B) Security-aware development (Section IV-E)
	Triggerable CPU intensive operations (Section III-A2)	- Applications may expose functionalities (either voluntarily or unintentionally) involving CPU intensive operations.	CPU Exhaustion Attack (Section III-B3)	Programmable monitoring (Section IV-B) Security-aware development (Section IV-E)
	Lack of authentication mechanisms in the data plane (Section III-A3)	- Lack of authentication and integrity check methods for the received packets in switch-to-switch communication, - Event based independent decision making by the switch.	State Inconsistency Attack (Section III-B2)	Secure in-band signaling and verifiable state transition (Section IV-C) Programmable monitoring (Security IV-B)
	Lack of a central data plane state management (Section III-A4)	- As stateful SDN "moves" flow state down to the data plane, this may create inconsistent state between switches and the control plane, and also among the switches.	State Inconsistency Attack (Section III-B2)	State consistency check (Section IV-D) Verifiable state transition (Section IV-C)
Compilers, languages, and frameworks (Section II-C2)	Unbounded flow state memory allocation (Section III-A1)	- Lack of memory limiting and management, - Event-driven resources usage.	Switch Memory Saturation (Section III-B1)	Security-aware development (Section IV-E)
	Lack of a central data plane state management (Section III-A4)	- Inappropriate definition of access policies to the state storage.	State Inconsistency Attack (Section III-B2)	Security-aware development (Section IV-E) State consistency check (Section IV-D)
Applications and use cases (Section II-D)	Unbounded flow state memory allocation (Section III-A1)	- Applications do not impose limit to the allocated state storage space, - Applications do not define state removal conditions.	Switch Memory Saturation (Section III-B1)	Programmable monitoring (Section IV-B) Security-aware development (Section IV-E)
	Triggerable CPU intensive operations (Section III-A2)	- Lack of control on "triggerable" CPU intensive operations, such as cryptographic operations due to switch-to-controller communication over a SSL/TLS channel.	CPU Exhaustion Attack (Section III-B3)	Security-aware Development (Section IV-E)
	Lack of a central data plane state management (Section III-A4)	- Applications cannot rely on a centralized network state management.	State Inconsistency Attack (Section III-B2)	State consistency check (Section IV-D)

that should be taken into account, which we explain in this section. Although the goal of this paper is limited to raise the awareness about the security issues in stateful SDN data plane, for completeness, in this section we provide useful practical guidelines to be applied for a security-aware design of stateful SDN, and draw some possible future research directions.

A. Analogies with other networking contexts

While, of course, stateful SDN is a newly emerging trend, at least from what concerns security challenges it shares similarities with work done in traditional networking systems, which also *rely on switch states dynamically set and updated by data plane events*, e.g., arrival of specific packet types, etc. In other words, the security issues of stateful SDN data plane that we opened in this paper may be considered, to some extent, as a "superset" of the security challenges that exist in already established networking contexts, such as Ethernet [33] or, more recently, Information-Centric Networking (ICN) [61]. In the following, we clarify this matter by briefly discussing common security issues affecting such two example contexts[7], as well as relevant (and already existing) mitigation techniques which can thus inspire adaptation to the more general stateful SDN scenario.

1) Ethernet

Ethernet switches have a *dynamically updated* Forwarding Table which maps MAC addresses to the switch ports where relevant frames must be forwarded. As very well known, Ethernet frames are forwarded based on their *destination* MAC address, but at the same time the Forwarding Table is dynamically set up and updated through a dynamic *Learning* process. Upon a packet arrival, the table is updated by adding (or modifying) an entry which maps the *source* MAC address to the switch's *input* port, i.e., the port from which the packet was received. This table, frequently referred to as Content Addressable Memory (CAM) owing to the way it is customarily implemented in hardware (HW), may also store other information, such as time-out or Virtual LAN

[7]Of course many other traditional networking contexts which leverage state information dynamically updated via data plane events can be considered – our discussion is indeed not nearly meant to be complete, but is meant to encourage the reader not only to focus on new solutions, but also to learn from the past and look for already existing solutions in different (but still somewhat related) networking areas.

identifier. The similarity between the structure of the CAM table and the notion of *state table* used in stateful SDN is evident, as they both perform automatic packet-driven updates inside the switch. Although Ethernet has several well established desirable features, such as flexibility, scalability, and simplicity [62], it is prone to several DoS and spoofing attacks, such as MAC flooding, ARP poisoning, port stealing, and resource exhaustion attacks (for more information please refer to the Ethernet security survey in [33]). Noteworthy, among the possible mitigation techniques, by far the most practical and widely established is *Port Security*, which is in essence a local monitoring technique running *on top* of the Ethernet Switch operation. It monitors the MAC addresses on a switch port so as to limit the maximum number of MAC addresses that can be learned by the same port, and protect the Local Area Network against MAC spoofing and Forwarding Table overflow attacks [33], [63]. The Ethernet case suggests that switch-level (local) monitoring is a very first candidate mitigation approach also for the more general case of stateful SDN – we refer the reader to the more detailed discussion carried out in the next Section IV-B.

2) Information-Centric Networking

ICN is an emerging networking paradigm that replaces the traditional IP-centric Internet communication with a data-centric publish/subscribe one, at the network layer. Among the various ICN instantiations, the most representative and established ones are Content-Centric Networking (CCN) [64], and Named-Data Networking (NDN) [65]; CCN and NDN differ in design and implementation, but share the same core principles. In ICN contents are produced by *producers*, and requested by *consumers*; a content is delivered to a consumer in *data packets*, each of which is requested via an *interest packet*. Each data packet is uniquely addressed by a hierarchical name (e.g., `com/cnn/news`), which is specified inside interest packets, and used for routing. ICN routers maintain *state* about incoming requests (i.e., interests) in an internal data structure called *Pending Interest Table* (PIT). Using this table, an ICN router records the interface I from which an interest has been received, and the content name N carried by the interest packet. Routers use another table named *Forwarding Interest Base* (FIB) to route interest packets according to the name prefix and corresponding interface. Each data packet is routed back to the consumer through the reverse path followed by the corresponding interest packet: when a data packet is received by a router, it checks for a matching entry inside the PIT; if a matching is found, the router forwards the packet back from the interface(s) indicated in such entry, and removes the entry; otherwise, it simply discards the data packet. It is easy to see the similarities between the PIT data structure used by ICN routers, and the *state table* used in stateful SDN. Indeed, researchers showed how such PIT could be implemented on top of the stateful SDN data plane [66]. While ICN has several benefits compared to the IP-based network, such as low latency, and improved scalability and mobility [67], [68], it introduces new security challenges. In particular, the required PIT *state* management in ICN makes it vulnerable against several DoS attacks, e.g., resource exhaustion or state inconsistency [68]–[73]. The security community has already considered these security issues and addressed some of them. As an example, the work in [72] proposes a mitigation method against a type of DoS attack that targets the PIT inside routers. Finally, many approaches proposed in the ICN field also rely on local monitoring-based solutions [68], such as using local metrics on a router, e.g., PIT usage [72], interest traceback [71], and limiting the request rate [74].

B. Towards programmable monitoring

As the previous discussion on the security of Ethernet switches and ICN routers has anticipated, the reliance on local monitoring techniques appears to be a very first and natural defensive strategy for mitigating stateful SDN security threats. The general idea is to design monitoring techniques (and associated mitigation strategies) which locally run on the network nodes and which are devised to identify anomalous traffic patterns and occurrence of attacks to the stateful SDN data plane. Local monitoring techniques and relevant feature extraction and statistics collection may then further feed more sophisticated network-wide monitoring frameworks involving the SDN controller and its complete view of the network state.

However, the key difference between stateful SDN and traditional networking scenarios (say Ethernet or ICN switches) is that in these latter cases the network node operation is *known a priori*, and hence the monitoring techniques can be specifically designed to detect *specific* anomalies and attacks relevant to the specialized network node operation. Conversely, in the case of stateful SDN, the node behavior is not *a priori specified*, but it is arbitrarily programmed (e.g., via finite state machines [18], P4 programs [23], or other data plane programming abstractions discussed in section II-C). In other words, the monitoring problem space in stateful SDN data plane is significantly wider than the corresponding one widely addressed in traditional (specialized) network node operations.

How to detect threats and anomalies for fully programmable and repurposable network nodes, where whole behavior is not a priori known? We posit that a compelling solution consists in devising complementary techniques and methodologies for permitting not only programmability of the network nodes' operation, but also of their *monitoring*. The research challenge is thus the design of abstractions and platforms for programmable monitoring tasks, so as to permit the network application developer to *further* design a tailored (application-specific) monitoring strategy to deploy along with the node operation – in essence associate to the SDN paradigm a corresponding *Software-Defined Monitoring* paradigm which may provide the application designer with easy to use modules, primitives, and languages, to design and deploy custom monitoring algorithms on top of custom-programmed network nodes. An initial contribution in this very interesting (and, we believe, very important) research direction can be found in [28], where the authors have described a programmable monitoring method based on extended finite

state machines (as platform agnostic abstractions) which permits the programmer to formally describe and execute a desired real-time, multi-step, customized local monitoring operation.

Another very interesting and forward-looking evolution of the stateful SDN network nodes for supporting *both* forwarding and monitoring tasks, consists in designing nodes that implement in HW, and at wire-speed, more advanced primitives. These primitives go beyond forwarding, and rather permit to further customize (per-flow) feature extraction and statistics collection. The reader may refer to [21] for a very recent work in this area. This work presents an HW architecture for a stateful programmable network node, which evolves [18] by further supporting programmable registries and relevant arithmetic and logic update functions operating at wire-speed. Another very recent work somewhat related to this goal is [22], where the authors describe advanced HW techniques to embed arithmetic and logic operations in the flow processing pipeline. These additional functions permit the programmer to not only configure a desired network protocol operation or forwarding behavior, but also to program, directly on the same network hardware, custom per-flow statistics and features extraction at wire-speed, for instance (following [21]) average, standard deviation, and exponentially weighted moving average of statistics such as inter-packet arrival time, packet rate, packet length, flow duration, and so on.

These programmable features can either be directly used as *raw* data for tailored monitoring algorithms, or may be exploited by overlay Machine Learning (ML) algorithms [75]–[78] dedicated to traffic analysis and anomaly detection. Note that such algorithms can be run at a (much) lower time scale than wire-speed packet processing, and will benefit of the fact that feature extraction and statistics collection on a per-flow basis and at packet-level speed (usually a very demanding task in terms of computational load) would now be offloaded to the network node HW.

C. Secure in-band signaling and verifiable state transition

Leveraging stateful SDN data plane capabilities, recent proposals provided solutions, e.g., failure recovery or load balancing, that use in-band signaling between switches, e.g., by means of control packets or even ordinary data packets (eventually labeled or marked with control information). However, as discussed in Section III-A and Section III-B, lack of an authentication and integrity verification mechanism for packets exchanged between switches allows an adversary to spoof and/or inject malicious control packets into the network, and may cause DoS attacks.

An obvious research direction for addressing such issues consists in devising trust models for secure inter-switch (control) packets exchange, and the relevant associated cryptographic mechanisms for verifying the integrity and provenance of the signalling packets. Unfortunately, state of the art cryptographic authentication and integrity verification mechanisms may hardly attain the wire-speed packet processing speed that some applications may require (e.g., the stateful failure recovery application described in Section II-D1 and attacked in Section III-B2). For instance, even if highly optimized, an ordinary RSA digital signature verification may take at best tens or even hundreds of microseconds (see, for example, [70]), which rules away any possibility to perform it directly on the switch's fast path, where the processing timescale is in the order of few nano-seconds.

It is true that ultra-lightweight integrity verification mechanisms may be devised (for instance the very recently proposed Walnut signature [79], [80] is claimed to be verifiable in as little as one clock cycle). However, even assuming that suitable mechanisms may be ultimately found, they would come along with two major practical disadvantages. First, their implementation may be costly as it may require dedicated HW to be implemented over the switch's fast path. Second, in cryptography, *novel* constructions (as it seems to be necessarily the case for mechanisms capable to perform at wire speed) are not always advisable. Despite their possible technical merits, acceptance of a novel approach requires time for a thorough scrutiny, and may require multiple revisions along this path (as an example, the lightweight NTRU signature was broken multiple times [81] when initially proposed).

As a possible alternative to fast-path integrity verification, we believe that a potentially promising approach consists in *abandoning* the idea of *directly* verifying a control packet at line rate, but thinking to a state transition triggered by an in-band signaling packet as a "two-phase process", i.e., as a transactional process where the in-band data packet just "starts" a state transition, but the actual commitment of the state transition is made only once a cryptographically verifiable confirmation of the triggering control message is received. In other words, the idea is to take a first (and very fast) intermediate reaction – which depends on the specific application considered – upon reception of the in-band data packet, but wait for a subsequent (and more reliable) "confirmation" before committing the transaction. Note that such "confirmation" may either come directly from the cryptographic verification of the received data packet – mandated to the switch slow path and hence delayed with respect to the original packet arrival time – or via the involvement of the SDN controller.

We remark that the above proposed research idea is just a very first sketch here proposed just to stimulate further research. Its actual application to concrete scenarios requires a careful design and many questions appear to pop up. For instance, should such two-phase approach be applied selectively on specific packets or events? Does it require a tailored per-application design? How to involve the controller? Might the second (commit) phase have the side effect of opening the door to other attack vectors such as CPU exhaustion attacks (see Section III-B3), or control plane saturation attacks [57]? And so on.

Lacking (to the best of our knowledge) more specific related work, we here just limit to mention that *two-phase commit transactional* mechanisms have been recently addressed in

traditional SDN as a solution to ensure fault tolerance and consistent network updates. Ravana [82] and LegoSDN [83] are two examples of recently proposed *transactional* fault tolerant controllers; and the works in [84], [85] are examples of solutions for consistent network updates. In these methods, the SDN controller follows the *commit* or *abort* rule in transactional networking management [86], allowing the conflicting rules (or network status) to *roll back* to the previous consistent status.

D. State consistency check

As explained in Section II, an important feature of the stateful SDN data plane is that the flow state transition inside the switch is based on packet-level events. Therefore, the controller does not have a full control of the flow states inside the switches. Those behaviors might result in having inconsistent local states inside the switches, as well as inconsistency between the switch and the controller. In order to protect the network against such a vulnerability, the stateful data plane needs to have the controller inspection. Even though the controller may not have an active role in enforcing flow states inside the switches (as those can be locally enforced), it still has to retain some form of *loose* control on how those local states evolve. In this regard, some researchers recently proposed, NeSMA, a framework to support network-level state-aware applications [87], in which the switches update the controller upon state transition. However, the situation in which the switch needs to update the controller should be precisely defined, since otherwise the updating procedure may lead to CPU failure (see Section III-B3).

One possible solution to face the state inconsistency issue could be policy enforcing methodology proposed in SNAP [16]. In SNAP, the central controller enforces some access policies, based on which the switches decide where and when to access and modify the state variables. At the same time, SNAP also provides the data plane programmability feature. It should be noted that, since the stateful data plane takes all the decisions based on the locally stored state information in the switch, state inconsistency can be a hazardous issue. Therefore, having a consistent state among all the switches is of importance. Possible solutions to provide state consistency could be: considering consistent switch updates, which is proposed in [48], and centralized state storage, which is proposed in SNAP [16]. However, centralized storage of the states in one switch requires careful ordering of the packets and state transitions, and ensuring that the packets will pass by the correct switch. We believe that an important future research direction in stateful SDN data plane could be seeking solutions to control the state storage and state transition inside the switch.

E. Security-aware development

As it can be seen in Table IV, several vulnerabilities emerge due to inappropriate implementation of state storing memories and access policies. In order to limit the attack surface, beside programmable monitoring, a careful application design and development process is recommended. In order to mitigate attacks that aim at exhausting the internal memory of stateful SDN switches (Section III-A1), developers could: (i) estimate, a priori, the required memory space for the state tables/variables, and (ii) define appropriate conditions for state removal and eviction. Furthermore, applications should carry out only basic (stateful) network functionalities, and rely, when possible, on local state to take decisions. The aim here is to prevent external attacker to either force memory allocation (Section III-A1), or to trigger CPU intensive operations (Section III-A2).

Other than a careful application design and development, stateful application designers could make use of formal verification and static code analysis techniques [88]; a list of open source static code checkers can be found in [89]. As a recent work in this area, the "BEhaviaural BAsed forwarding (BEBA)" European project[8] makes use of the NuSMV symbolic model checker [90] for bugs finding, leveraging the one-to-one mapping between the stateful SDN logic, and finite state machine representation. The usage of formal verification methods could also be helpful in protecting against state inconsistency vulnerability, while it requires further research to become fully practical.

V. Conclusion

Very recently, researches recognized the need for having programmable switches in SDN, and proposed the concept of stateful SDN data plane. Having programmable SDN switches enhances the network performance and flexibility in response to real-time network applications, since it allows the switches to manage dynamic applications faster and more efficiently. We believe that, stateful SDN data plane, although practically competent, is prone to various security attacks. There are several different schemes proposed in the literature, each of which addressing a specific need in bringing stateful SDN data plane into reality (such as different platforms, programming languages or applications). However, none of them has built-in security measures. Therefore, to the best of our knowledge, we are the first to review and analyze the main features of the stateful data plane schemes in the literature, followed by highlighted major security vulnerabilities that are in place due to the intrinsic properties of the stateful schemes.

We believe that our study will draw the community's attention to the security issues of the stateful SDN data plane and hopefully motivates the researchers to propose security solutions for this new networking paradigm. We anticipate new security challenges to be emerged due to the upward trend in stateful data plane development. In this regard, we provided some basic recommendations to be considered in new stateful SDN proposals to cope with the explained vulnerabilities, as well as possible future research directions. Albeit we call security research community to play a proactive role in proposing proper security measures in this context.

[8] http://www.beba-project.eu/public_deliverables/BEBA_D4.3.pdf

VI. Acknowledgements

Tooska Dargahi and Giuseppe Bianchi are partially supported by the EU H2020 Projects BEBA (grant number 644122), and Superfluidity (grant number 671566). Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980). This work is also partially supported by the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061), the EU-India REACH Project (agreement ICI+/2014/342-896), "Physical-Layer Security for Wireless Communication", and "Content Centric Networking: Security and Privacy Issues" funded by the University of Padua.This work is partially supported by the grant n. 2017-166478 (3696) from Cisco University Research Program Fund and Silicon Valley Community Foundation.

References

[1] 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 Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[2] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and OpenFlow: from concept to implementation," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2181–2206, 2014.

[3] S. T. Ali, V. Sivaraman, A. Radford, and S. Jha, "A survey of securing networks using software defined networking," *IEEE Transactions on Reliability*, vol. 64, no. 3, pp. 1086 – 1097, 2015.

[4] S. Scott-Hayward, S. Natarajan, and S. Sezer, "A Survey of Security in Software Defined Networks," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 623–654, 2015.

[5] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov, "Security in software defined networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2317–2346, 2015.

[6] I. Alsmadi and D. Xu, "Security of software defined networks: A survey," *Computers & Security*, vol. 53, pp. 79–108, 2015.

[7] Q. Yan, R. Yu, Q. Gong, and J. Li, "Software-defined networking (SDN) and distributed denial of service (DDoS) attacks in cloud computing environments: A survey, some research issues, and challenges," *IEEE Communications Surveys & Tutorials*, vol. PP, no. 99, 2015.

[8] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: an intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[9] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[10] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: Dynamic access control for enterprise networks," in *1st ACM Workshop on Research on Enterprise Networking (WREN09)*, 2009.

[11] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ACM SIGPLAN Notices*, vol. 46, no. 9, 2011, pp. 279–291.

[12] A. Voellmy, H. Kim, and N. Feamster, "Procera: a language for high-level reactive network control," in *Proc. of the 1st workshop on Hot topics in software defined networks*, 2012, pp. 43–48.

[13] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software-Defined Networks," in *USENIX NSDI*, 2013, pp. 1–13.

[14] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi, "Tierless programming and reasoning for software-defined networks," *USENIX NSDI*, 2014.

[15] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control," in *Proc. of the 12th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'15, 2015, pp. 59–72.

[16] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "SNAP: Stateful network-wide abstractions for packet processing," in *Proc. of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM'16. ACM, 2016.

[17] J. Xie, D. Guo, Z. Hu, T. Qu, and P. Lv, "Control plane of software defined networks: A survey," *Computer Communications*, vol. 67, pp. 1–10, 2015.

[18] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: programming platform-independent stateful OpenFlow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014.

[19] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for SDN," in *Proc. of the 3rd workshop on Hot topics in software defined networking*, ser. HotSDN'14. ACM, 2014, pp. 61–66.

[20] S. Zhu, J. Bi, C. Sun, C. Wu, and H. Hu, "SDPA: Enhancing stateful forwarding for software-defined networking," in *Proc. of the 23rd International Conference on Network Protocols*, ser. ICNP'15. IEEE, 2015, pp. 10–13.

[21] G. Bianchi, M. Bonola, S. Pontarelli, D. Sanvito, A. Capone, and C. Cascone, "Open packet processor: a programmable architecture for wire speed platform-independent stateful in-network processing," *CoRR*, vol. abs/1605.01977, 2016. [Online]. Available: http://arxiv.org/abs/1605.01977

[22] A. Sivaraman, M. Budiu, A. Cheung, C. Kim, S. Licking, G. Varghese, H. Balakrishnan, M. Alizadeh, and N. McKeown, "Packet transactions: High-level programming for line-rate switches," in *Proc. of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM'16. ACM, 2016.

[23] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[24] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *USENIX NSDI*, 2015.

[25] The P4 language Consortium, "The P4 Language Specification, version 1.0.2," March 2015.

[26] W. Han, H. Hu, Z. Zhao, A. Doupé, G.-J. Ahn, K.-C. Wang, and J. Deng, "State-aware network access management for software-defined networks," in *Proc. of the 21st ACM on Symposium on Access Control Models and Technologies*, ser. SACMAT'16. ACM, 2016, pp. 1–11.

[27] 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 *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 99–110.

[28] M. Bonola, G. Bianchi, G. Picierro, S. Pontarelli, and M. Monaci, "StreaMon: a data-plane programming abstraction for software-defined stream monitoring," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, 2015.

[29] C. Kim, P. Bhide, E. Doe, H. Holbrook, A. Ghanwani, D. Daly, M. Hira, and B. Davie, "In-band Network Telemetry (INT)," http://p4.org/wp-content/uploads/fixed/INT/INT-current-spec.pdf, Tech. Rep., Sep. 2015.

[30] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "HULA: Scalable load balancing using programmable data planes," in *Proc. of the 2nd ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR'16. ACM, 2016.

[31] A. Capone, C. Cascone, A. Q. T. Nguyen, and B. Sansò, "Detour planning for fast and reliable failure recovery in SDN with OpenState," in *Proc. of the 11th International Conference on the Design of Reliable Communication Networks*, ser. DRCN'15. IEEE, 2015, pp. 25–32.

[32] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "Netpaxos: Consensus at network speed," in *Proc. of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR'15. ACM, 2015, p. 5.

[33] T. Kiravuo, M. Sarela, and J. Manner, "A survey of Ethernet LAN security," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 3, pp. 1477–1491, 2013.

[34] S. M. Bellovin, "Security problems in the TCP/IP protocol suite," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 2, pp. 32–48, 1989.

[35] K. M. Lepak and M. H. Lipasti, "Silent stores for free," in *Proc. of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-33. IEEE, 2000, pp. 22–31.

[36] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A Survey on Software-Defined Networking," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 27–51, 2015.

[37] Y. Jarraya, T. Madi, and M. Debbabi, "A survey and a layered taxonomy of software-defined networking," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 1955–1980, 2014.

[38] B. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[39] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[40] H. Farhady, H. Lee, and A. Nakao, "Software-defined networking: A survey," *Computer Networks*, vol. 81, pp. 79–95, 2015.

[41] O. N. Fundation, "Software-defined networking: The new norm for networks," *ONF White Paper*, 2012.

[42] S. Shenker, M. Casado, T. Koponen, N. McKeown *et al.*, "The future of networking, and the past of protocols," *Keynote slides, Open Networking Summit*, vol. 20, 2011.

[43] OpenFlow Switch Specification, v.1.0.0, Dec 2009. http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf.

[44] The Benefits of Multiple Flow Tables and TTPs, version 1.0, ONF TR-510 Technical Report, February 2015. https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_Multiple_Flow_Tables_and_TTPs.pdf.

[45] OpenFlow Switch Specification, v.1.3.4, March 2014. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.4.pdf.

[46] OpenFlow Switch Specification, v.1.5.1, ONF TS-025, March 2015. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf.

[47] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. of the 2nd ACM SIGCOMM workshop on Hot topics in software defined networking*, ser. HotSDN'13. ACM, 2013, pp. 127–132.

[48] J. McClurg, H. Hojjat, N. Foster, and P. Cerny, "Specification and compilation of event-driven SDN programs," *arXiv preprint arXiv:1507.07049*, 2015.

[49] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design principles for packet parsers," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS'13. IEEE, 2013, pp. 13–24.

[50] D. Hancock and J. van der Merwe, "HyPer4: Using P4 to Virtualize the Programmable Data Plane," in *Proc. of the 12th International on Conference on emerging Networking EXperiments and Technologies*, ser. CoNEXT'16. ACM, 2016, pp. 35–49.

[51] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic ddos defense," in *Proc. of the 24th USENIX Security Symposium*, ser. USENIX Security'15, 2015, pp. 817–832.

[52] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: state distribution trade-offs in Software Defined Networks," in *Proc. of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 1–6.

[53] P. Perešíni, M. Kuzniar, and D. Kostić, "Rule-level data plane monitoring with monocle," in *Proc. of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, 2015, pp. 595–596.

[54] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," in *Passive and Active Measurement*. Springer, 2015, pp. 347–359.

[55] M. Krzywinski. (2004) A critique of port knocking - author's response. http://www.portknocking.org/view/about/critique.

[56] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: Scalable and vigilant switch flow management in software-defined networks," in *Proc. of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 413–424.

[57] M. Ambrosin, M. Conti, F. De Gaspari, and R. Poovendran, "Lineswitch: Efficiently managing switch flow in software-defined networking while effectively tackling dos attacks," in *Proc. of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 2015, pp. 639–644.

[58] ——, "Lineswitch: Tackling control plane saturation attacks in software-defined networking," *IEEE/ACM Transactions on Networking*, 2016.

[59] M. Conti, F. De Gaspari, and L. V. Mancini, "Know your enemy: Stealth configuration-information gathering in sdn," *arXiv preprint arXiv:1608.04766*, 2016.

[60] R. Klöti, V. Kotronis, and P. Smith, "Openflow: A security analysis," in *Proc. of the 21st IEEE International Conference on Network Protocols*, ser. ICNP'13. IEEE, 2013, pp. 1–6.

[61] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman, "A survey of information-centric networking," *IEEE Communications Magazine*, vol. 50, no. 7, pp. 26–36, 2012.

[62] L. Zier, W. Fischer, and F. Brockners, "Ethernet-based public communication services: challenge and opportunity," *IEEE Communications Magazine*, vol. 42, no. 3, pp. 88–95, 2004.

[63] I. Dubrawsky, "White paper: Safe layer 2 security in-depthâĂŤversion 2," Cisco Systems, Tech. Rep., 2004.

[64] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proc. of the 5th international conference on Emerging networking experiments and technologies*. ACM, 2009, pp. 1–12.

[65] Named Data Networking project (NDN). www.named-data.org.

[66] G. Bianchi, M. Bonola, and S. Pontarelli, "On the feasibility of "breadcrumb" trails within openflow switches," in *2016 European Conference on Networks and Communications*, ser. EuCNC'16, 2016, pp. 122–127.

[67] S. K. Fayazbakhsh, Y. Lin, A. Tootoonchian, A. Ghodsi, T. Koponen, B. Maggs, K. Ng, V. Sekar, and S. Shenker, "Less pain, most of the gain: Incrementally deployable icn," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 147–158.

[68] E. G. AbdAllah, H. S. Hassanein, and M. Zulkernine, "A survey of security attacks in information-centric networking," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 3, pp. 1441–1454, 2015.

[69] M. Wählisch, T. C. Schmidt, and M. Vahlenkamp, "Lessons from the past: Why data-driven states harm future information-centric networking," in *Proc. of the IFIP Networking Conference*. IEEE, 2013, pp. 1–9.

[70] P. Gasti, G. Tsudik, E. Uzun, and L. Zhang, "Dos and ddos in named data networking," in *Proc. of the 22nd International Conference on Computer Communication and Networks*, ser. ICCCN'13. IEEE, 2013, pp. 1–7.

[71] H. Dai, Y. Wang, J. Fan, and B. Liu, "Mitigate ddos attacks in NDN by interest traceback," in *Proc. of the IEEE Conference on Computer Communications Workshops*, ser. INFOCOM WKSHPS'13. IEEE, 2013, pp. 381–386.

[72] A. Compagno, M. Conti, P. Gasti, and G. Tsudik, "Poseidon: Mitigating interest flooding DDoS attacks in named data networking," in *Proc. of the 38th Conference on Local Computer Networks*, ser. LCN'13. IEEE, 2013, pp. 630–638.

[73] A. Compagno, M. Conti, C. Ghali, and G. Tsudik, "To nack or not to nack? negative acknowledgments in information-centric networking," in *Proc. of the 24th International Conference on Computer Communication and Networks*, ser. ICCCN'15. IEEE, 2015, pp. 1–10.

[74] A. Afanasyev, P. Mahadevan, I. Moiseenko, E. Uzun, and L. Zhang, "Interest flooding attack and countermeasures in named data networking," in *IFIP Networking Conference*. IEEE, 2013, pp. 1–9.

[75] M. V. Mahoney and P. K. Chan, "Learning rules for anomaly detection of hostile network traffic," in *Proc. of the 3rd IEEE International Conference on Data Mining*, ser. ICDM '03. IEEE, 2003, pp. 601–.

[76] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[77] A. S. da Silva, J. A. Wickboldt, L. Z. Granville, and A. Schaeffer-Filho, "Atlantic: A framework for anomaly traffic detection, classification, and mitigation in sdn," in *Proc. of the 15th IEEE/IFIP Network Operations and Management Symposium*, ser. NOMS'16. IEEE, 2016, pp. 27–35.

[78] A. S. da Silva, C. C. Machado, R. V. Bisol, L. Z. Granville, and A. Schaeffer-Filho, "Identification and selection of flow features for accurate traffic classification in SDN," in *Proc. of the 14th International Symposium on Network Computing and Applications*, ser. NCA'15. IEEE, 2015, pp. 134–141.

[79] I. ANSHEL, D. ATKINS, D. GOLDFELD, and P. E. GUNNELLS, "Post quantum group theoretic cryptography," SecureRF Corporation, Tech. Rep., 2016.

[80] D. ATKINS, I. ANSHEL, D. GOLDFELD, and P. GUNNELLS, "Walnut digital signature algorithm: A lightweight, quantum-resistant signature scheme for use in passive, low-power, and iot devices," in *NIST Lightweight Cryptography Workshop*. NIST, 2016.

[81] C. Gentry and M. Szydlo, "Cryptanalysis of the revised ntru signature scheme," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2002, pp. 299–320.

[82] N. Katta, H. Zhang, M. Freedman, and J. Rexford, "Ravana: Controller fault-tolerance in software-defined networking," in *Proc. of the 1st ACM*

SIGCOMM Symposium on Software Defined Networking Research, ser. SOSR. ACM, 2015, p. 4.
[83] B. Chandrasekaran, B. Tschaen, and T. Benson, "Isolating and tolerating SDN application failures with LegoSDN," p. 4, 2016.
[84] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, ser. SIGCOMM'12. ACM, 2012, pp. 323–334.
[85] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "A distributed and robust sdn control plane for transactional network updates," in *Proc. of the IEEE conference on computer communications*, ser. INFOCOM'15. IEEE, 2015, pp. 190–198.
[86] ——, "Software transactional networking: Concurrent and consistent policy composition," in *Proc. of the 2nd ACM SIGCOMM workshop on Hot topics in software defined networking*, ser. HotSDN'13. ACM, 2013, pp. 1–6.
[87] C. Sun, J. Bi, H. Hu, and Z. Zheng, "Nesma: Enabling network-level state-aware applications in sdn," in *Proc. of the 24th International Conference on Network Protocols*, ser. ICNP'16. IEEE, 2016, pp. 1–7.
[88] P. Louridas, "Static code analysis," *IEEE Software*, vol. 23, no. 4, pp. 58–61, 2006.
[89] Source Code Analysis Tools. https://www.owasp.org/index.php/Source_Code_Analysis_Tools.
[90] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *International Conference on Computer Aided Verification*. Springer, 2002, pp. 359–364.

APPENDIX A
EXPERIMENTAL ATTACKS ASSESSMENT

In this section, we present an experimental assessment of the attack examples introduced in Section III-B.

In our experiments, we emulated an SDN switch using Mininet, with OpenFlow (Version 1.3). We implemented our use cases adopting the open source implementation of OpenState (which is based on OpenFlow), and its existing applications[9]. Given this setup, all OpenState applications run on top of the `ofdatapath` process that is the userspace implementation of the OpenFlow datapath. We run all our experiments on a desktop machine equipped with a Core i7 CPU with 2 cores and 16 GB of RAM, running Mininet VM with 4 virtualized cores and 512 MB of RAM. It is worth noting that the capacity of RAM in a real SDN switch is in the order of tens of MB. Therefore, when possible, we report our results in percentage, to provide a platform independent presentation of the attacks, and allow the reader to easily "mentally scale" to different real-world switch settings. In what follows, we refer to *OS* for an implementation using OpenState switches, and *OF* when considering normal OpenFlow switches. Moreover, we consider the data structure that stores the flow states to be a *state table*. We stress that the same reasoning behind the attacks we explain in this section, holds for any other data structures as well, e.g., variables.

A. Port Knocking Attack

We now show the feasibility and effectiveness of the *memory saturation attack* on the *port knocking* application, introduced in Section III-B1a. In our experimental setup, we assume an *informed attacker* $\mathcal{A}$ who is aware of the first port number (i.e., 3306) of the sequence. $\mathcal{A}$ aims to flood the memory of a switch, s_1, running the port knocking application on OpenState. To this end, $\mathcal{A}$ floods port 3306 with an attack rate of 205 Mbps, using several spoofed source IP addresses. The switch s_1 is configured with a timeout field of 5 seconds: if the switch does not receive the second packet from the same flow "knocking" the next port, s_1 will reset the state of the flow and remove flow's state from the state table.

Figure 13 presents the results of our evaluation. As we can see in this figure, the memory used by the `ofdatapath` process, i.e., the main process of the OpenState implementation in use, increases rapidly over time, up to almost 70% of the whole switch memory space after 34 sec, on average. This leads to a crash of the `ofdatapath` process. The same situation holds for the switch CPU usage. We recall that the whole RAM space of the Mininet VM is 512 MB.

Figure 14 shows the number of stored states in the state table of the switch during the experiment, before the crash. As we can see, the number of stored states remains *constant* after reaching $\approx 200k$ flows; this is due to the 5 second timeout the switch waits before removing the state form its table.

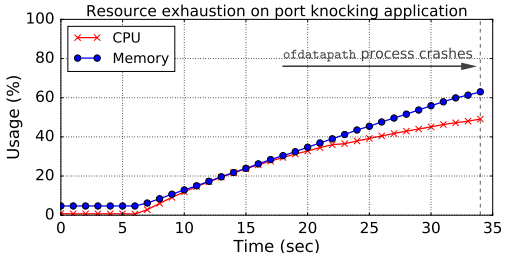

Fig. 13: Percentage of switch resources occupied during our resource exhaustion attack on the port knocking application

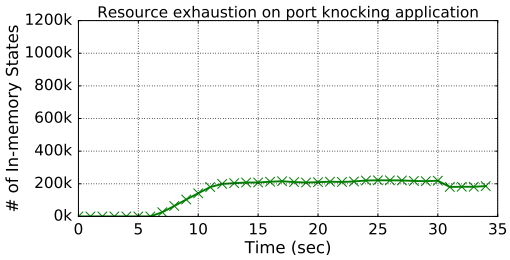

Fig. 14: Number of "in-memory" states for the switch during resource exhaustion attacks on the port knocking application

An interesting observation from this experiment is that, even though the (maximum) number of stored states remains approximately constant (Figure 14), the memory usage of the switch keeps growing over time (Figure 13). We suspect that this trend derives from an inefficient memory management of the OpenState's underlying OFSoftswitch platform's implementation, which causes memory leak.

B. Attack the UDP Flooding Mitigation

Here, we present the results obtained from the *UDP Flooding Mitigation attack* implementation introduced in

[9]https://github.com/OpenState-SDN/

Section III-B1b, showing the feasibility of memory saturation attack in this scenario. In this experiment, we consider an attacker $\mathcal{A}$ that floods UDP connections through a large number of spoofed source IP addresses. The attacker simply opens a new UDP connection per each source IP address, and this way, forces the switch to consider a new state for each UDP connection inside the state table. Figure 15 shows our experimental results of the implemented attack.

From Figure 15 we can observe that, similar to the port knocking flooding attack example, the memory required by the `ofdatapath` process increases swiftly as well, and captures more than 60% of the memory in approximately 34 sec, which causes the `ofdatapath` process to crash. Figure 16 illustrates the number of states stored in the state table inside the switch. It is worth noting that, different from the port knocking attack scenario (Figure 14), with this attack the number of stored states in the state table grows linearly. This situation holds even when a timeout for states removal from the state table is set. The reason behind this behavior is that, by sending packets in constant/dynamic intervals from each of the already used source IP addresses, $\mathcal{A}$ forces the switch to perform state transition for the corresponding IP address and maintain its new state. In this way, due to the large number of imposed states in the state table, the `ofdatapath` process saturates the switch memory, which consequently enables $\mathcal{A}$ to exhaust the memory of the switch. We recall that the whole RAM memory for the Mininet VM is of 512 MB.

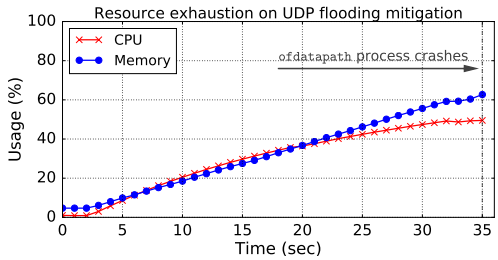

Fig. 15: Percentage of switch resources captured during our resource exhaustion attack on the UDP flooding mitigation application

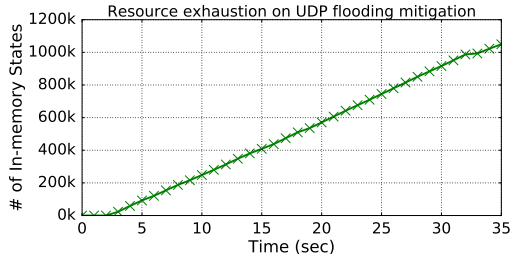

Fig. 16: Number of "in-memory" states for the switch during our resource exhaustion attack on the UDP flooding mitigation application

C. State Inconsistency Attack

To illustrate the feasibility of the State Inconsistency Attack we introduced in Section III-B2, we implemented the failure recovery application in [31] using OpenState. The application uses an identifier to represent a failure between each pair of nodes X and Y. Such identifier is a progressive sequence of numbers starting from 1, which is used as: (1) an MPLS (Multiprotocol Label Switching) tag, to announce the failure; and (2) a flow state value, to show that the corresponding flow has faced a failure in a normal forwarding path. MPLS tags and state values are shifted by 16, as the values between 0 and 15 are reserved. Therefore, identifier 16 denotes *"no failure"*, while failure events are numbered starting from 17. Given that, if an attacker can forge a packet having the correct MPLS tag (i.e., tag 17), she will be able to pretend that there is a failure in the forwarding path and make the switch transit from a normal state, to failure state. This makes the switch to believe that a re-routing of the traffic is required.

Our evaluation considered the network setting in Figure 17. In this simple test scenario, host $\mathcal{H}_1$ (with IP address 10.0.0.1) sends packets to the host $\mathcal{H}_6$ (with IP address 10.0.0.6) through the following path:

$$\mathcal{H}_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow \mathcal{H}_6.$$

On the other side, the host $\mathcal{H}_6$ replies to $\mathcal{H}_1$ through a different path: $\mathcal{H}_6 \rightarrow S_4 \rightarrow S_5 \rightarrow S_2 \rightarrow \mathcal{H}_1$. Note that, this is the original configuration for the failure recovery application implemented in OpenState [31]. Figure 19 shows the execution of our prototype in absence of attack, i.e., with normal behavior of switches, considering default configuration of the failure recovery application. As it can be seen in Figure 19(a), in an attack-free situation, since there is no failure in the network, the state table of the switches considered to be *empty*, i.e., just maintaining a *Default* state. Figure 19(b) shows the normal traffic from the source address 10.0.0.1 to the destination address 10.0.0.6 through the *eth2* interface of the S_3. While, Figure 19(c) illustrates the response path from 10.0.0.6 to 10.0.0.1 via the *eth1* interface of the S_5.

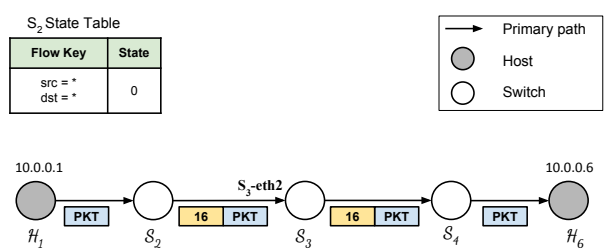

Fig. 17: Network setting for our failure recovery experiment.

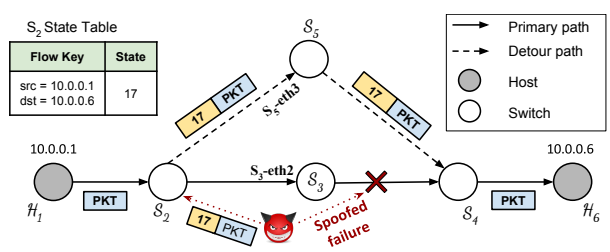

Fig. 18: Network setting for our re-routing attack experiment.

In order to demonstrate the *re-routing attack*, we consider the attack scenario depicted in Figure 18. An attacker $\mathcal{A}$

sends a spoofed MPLS packet labeled as 17 to the switch S_2 (see Figure 20(a)). This way, $\mathcal{A}$ pretends that there is a link failure along the path and makes S_2 to perform re-routing of the corresponding flow traffic through the recovery path. In such a situation, upon reception of the spoofed MPLS, S_2 inserts a new record in its state table for the flow sourced from 10.0.0.1 destined to 10.0.0.6, as shown in Figure 20(d). As it is depicted in the figure, S_2 defines a new state, i.e., state 17, for the corresponding flow in the table. Then, due to the current state, S_2 re-routes the other incoming packets for the corresponding flow through switch S_5 (Figure 20(c)), instead of S_3 (Figure 20(b)). As we can see from Figure 20(b), switch S_3 in the regular path only receives spoofed packets, while all the other traffic will be received by the switch in the re-route path, i.e., switch S_5.

```
ENDING (xid=0xF0FF00F0):
stat_req{exp_type="state", table="all", match=oxm{all m
atch}}

RECEIVED (xid=0xF0FF00F0):
stat_repl{exp_type="state", stats=[

TABLE = 0

{table="0", key={eth_src="*", eth_dst="*"}, state="0"}

]}
```

(a)

```
> tcpdump -i s3-eth2
tcpdump: WARNING: s3-eth2: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for f
ull protocol decode
listening on s3-eth2, link-type EN10MB (Ethernet), capt
ure size 65535 bytes
02:52:45.261540 MPLS (label 16, exp 0, [S], ttl 64) IP
10.0.0.1 > 10.0.0.6: ICMP echo request, id 6693, seq 1,
 length 64
02:52:50.267321 MPLS (label 16, exp 0, [S], ttl 64) IP
10.0.0.1 > 10.0.0.6: ICMP echo request, id 6693, seq 2,
 length 64
02:52:55.266534 MPLS (label 16, exp 0, [S], ttl 64) IP
10.0.0.1 > 10.0.0.6: ICMP echo request, id 6693, seq 3,
 length 64
02:53:00.273345 MPLS (label 16, exp 0, [S], ttl 64) IP
10.0.0.1 > 10.0.0.6: ICMP echo request, id 6693, seq 4,
 length 64
```

(b)

```
> tcpdump -i s5-eth1
tcpdump: WARNING: s5-eth1: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for f
ull protocol decode
listening on s5-eth1, link-type EN10MB (Ethernet), capt
ure size 65535 bytes
02:52:45.262246 MPLS (label 16, exp 0, [S], ttl 64) IP
10.0.0.6 > 10.0.0.1: ICMP echo reply, id 6693, seq 1, l
ength 64
02:52:50.267732 MPLS (label 16, exp 0, [S], ttl 64) IP
10.0.0.6 > 10.0.0.1: ICMP echo reply, id 6693, seq 2, l
ength 64
02:52:55.266820 MPLS (label 16, exp 0, [S], ttl 64) IP
10.0.0.6 > 10.0.0.1: ICMP echo reply, id 6693, seq 3, l
ength 64
02:53:00.274078 MPLS (label 16, exp 0, [S], ttl 64) IP
10.0.0.6 > 10.0.0.1: ICMP echo reply, id 6693, seq 4, l
ength 64
```

(c)

Fig. 19: Normal behavior of the network configuration for failure recovery application on OpenState: (a) State table of switch S_2 when there is no link failure reported in the network; (b) Traffic on the forward path goes through the switch S_3 in case of no link failure. Note that MPLS label is 16, meaning there is no link failure; (c) Traffic on the reverse path goes through the switch S_5 when there is no link failure.

```
> tcpdump -r spoof17.pcap
reading from file spoof17.pcap, link-type EN10MB (Ether
net)
06:30:20.048810 MPLS (label 17, exp 0, [S], ttl 64) IP
10.0.0.1 > 10.0.0.6: ICMP echo request, id 19820, seq 3
, length 64
```

(a)

```
> tcpdump -i s3-eth2
tcpdump: WARNING: s3-eth2: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for f
ull protocol decode
listening on s3-eth2, link-type EN10MB (Ethernet), capt
ure size 65535 bytes
02:58:35.297369 MPLS (label 17, exp 0, [S], ttl 64) IP
10.0.0.1 > 10.0.0.6: ICMP echo request, id 19820, seq 3
, length 64
02:58:44.046088 MPLS (label 17, exp 0, [S], ttl 64) IP
10.0.0.1 > 10.0.0.6: ICMP echo request, id 19820, seq 3
, length 64
```

(b)

```
> tcpdump -i s5-eth1
tcpdump: WARNING: s5-eth1: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for f
ull protocol decode
listening on s5-eth1, link-type EN10MB (Ethernet), capt
ure size 65535 bytes
02:58:38.542110 MPLS (label 17, exp 0, [S], ttl 64) IP
10.0.0.1 > 10.0.0.6: ICMP echo request, id 7638, seq 5,
 length 64
02:58:38.542797 MPLS (label 16, exp 0, [S], ttl 64) IP
10.0.0.6 > 10.0.0.1: ICMP echo reply, id 7638, seq 5, l
ength 64
02:58:43.548603 MPLS (label 17, exp 0, [S], ttl 64) IP
10.0.0.1 > 10.0.0.6: ICMP echo request, id 7638, seq 6,
 length 64
02:58:43.549309 MPLS (label 16, exp 0, [S], ttl 64) IP
10.0.0.6 > 10.0.0.1: ICMP echo reply, id 7638, seq 6, l
ength 64
02:58:44.046169 MPLS (label 17, exp 0, [S], ttl 64) IP
10.0.0.1 > 10.0.0.6: ICMP echo request, id 19820, seq 3
, length 64
02:58:44.046869 MPLS (label 16, exp 0, [S], ttl 64) IP
10.0.0.6 > 10.0.0.1: ICMP echo reply, id 19820, seq 3,
```

(c)

```
ENDING (xid=0xF0FF00F0):
stat_req{exp_type="state", table="all", match=oxm{all m
atch}}

RECEIVED (xid=0xF0FF00F0):
stat_repl{exp_type="state", stats=[

TABLE = 0

{table="0", key={eth_src="00:00:00:00:00:01", eth_dst="
00:00:00:00:00:06"}, state="17", dur_s="3", dur_ns="298
386000", idle_to="0", idle_rb="0", hard_to="10000000",
hard_rb="100"},

{table="0", key={eth_src="*", eth_dst="*"}, state="0"}

]}
```

(d)

Fig. 20: Experimental results of the re-routing attack: (a) Spoofed MPLS packet (labeled 17) used to pretend a link failure, (b) Traffic through the switch S_3 in case of having attack in the network. This switch only receives spoofed packets sent by the attacker, (c) Forward and reverse path traffic passing through the switch S_5, while having attack in the network. Switch S_5 receives all the packets related to source 10.0.0.1 and destination 10.0.0.6 from switch S_2, (d) State table of switch S_2 in case of receiving failure report.

View publication stats