StateSec: Stateful Monitoring for DDoS Protection in Software Defined Networks

Julien Boite, Pierre-Alexis Nardin, Filippo Rebecchi, Mathieu Bouet, Vania Conan

StateSec: Stateful Monitoring for DDoS Protection in Software Defined Networks

Julien Boite, Pierre-Alexis Nardin, Filippo Rebecchi, Mathieu Bouet, and Vania Conan
Thales Communications & Security
{julien.boite, pierre-alexis.nardin, filippo.rebecchi, mathieu.bouet, vania.conan}@thalesgroup.com

***Abstract*—Software-Defined Networking (SDN) allows for fast reactions to security threats by dynamically enforcing simple forwarding rules as counter-measures. However, in classic SDN all the intelligence resides at the controller, with the switches only capable of performing *stateless* forwarding as ruled by the controller. It follows that the controller, in addition to network management and control duties, must collect and process any piece of information required to take advanced (stateful) forwarding decisions. This threatens both to overload the controller and to congest the control channel. On the other hand, *stateful SDN* represents a new concept, developed both to improve reactivity and to offload the controller and the control channel by delegating local treatments to the switches. In this paper, we adopt this *stateful* paradigm to protect end-hosts from Distributed Denial of Service (DDoS). We propose *StateSec*, a novel approach based on in-switch processing capabilities to detect and mitigate DDoS attacks. *StateSec* monitors packets matching configurable traffic features (e.g., IP src/dst, port src/dst) without resorting to the controller. By feeding an entropy-based algorithm with such monitoring features, *StateSec* detects and mitigates several threats such as (D)DoS and port scans with high accuracy. We implemented *StateSec* and compared it with a state-of-the-art approach to monitor traffic in SDN. We show that *StateSec* is more efficient: it achieves very accurate detection levels, limiting at the same time the control plane overhead.**

I. Introduction

Denial of Service (DoS) attacks gained momentum in the recent years, threatening both network and web infrastructures around the world. By using distributed spambots (DDoS) and/or sophisticated reflection techniques (DRDoS), attackers can easily generate traffic volumes in the order of Tbps [1], also causing catastrophic consequences, as proven by the Dyn attack in October, 2016 that led to major disruptions of internet services both in US and Europe [2].

Following a recent trend, traditional network security appliances are migrating towards programmability concepts. Indeed, Software-Defined Networking (SDN) enables an easier network and service management, resulting an interesting means to enforce security policies. In particular, many previous works in the literature illustrated how to take benefit from the flexibility of SDN in order to quickly setup counter-measures to security threats, such as DDoS, generally by dynamically enforcing simple forwarding rules [3]. Also, security equipment manufacturers have already started to develop and sell commercial products based on SDN concepts [4]. Following the classical SDN approach – all the network intelligence resides at the controller, and the switches have the simple task of enforcing a *stateless* forwarding as dictated by the controller – most of the state-of-the-art approaches rely completely on the controller for dealing with threat protection processes. In this case, a controller has to deal both with stateful forwarding decisions and application processing, thus risking being overloaded and failing consequently.

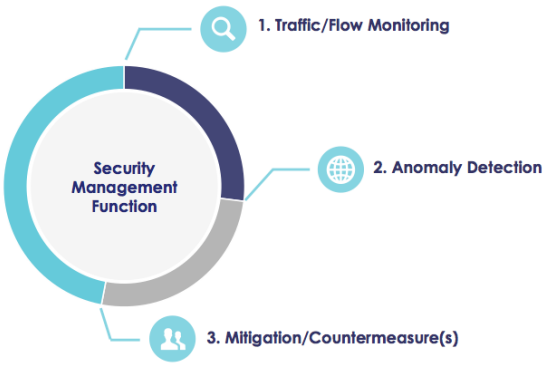

Fig. 1. The three main steps of the security management function, generally implemented at the SDN controller.

Recently, a novel approach called *stateful SDN* has been proposed to improve reactivity and to offload the controller and the control plane channel by delegating some local treatments to the switches [5]. In *stateful SDN*, the forwarding logic is modeled as multiple Finite State Machines (FSM) implemented into switches as *state tables* associated with pre-configured flow tables in the forwarding pipeline.

In this paper, we start from the *stateful SDN* concept as defined in [5], and we design on top of it a novel DDoS detection and mitigation strategy named *StateSec* to showcase the benefits of such an approach. *StateSec* exploits the efficient monitoring capabilities offered by *stateful SDN*, namely the delegation of local processing to switches, to reduce the burden on the controller, still achieving very accurate detection levels. *StateSec* consists of three main steps, shown in Fig. 1 and having the following properties:

- Traffic monitoring of pertinent features (e.g., `IP src`, `IP dst`, `port src`, `port dst`) is handled inside the switch by using stateful programming, thus allowing scalable and very precise monitoring of traffic to feed the detection algorithm.
- Anomaly detection is based on an entropy-based algorithm [6] currently implemented on the controller side (the detection could be integrated directly into the switch when Extended Finite State Machine (XFSM) [7] will

be available). By feeding this algorithm with exact information about the traffic, we manage to detect and differentiate several types of attacks with high accuracy (e.g., DoS, DDoS, Port Scan, etc.).
- Whenever anomalies are detected, mitigation actions are taken at the controller by setting up the most appropriate set of reactions. For instance, DoS attacks originating from a single source are simply filtered. However, more complex mitigation actions can be applied in case of evolved attacks (e.g., rate-limiting, forwarding suspicious traffic towards a blackhole or a DPI engine, etc.).

To validate the benefits of *StateSec*, we compare its performance against native OpenFlow monitoring and sFlow [8], which is a well-known approach for traffic monitoring in SDN. Evaluations are carried out in a controlled environment that mimics a company-wide network deployment and the results confirm the effectiveness of *StateSec* in both reducing the overhead on the control plane and quickly detecting and mitigating ongoing DDoS attacks.

The remainder of this paper is structured as follows. In Section II, we discuss the related work. *StateSec*'s design considerations are presented in Section III, and details on its current implementation are given in Section IV. The evaluation of *StateSec* with a DoS use case first, and then with different DDoS attacks is presented in Section V. Finally, Section VI draws the conclusions and points out future work.

II. Related Work

Different types of DDoS attacks exist [9]. Their impact can be significant: they are able to generate such a huge amount of traffic causing targets to crash and the associated service(s) to become partially or totally unavailable. Detecting and mitigating DDoS attacks is far from being straightforward. First, they usually do not come from a single identified source, which makes remediation very difficult without also affecting legitimate traffic. Second, they appear either very suddenly, thus requiring fast reaction to counter their effects, or very slowly, thus making the detection even more complicated [10].

Simplicity and flexibility are some of the key features offered by SDN, making it an ideal candidate for managing network security. Hence, multiple SDN-based DDoS detection and mitigation schemes have been proposed in the literature. Some of them discuss the bottlenecks introduced by SDN (e.g., related to the flow tables exhaustion at switches and the controller overload), focusing on the protection of the network itself [11], [12]. Others deal more with the protection of communication endpoints (user terminals and application servers) [3], [13]–[16]. Similarly, in this paper we focus on the protection of the communication endpoints by taking advantage of SDN concepts.

In SDN, the controller has a global view of the network and interacts with switches using a dedicated protocol (e.g., OpenFlow [17]). For this reason, implementing a DDoS protection application on top of the controller is somehow a standard approach. As a result, most of the related work describe methods implemented at the controller level. They mainly differ from each other depending on how they perform the three steps required to handle DDoS protection, namely 1) traffic monitoring, 2) attack detection, and 3) attack mitigation. For instance, a controller application can store information such as host/port bindings by frequently requesting switches for ports and flows statistics through standard OpenFlow messages [13]. It is thus possible to detect anomalies and react with *Quality of Service* and *Block State* mitigation actions. However, performing many, yet simple, computations on the controller side has a negative impact in terms of performance (e.g., throughput decrease, latency increase). Another option is to replicate the traffic towards an Intrusion Detection System (IDS) [14]. Threat identification can be performed by correlating IDS alerts through attack graphs at the controller. Simpler yet powerful options are to use statistical tests [18] or entropy-based algorithms for anomaly detection [6].

Regardless of the detection strategy employed, the native OpenFlow approach for monitoring, with the controller that periodically gathers the flow table entries to collect counters, has the unpleasant effect of congesting the control plane [3]. Performance can be improved by using the sampling techniques of sFlow [8]. However, sFlow must be carefully configured because the gains on the control plane load trades off the monitoring precision [19]. Another option is to delegate as much computation as possible to the switches without compromising their performance, letting the controller being only in charge of mitigation [15], [16], [20]. An emerging approach in this sense is *stateful SDN* [5], that implies faster reaction times and less load over the control plane [21]. *StateSec* implements a stateful monitoring process running into the switch to gather very precise information and feed an entropy-based detection algorithm.

III. *StateSec*'s Design Basics

We have developed *StateSec* with the overall objective of protecting the communication endpoints from various security threats by employing SDN concepts. While throughout the paper we focus on (D)DoS attacks, *StateSec* can be easily employed to detect and mitigate other security threats such as port scans and ICMP flooding. In the design phase, we covered the three stages required to handle security management function, namely *monitoring*, *detection* and *mitigation* as depicted in Fig. 1. Our objective is a quick reaction time and a reduction of the overhead induced on the control channel by the communications between a switch and its controller. To this end, *StateSec* delegates part of the processing to the switches, following the *stateful SDN* principles [5].

A. Monitoring & detection: local information stays local

As it emerges from Fig. 2, most of the related works perform all the three stages of the security management function at the controller. Following this classical approach, a SDN switch can be seen as a "dumb" device, with all the network intelligence located at the controller. This implies that, in order to monitor the traffic, the controller has to retrieve the entire

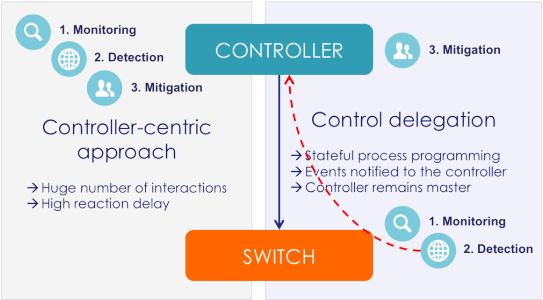

Fig. 2. Classical approaches to DDoS detection and mitigation are controller-centric (left). *StateSec* delegates monitoring and detection functions to the switch since they are performed on local information (right).

flow table entries of any controlled switch. As we will see in Section IV-A, this approach is neither efficient nor scalable.

Instead, *StateSec* delegates monitoring and (in the near future) detection directly into the switches. Since both traffic monitoring and anomaly detection involve only local information, they can be done not necessarily at a central location (e.g., the controller). When an anomaly is detected, the switch notifies the controller so that counter-measures can be elaborated and applied, potentially network-wide. On the one hand, this model does not break the SDN philosophy: the controller retains the full control of the network management (e.g., by orchestrating mitigation actions when needed). On the other hand, switches have to become smarter in order to handle some local processing in place of the controller.

Since monitoring and detection are tightly coupled, one may want to process the monitored information directly where observed. *StateSec* targets this objective, thus saving the communications required to collect the information at a different place before processing it. However, delegating both monitoring and detection to a switch is not straightforward, since it depends on the abstraction used for the in-switch processing.

B. In-switch Processing Abstraction

StateSec relies on *OpenState* [22], an open source implementation of the *stateful SDN* concept originally proposed in [5][1]. In-switch processing benefits from three additional components at the switch, as detailed in Fig. 3:

1) *State table*: linked to a *Flow table* configured as stateful, the *State table* associates a configurable key (e.g., a `src IP`) to a state. Whenever a feature of the received packet matches one of the keys, the associated state value is added to the packet's metadata and forwarded to the *Flow table*.
2) *State match field*: an additional field to match on in the *Flow table*. Using the state of a packet (carried in metadata) as a match field allows applying state-dependent policies to packets.
3) *Set_state action*: a new action to update the state for a configurable key (e.g., a `src IP`) inside the *State table*. For instance, this action could be employed to increase the counter associated with a given `src IP` key.

[1] *OpenState* is developed on top of the *Ryu* controller and the *ofsoftswitch13* software switch.

At the foundation of *StateSec* resides a programmable control loop embedded into the switch. State transitions are configured in the *Flow tables* and different actions can be applied depending on the state of a packet. The bottom line is that this stateful implementation allows the switch to 1) keep states associated with packet features, and 2) apply programmatically forwarding actions to packets according to these states. This approach goes in the direction of delegating tasks to the switch by programming it to perform conditional actions without resorting to the controller. *OpenState* also defines OpenFlow-compatible messages (by using the experimenter field) for the controller to configure and operate the above-mentioned structures in the switch (e.g., to get the state table entries).

C. What is possible today, what to expect for tomorrow?

The current *OpenState* implementation allows the switch to run only simple Finite State Machines (FSMs), where transitions are limited to a change of state. Even if this abstraction is already powerful (e.g., examples of applications are proposed in [22]), it does not allow the switch to perform comparisons or complex computations yet. As a consequence, the detection process, which consists in comparing values against some thresholds, cannot be implemented inside the switch at the current stage of development. However, authors of *OpenState* are currently working on an extended abstraction to allow performing simple computations and memory operations during state transitions [7]. *StateSec* is ready by design for this evolution. Indeed, we made the choice of a simple enough detection algorithm to be easily implemented with the operations that will be available in the extended abstraction. We expect its rapid integration into the switch as soon as the extended in-switch processing abstraction will be available.

IV. *StateSec*'s Implementation

Following the design decisions explained before, in this section we give an overview of the implementation of *StateSec*. We describe the three main stages required to handle the security management function, namely monitoring, detection and mitigation. *StateSec* implements the whole DDoS detection and mitigation process. In order to offer efficient, fast and reliable protection, it targets delegating both the monitoring and detection processes to the switch, while the controller elaborates and orchestrates the mitigation actions.

A. Monitoring

Information about traffic flows must be gathered in order to feed the detection algorithm. In particular, *StateSec* needs to collect information about each *traffic feature* taken into account in the detection. In our evaluations, we considered four main *traffic features*, namely `dst IP`, `dst port`, `src IP`, `src port` along with the information on the type of transport layer protocol employed (e.g., TCP, UDP, etc.). These features are employed in the detection process, so the switch needs to monitor them all.

Several methods already exist in SDN environments for traffic monitoring. In the following, we will list these approaches,

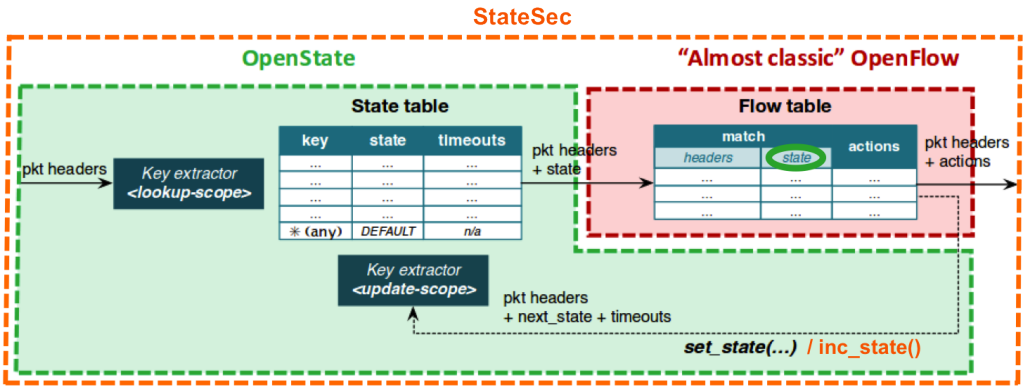

Fig. 3. *StateSec* switch architecture.

highlighting also their drawbacks in terms of switching performance and control plane overhead. In effect, they turn out to be far from optimal for listing *traffic features* (e.g., destination IP addresses) and counting how many times they appear during a time interval. On the contrary, we developed a monitoring scheme that makes the tracking of *traffic features* quick and efficient, by exploiting the standalone reconfiguration of state tables offered by stateful in-switch processing.

1) Native OpenFlow Monitoring: the OpenFlow protocol defines the messages exchanged between the controller and a switch [17]. The controller can natively gather the content of flow tables, including packet and byte counters associated to each entry, using the *FlowStatsRequest* message. While interesting for gathering aggregate flow statistics, the native OpenFlow monitoring does not scale well when listing and counting many traffic features with small granularity (e.g., statistics over each `src IP` or `src port`), as required for DDoS detection. Indeed, for any individual feature to monitor, this approach requires inserting a new flow rule in the flow table of the switch. A direct effect dictated by the increased size of flow tables is a reduction in forwarding performance. For instance, let consider one *traffic feature*: the destination IP address. Monitoring (e.g., listing and counting occurrences) all the destination IP addresses seen in packets' headers requires inserting one forwarding rule per IP address (a second flow rule may also be created for each IP address as source when the source IP address *traffic feature* is monitored). The same principle applies to each monitored *traffic feature*. With this approach, flow tables at switches are quickly overfilled, impacting negatively the performance of forwarding.

2) sFlow Sampling: sFlow is a well-known monitoring approach relying on packet sampling [8]. An sFlow agent integrated into the switch is in charge of both sampling packets at a configurable rate [23] and transmitting their header to a collector, logically running near the controller. Many SDN equipment manufacturers have already integrated sFlow agents in their products. Moreover, one of the reference software switches, Open vSwitch (OVS), is also sFlow-compatible.

sFlow offers significant benefits when compared to the native OpenFlow approach, since the monitoring functionality is totally decoupled from forwarding, i.e., it is not necessary to add flow rules to ensure both forwarding and monitoring of flows. Based on sampled packets, the sFlow collector allows performing DDoS detection by maintaining an up-to-date list of features and counters. However, sampling may introduce a significant approximation, potentially harming the detection accuracy. Indeed, the configuration of the sampling rate is very important [19], as we will show in Section V. Obviously, by lowering the sampling rate the precision decreases. On the other hand, a too high sampling rate concurs in overloading the control plane channel. Indeed, it is of paramount importance – and most of the time this step requires a fine tuning – to find the right tradeoff between detection accuracy and overhead on the control channel.

3) StateSec's State-Based Monitoring: we developed a new system that relies on stateful in-switch processing to implement forwarding-independent and modular monitoring of *traffic features*. *StateSec*'s monitoring uses the state and flow tables in an *OpenState*-compliant switch to list features and count the exact number of times they appear, independently from the forwarding rules. Considering a single *traffic feature*, e.g., `src IP`, the controller initializes the following elements in the switch (cf. Fig. 3):

- a flow table is configured as stateful, and the controller inserts in the associated state table an entry to make any unknown key associated to the 'DEFAULT' state (in the case of counting packets, the 'DEFAULT' state is 0),
- the lookup scope extracts from the packet headers the field corresponding to the monitored *traffic feature* (`src IP` in this example),
- the update scope looks at the same field as the lookup scope, corresponding to the monitored *traffic feature*.

Finally, we extended *OpenState* in order to perform incremental state updates. Indeed, the *OpenState* implementation offers a single function to update a state, namely `set_state(newState)`, which requires to pre-configure the value of the *newState* value in the flow table. This function is adapted to FSMs where states and transitions are known in

TABLE I
ENTROPY VARIATIONS RESULTING FROM DIFFERENT TYPE OF ATTACKS.

Type of Attack	**Dst IP**	**Dst Port**	**Src IP**	**Src Port**
DoS flooding	↘	↘	↘	↘
DDoS flooding	↘	↘	↗	↘
DoS flooding with spoofed src port	↘	↘	↘	↗
DDoS flooding with spoofed src port	↘	↘	↗	↗
ICMP DoS flooding	↘		↘	
ICMP DDoS flooding	↘		↗	
Port Scan	↘	↗	↘	↘
Port Scan with spoofed src port	↘	↗	↘	↗

advance. For traffic monitoring applications, in order to count the amount of packets having the same *traffic feature*, the next value of a state is simply an increment by one of its current value. We developed an efficient `inc_state()` action to perform such a state update, calling this action in a default flow table entry that matches all packets, whatever their state is. As a consequence, the state associated to the `src IP` (in general, the monitored *traffic feature*) of the packet is updated in the state table (a new entry is created for the first occurrence). This process is repeated for each monitored *traffic feature*, and different tables are linked to form a pipeline where each *State table* monitors one *traffic feature*, independently from forwarding.[2]

This process allows the switch to count the exact number of times each symbol appears for each *traffic features* one wants to monitor, ensuring thus a high accuracy of the detection algorithm. Regardless of where it is deployed, the detection algorithm can access this information through a new *StateSec* control message that atomically retrieves and flushes (to reset counters values after the lecture) the content of state tables. By periodically gathering the *traffic feature* keys and the associated states/counters from state tables, the detection algorithm can perform its duties.

B. Detection

In order to validate the benefits of the in-switch monitoring offered by *StateSec*, we adapted a simple statistical-based algorithm from the literature [6]. One advantage of this strategy is that it is directly applicable to the *traffic features* that are offered by the switch. Moreover, once XFSM will be available, this lightweight detection algorithm could be easily integrated into the switch. Indeed, unlike pattern-based detection tools, statistical approaches do not require large memory nor high processing power. We thus developed *StateSec*'s detection process using an entropy-based algorithm, a statistical approach for detecting anomalies in a distribution.

1) Entropy-based Algorithm: entropy measures the unpredictability of a distribution. Sudden variations in the measured entropy allow detecting anomalies in the distribution of *traffic features*. To this end, we employed the normalized entropy formula given below, where p_i stands for the probability of the symbol *i* to appear, and *n* represents the number of symbols.

$$H(X) = -\sum_{i=1}^{n} p_i \frac{log_2(p_i)}{log_2(n)}$$

A statistically high incidence for a given symbol to appear leads to a reduced entropy (and to a slightly more concentrated distribution). Conversely, low and dispersed incidences translate to higher entropy values. It follows that entropy-based algorithms are widely used for the detection of attacks in communication networks [3], [6]. By identifying significant changes in the randomness of consecutive *traffic features* distributions, this statistical approach can detect several type of attacks, including (D)DoS, with better accuracy than methods based on volume metrics [6].

Next, we correlate different *traffic features* entropy variations to identify the type of attack. Table I shows that monitoring the entropy of four *traffic features* distributions (`dst IP`, `dst port`, `src IP`, `src port`) allows differentiating between multiple types of attacks. For instance, a significant decrease of the entropy for both the `dst IP` and the `dst port` features may be observed when a (D)DoS attack occurs. Increases or decreases in the entropy of the `src IP` feature qualify whether or not the attack is distributed. Similarly, the `src port` may be employed to detect spoofing attacks.

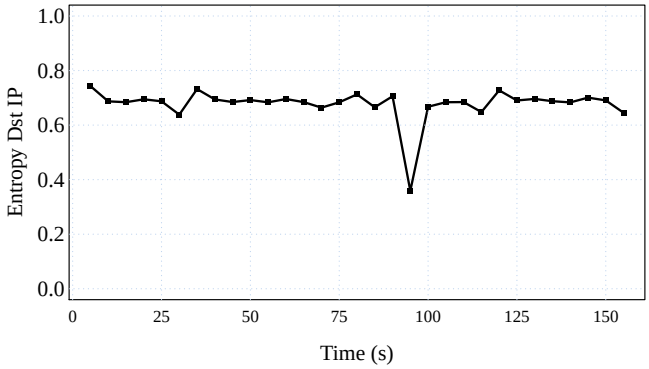

Fig. 4. Entropy variations for the `dst IP` *traffic feature*: an UDP flooding attack is performed around $t = 90s$ towards a single node. This generates a significant drop in the entropy, that returns back to normal levels after detection and mitigation of the attack (traffic from malicious hosts is discarded).

Fig. 4 serves to illustrate the variations in entropy induced on the `dst IP` *traffic feature* while a UDP DoS flooding attack occurs around $t = 90s$. The detected drop in the entropy value results from the fact that the target of the DoS attack represents a large share of the destination traffic seen at the switch. However, some IP addresses or ports may also send or receive more packets than others in real networks, e.g., servers providing very popular services. In this case, a hefty drop in entropy for the `dst IP` *traffic feature* might be related to certain properties of legitimate traffic and not to malicious actions (e.g., unexpected flash crowds arising after major events). Hence, selecting a good detection threshold and implementing additional protection mechanisms is paramount to avoid detecting potential false positives.

[2]Note that the forwarding table can stand either at the beginning or at the end of the pipeline.

2) Detection Thresholds and Sensitivity: the sensitivity of the detection process is one of the key elements allowing to identify threats with a limited number of false positives. We employed a statistical model to define the lowest and highest acceptable values of entropy. For any given *traffic feature* to monitor, the initial entropy values collected by the detection algorithm bootstrap a learning phase – assuming no attack occurs during the bootstrap, these samples can be considered as a snapshot of the steady-state situation. After the bootstrapping phase is completed, any new entropy value is compared against statistics from the sample of entropy values collected so far (learning never stops, but abnormal entropy values are discarded from reference samples). The statistics computed from this sample are the mean μ_e and the standard deviation σ_e. From the normal distribution defined by $\mathcal{N}(\mu_e, \sigma_e)$, we identify the upper and lower bounds that determine whether or not the last computed entropy results from a legitimate situation. Following the normal distribution, detection thresholds can be configured to be located at $\mu_e \pm m \cdot \sigma_e$ with $m \in \mathbb{N}$. The value of m relates to the algorithm sensitivity. For instance, if the system is configured to use thresholds one σ_e away from the mean ($m = 1$), it will be very sensitive but may generate more false positives than with thresholds three σ_e away from the mean ($m = 3$).

3) Outliers Protection: a key step consists in identifying the victim(s) (e.g., target hosts and service ports), and, if possible, the attacker(s), (e.g., hosts and source ports) that generate the attack. No matter the value chosen for m, in order to reduce further the impact of false positives we enforce an additional mechanism to protect outliers. In effect, in any network some legitimate hosts (e.g., IP addresses) or services (e.g., ports) are much more solicited than others (e.g., an HTTP server receives a lot more traffic at its address on port 80). This evidence does not sit well with a detection based only on statistical methods, and additional care should be taken in order to limit the occurrence of false positives. For this reason, when the last computed entropy for a *traffic feature* resides in-between the upper and the lower bounds, then no entropy violation is detected (we assume being in a legitimate situation). Similarly to the entropy detection phase, the algorithm computes the mean μ_d and the standard deviation σ_d, this time directly on the dictionary of symbols and their associated counters. Symbols below or above the $\mu_d \pm m \cdot \sigma_d$ thresholds are considered as outliers (e.g., IP addresses that are much more solicited than the others, in a non entropy-violation situation) and will be stored. Then, whenever an entropy violation is detected, an additional step is taken to verify whether the suspects are from these stored values. If that is the case, an alert is launched only if the counters are outside $\mu_d \pm m \cdot \sigma_d$.

4) Detection Process Primer: the detection process consists in computing the entropy of each monitored *traffic feature* (`src addr`, `src port`, `dst addr`, `dst port`), based on the list of symbols and their associated counters. Then, the algorithm computes the upper and lower thresholds and evaluates whether the entropy value is below or above these bounds, individually for each *traffic feature*. If the entropy of one or multiple *traffic features* generates a drop or raise that crosses thresholds, an anomaly is detected. In any case, outlier values are stored to improve the detection in the future.
This process is repeated at regular interval of times, called *Time Window* in the rest of the paper. When an anomaly is detected, the variations in the entropy for the different *traffic features* are analyzed, taking also into account the outliers, combined to identify the attack (following Table I), and the mitigation process takes place consequently.

C. Mitigation

Finally, mitigation action(s) are elaborated to protect legitimate users. Once a violation is detected, new flow rule(s) are installed into the switch with a high priority to match suspicious packets. The precision of mitigation rule(s) depends in large extent on the precision of the information identified in the detection phase (source and destination IP addresses and ports). The controller installs mitigation rules into the switch through standard OpenFlow functionalities. Existing actions allow to drop, queue, prioritize, blackhole or even forward traffic towards an IDS. We can also imagine many more complex actions: for instance, suspicious traffic could be forwarded towards a Deep Packet Inspection (DPI) device (or Virtual Network Function) running somewhere in the network for a further analysis conditioning a final remediation decision.

V. *StateSec*'s Evaluation

We evaluated *StateSec*'s performance in terms of false positives, detection accuracy and control plane overhead over a real testbed. This allows identifying and discussing the tradeoff between the cost of a precise monitoring and the detection efficiency. Since the core contribution of the paper concerns the efficiency of the monitoring phase, we compare *StateSec*'s results to those obtained with the same entropy-based detection strategy coupled with *sFlow* for monitoring.

A. Experimental Testbed Setup

The evaluation has been carried out in a controlled environment composed of two virtual machines (VMs): one running an extended version of the Ryu controller (3 vCPUs clocked at 2GHz, with 16 GB RAM), and the other (2 vCPUs clocked at 2GHz, with 8 GB RAM) running Mininet [24], a powerful tool used to emulate complex data plane topologies. In our experiments, Mininet emulates one switch running either our extended version of *OpenState*, or a standard *Open vSwitch* (OVS) coupled with an *sFlow* agent[3], one replaying host, several attacker hosts, and one application server as depicted in Fig. 5.

B. Scenario and Parameters

In our experiments, host *h1* replays legitimate traffic taken from the "BigFlows" trace (available at [25]), which captures a real network traffic on a busy private network's access point

[3]We used OVS for experiments with *sFlow* because this software switch can integrate an *sFlow* agent while *ofsoftswich13* cannot.

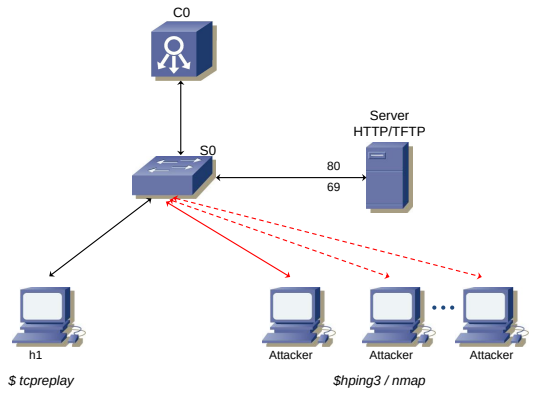

Fig. 5. Testbed topology: host $h1$ replays legitimate traffic from the trace "BigFlows" [25] with *tcpreplay*. Each attacker host generates a DoS attack towards the HTTP/TFTP server.

to the Internet. This approach makes it possible to mimic a much more complex topology than the one presented in Fig. 5. The employed trace contains 50000 packets and has been chosen to include a significant number of hosts (679) that exchange diverse types of legitimate traffic (9605, 40249 and 118 packets for UDP, TCP and ICMP respectively), making it comparable to the traffic generated in a large company. The trace is indefinitely replayed at the speed of 3.3 Mb/s.

The attacker(s) generates a (D)DoS flooding attack towards the server listening on ports 69 (TFTP) and/or 80 (HTTP), by using *hping3* [26] (1 pkt/1200 μs). During each experiment, we captured the traffic exchanged between the switch and the controller (control plane traffic) using *tcpdump*, and we observed also the reaction time of the detection process.
We evaluate the performance of *StateSec* and *sFlow* for different values of three key parameters:

- **Time Window**: the time interval in seconds between two calls to the detection algorithm. With *StateSec*, the controller has to gather first the counters for the monitored *traffic features* from the switch, then it can trigger the detection algorithm. With *sFlow*, samples are gathered in real-time with *sflowtool*. In both cases, the Time Window parameter has a direct impact on both reactivity and control overhead.
- **Detection Sensitivity**: the sensitivity value m that defines the thresholds to be used in the detection process (both for *StateSec* and *sFlow*), as described in Section IV-B. A fine tuning of this parameter is required in order to avoid as much as possible the presence of false positives.
- **Sampling Rate**: the period used to sample packets with *sFlow*. Its value defines how many packets must be seen before picking a sample and sending it to the collector. The sampling rate [23] is configured with values ranging from 1 (high – a sample for each packet) to 100 (low – a sample each 100 packets) in our experiments.

For each set of parameters, we ran the experiments 30 times with a fixed initial time (26 s) of the (D)DoS attack in order to highlight the particular interactions between the sampling rate value and the detection accuracy.

C. DoS Experiments Results

First, we analyze the impact of the detection sensitivity value m on the amount of false positives at stationary regime without attacks, by replaying the traffic trace and counting detections. The fine tuning of detection thresholds is particularly critical in order to cut down the number of false positives, thus preventing artificial denial of service to legitimate users while preserving sensitivity to real attacks. Fig. 6 gives an idea of the values at stake (note the logarithmic y-axis). As a result, in order to avoid as much as possible false positives, in the following we will use a detection sensitivity set to $m = 3$ if not explicitly mentioned. Fig. 6 highlights also two clear trends: 1) by increasing the Time Window value, the sensitivity value without false positives decreases; and 2) for increasing sampling levels (*sFlow* only), regardless of the sensitivity value, the amount of false positives decreases. Unfortunately, as we will see later, longer Time Window and higher sampling rates imply increased control overhead.

We confirm this observation by looking at Fig. 7 that represents the average traffic flowing on the control plane channel during one Time Window. The overhead for *StateSec* includes the messages exchanged to gather counters from the state tables, whose size depends on the variety of traffic flowing through the switch. Instead, with *sFlow* it includes the packet samples sent from the agent to the collector. In this case, the control plane traffic depends rather on the data traffic throughput at the switch. We observe that with frequent sampling (e.g., sampling rate set to 1), *sFlow* overloads the control plane. This is the price to pay for using *sFlow* with the same precision level offered by *StateSec*, which in turns provides always exact counters. On the other side, *StateSec* generates less load on the control plane than *sFlow* with sampling rates set both to 5 and 10 and Time Window values higher than 2 seconds. So, unless using *sFlow* with very low sampling rates (e.g., a sample every 50 or 100 packets) in exchange for the penalty of a lower precision, *StateSec* is more efficient: it generates less overhead maintaining very precise monitoring information.

Ultimately, the detection rate is the most meaningful parameter for any credible (D)DoS protection strategy. At the same time, reactivity plays a major role, since it relates to the expected outage time of targets before a threat is mitigated. In that sense, Fig. 8 depicts the average attack detection rates for *StateSec* and *sFlow* for different sampling rates and Time Window intervals in case of an attack. We differentiate the reactivity performance by considering two types of detection, namely whether an attack is detected within two Time Windows or not. As expected, *sFlow* with low sampling rates leads to poor detection rates, implying also longer reaction times. In a nutshell, low sampling rates lead to less accurate detections and lower reactivity. Moreover, we note that *sFlow* performance depends heavily on the interplay between Time Window and sampling rates. Generally, longer time intervals permit *sFlow* to sample multiple suspicious packets increasing thus the detection likelihood (e.g., compare

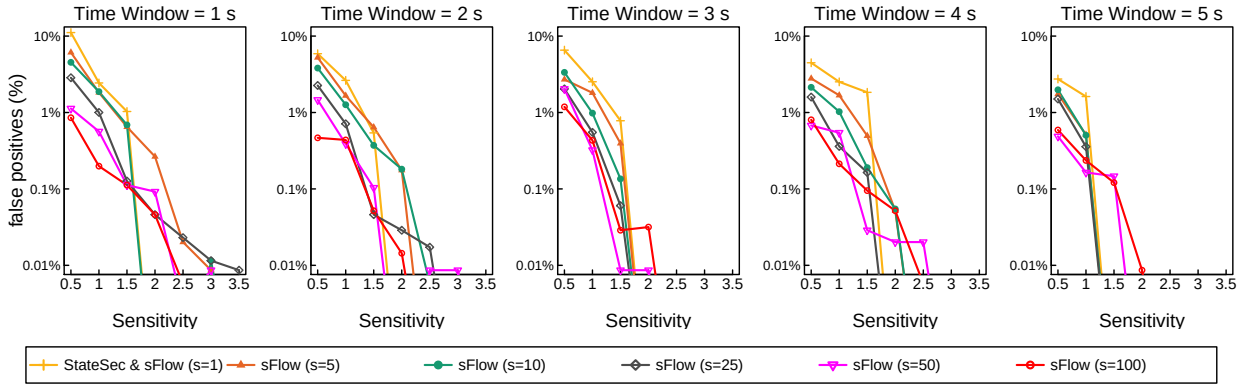

Fig. 6. False positives detected by *StateSec* and *sFlow* depending on the detection sensitivity m when no attacks are performed. For $m > 2.5$ the amount of false positives is below 0.1% for any configuration.

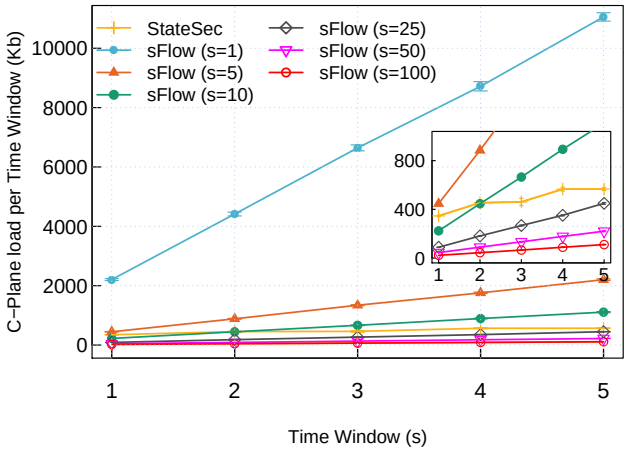

Fig. 7. Control plane channel load during one Time Window.

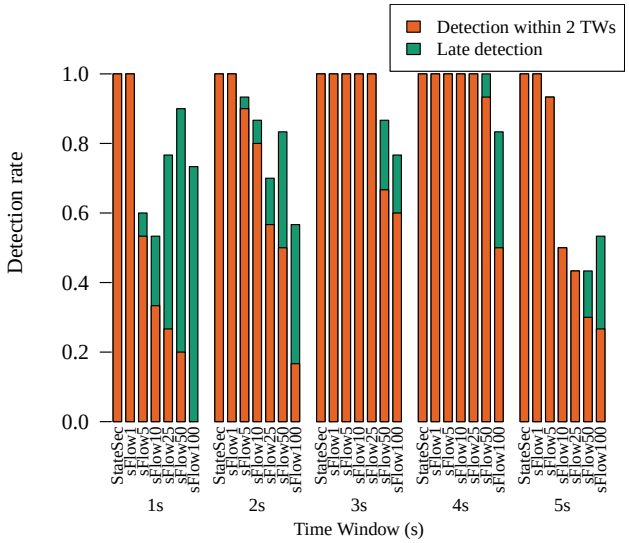

Fig. 8. Detection rates with detection sensitivity set to $m = 3$. Note the hefty drop in detection with *sFlow* for large Time Window and low sampling rates.

the improvement of detection rates for sFlow). On the other hand, lower sampling rates coupled with longer Time Windows tends to smooth out the entropy variations, reducing significantly the detection rate for the magnitude of attack that we tested (an example of this event can be seen in Fig. 8 for Time Window $5s$ and sampling rate 50).

D. Distributed DoS Experiments Results

We extend the DoS scenario used so far to evaluate how *StateSec* detects and mitigates multiple DDoS attacks.

DDoS Attacks: in principle, DDoS attacks have a distributed source. If attackers are limited in number compared to the legitimate hosts, *StateSec* is able to identify most of the attackers. In that case, *StateSec* filters out each attacker as it was generating a simple DoS attack towards a unique target. On the other hand, whenever the DDoS attack is much stronger, i.e., it is generated by a number of attackers comparable to the number of hosts sending legitimate traffic, *StateSec* still can detect the attack without being able always to identify attackers. In order to evaluate this, we replayed the same traffic trace as for the DoS evaluation, this time employing multiple attacking hosts, and we observed that *StateSec* mitigates precisely simultaneous attacks from up to 35 hosts (out of the 679 total hosts in the *BigFlows* trace). In case the number of attackers is larger, *StateSec* protects the application service by temporarily dropping all the traffic towards the victim.

Slow DDoS Attacks: by using slow traffic that appears legitimate in terms of protocols and rates, slow DDoS attacks try to pass undetected by traditional statistical strategies [10]. Indeed, when entropy variations are not pronounced enough, the algorithm will not detect any anomaly. This traffic, harmless at first sight, aims at exhausting the victims resources.

We extended *StateSec* to detect also this type of attacks by comparing the last computed entropy value not only to the distribution of the whole history of entropies, but also to the distribution of the history excepting the last n values (in our

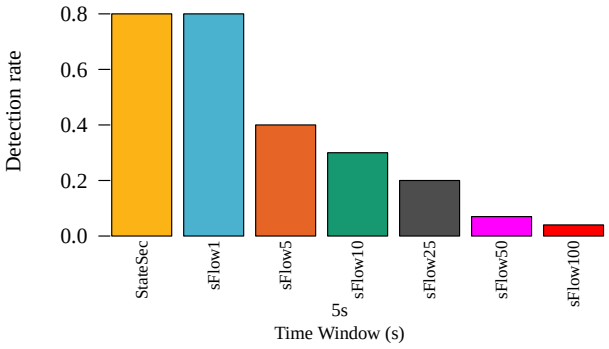

Fig. 9. Slow DDoS detection rates for a Time Window of 5s and $n = 4$.

evaluations, we considered $n = 4$). If there is a significant decrease in the current entropy value compared to values seen in the past, a slow DDoS attack is detected.

We compared how *StateSec* and *sFlow* face slow DDoS attacks using a Time Window of 5 seconds, and a detection sensibility set to $m = 2$ (entropy variations are not significant enough to use thresholds 3 times away from the mean). To do so, we emulated a slow DDoS attack with 20 attackers flooding UDP packets towards the same destination and port. Only one attacker is present at the beginning of the attack, and a new attacker starts flooding every 5 seconds with a low intensity attack (1 pkt/10000 μs). Fig. 9 shows that *StateSec* offers detection rates near 80%. When *StateSec* detects the attack, nearly 85% of attackers are identified and the traffic they send is blocked. In addition, this result highlights again the consequence of a low precision in the monitoring: when the attack comes slowly, it is even clearer that *sFlow* used with low sampling rates cannot accurately detect and mitigate threats.

VI. Conclusion and Perspectives

In this paper, we presented *StateSec*, a novel approach based on stateful SDN to protect communication endpoints from (D)DoS attacks. *StateSec* relies on in-switch processing capabilities to delegate the traffic monitoring phase to the switch. We developed the required extensions to make this scheme a reality, and evaluated its performance on a real testbed. Comparisons against sFlow confirm that *StateSec* is efficient in terms of control plane occupation, while it is clearly better in terms of monitoring precision, and, as a consequence, on detection accuracy.

Future work consists in adapting *StateSec* to the extended in-switch processing abstraction that authors of OpenState are currently developing, in order to actually integrate our entropy-based detection process into the switch. We also plan to elaborate more sophisticated mitigation actions, potentially taking network-wide decisions over more complex topologies.

Acknowledgment

This work has been partly funded by the EU in the context of the H2020 BEBA project (Grant Agreement 644122).

References

[1] S. Khandelwal, "World's largest 1 Tbps DDoS Attack launched from 152,000 hacked Smart Devices," 2016. [Online]. Available: http://thehackernews.com/2016/09/ddos-attack-iot.html
[2] K. York, "Dyn Statement on 10/21/2016 DDoS Attack," 2016. [Online]. Available: http://dyn.com/blog/dyn-statement-on-10212016-ddos-attack/
[3] K. Giotis, C. Argyropoulos, G. Androulidakis, D. Kalogeras, and V. Maglaris, "Combining OpenFlow and sFlow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments," *Computer Networks*, vol. 62, pp. 122–136, 2014.
[4] RADWARE, "DefenseFlow - SDN Based Network DDoS, Application DoS and APT Protection," in *ONS'14*, 2014. [Online]. Available: http://opennetsummit.org/archives/mar14/site/pdf/2014/sdn-idol/Radware-SDN-Idol-Proposal.pdf
[5] 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.
[6] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," in *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, 2005, pp. 217–228.
[7] 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," pp. 1–14, 2016. [Online]. Available: http://arxiv.org/pdf/1605.01977.pdf
[8] P. Phaal, "sFlow specification Version 5," 2014. [Online]. Available: http://www.sflow.org/sflow_version_5.txt
[9] C. Douligeris and A. Mitrokotsa, "DDoS attacks and defense mechanisms: Classification and state-of-the-art," *Computer Networks*, vol. 44, no. 5, pp. 643–666, 2004.
[10] J. Park, K. Iwai, H. Tanaka, and T. Kurokawa, "Analysis of slow read dos attack," in *ISITA*, 2014, pp. 60–64.
[11] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks," in *NDSS*, 2015.
[12] N. G. Dharma, M. F. Muthohar, J. D. A. Prayuda, K. Priagung, and D. Choi, "Time-based DDoS Detection and Mitigation for SDN Controller," in *APNOMS*, 2015, pp. 550–553.
[13] Y. E. Oktian, S. Lee, and H. Lee, "Mitigating Denial of Service (DoS) attacks in OpenFlow networks," in *ICTC*, 2014, pp. 325–330.
[14] C.-J. Chung, P. Khatkar, T. Xing, J. Lee, and D. Huang, "NICE: Network Intrusion Detection and Countermeasure Selection in Virtual Network Systems," *IEEE Trans Dependable Secure Comput*, vol. 10, no. 4, pp. 198–211, 2013.
[15] T. Chin, X. Mountrouidou, X. Li, and K. Xiong, "An SDN-Supported Collaborative Approach for DDoS Flooding Detection and Containment," in *MILCOM*, 2015, pp. 659–664.
[16] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks," in *ACM CCS*, 2013, pp. 413–424.
[17] N. Mckeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, and S. Louis, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, p. 69, 2008.
[18] F. Soldo and A. Metwally, "Traffic anomaly detection based on the ip size distribution," in *IEEE INFOCOM*, 2012, pp. 2005–2013.
[19] V. Mann, A. Vishnoi, and S. Bidkar, "Living on the edge: Monitoring network flows at the edge in cloud data centers," in *COMSNETS*, 2013, pp. 1–9.
[20] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *ACM SIGCOMM*, 2016, pp. 101–114.
[21] R. Bifulco, J. Boite, M. Bouet, and F. Schneider, "Improving SDN with InSPired switches," in *ACM SOSR*, 2016.
[22] "OpenState SDN project," 2016. [Online]. Available: http://openstate-sdn.org
[23] sFlow blog, "sFlow sampling rates," 2009. [Online]. Available: http://blog.sflow.com/2009/06/sampling-rates.html
[24] Mininet Team, "Mininet," 2016. [Online]. Available: http://mininet.org
[25] Tcpreplay sample captures. [Online]. Available: http://tcpreplay.appneta.com/wiki/captures.html
[26] S. Sanfilippo, "hping," 2006. [Online]. Available: http://www.hping.org/