DDoS protection with stateful software-defined networking

Filippo Rebecchi^{ID} | Julien Boite | Pierre-Alexis Nardin | Mathieu Bouet | Vania Conan

Advanced Information Technologies Lab,
Thales Communications & Security,
Gennevilliers, France

Correspondence

Filippo Rebecchi, Advanced Information
Technologies Lab, Thales
Communications & Security,
Gennevilliers, France.
Email: filippo.rebecchi@thalesgroup.com

Funding information

H2020 BEBA project, Grant/Award
Number: 644122; Celtic-Plus
SENDATE-Tandem; 5G-PPP NRG-5
project, Grant/Award Number: 762013

Summary

Distributed denial of service (DDoS) attacks represent one of the most critical security challenges facing network operators. Software-defined networking (SDN) permits fast reactions to such threats by dynamically enforcing simple forwarding/blocking rules as countermeasures. However, the centralization of the control plane requires that the SDN controller, besides network management operations, should also collect information to identify and mitigate the security menaces. A major drawback of this approach is that it may overload the controller and the control channel. On the other hand, *stateful SDN* represents a new concept, developed to improve reactivity and offload the controller by delegating local treatments to the switches. In this article, we embrace this paradigm to protect end-hosts from DDoS attacks. We propose *StateSec*, a novel approach based on in-switch processing capabilities to detect and mitigate flooding threats. *StateSec* monitors packets matching configurable traffic features without resorting to the controller. By feeding an entropy-based detection algorithm with such monitoring features, it detects and mitigates several threats such as (D)DoS with high accuracy. We implemented *StateSec* in an SDN platform comparing it with state-of-the-art approaches. We show that *StateSec* is far more efficient: It achieves very accurate detection levels, reducing at the same time the control plane overhead. We have also evaluated the memory footprint of *StateSec* for a possible use in production. Finally, we deployed *StateSec* over a real network to tune its parameters and assess its suitability to real-world deployments.

1 | INTRODUCTION

Distributed denial of service (DDoS) attacks gained momentum in the recent years, threatening both network and web infrastructures around the world.¹ Today, the blossoming of low cost services enables everyone to launch devastating attacks relying on the existence of distributed botnets (DDoS) and/or sophisticated reflection/amplification techniques (DRDoS). Embedded Internet of Things devices, exposing nonpatched vulnerabilities and offering always on connectivity, represent an ideal vector of attack (eg, the Mirai botnet). Since Internet of Things is gaining in popularity, larger attacks are expected. Attackers can easily generate traffic volumes in the order of Tbps,² also causing catastrophic consequences and leading to profit loss, as proven by the Dyn attack in October, 2016, that led to major disruptions of Internet services in both the United States and Europe.³ While in the past only some type of business would be likely targets for DDoS, today virtually anyone is concerned.

FIGURE 1 The three main steps of the distributed denial of service (DDoS) detection and mitigation in software-defined networking. State-of-the-art approaches generally implement all three phases inside the software-defined networking controller

Following a recent trend in the networking and IT fields, traditional security appliances are migrating towards programmability concepts. Indeed, software-defined networking (SDN) enables painless network and service management, representing at the same time a credible alternative to enforce security policies. In particular, many previous works in the literature illustrated how to take benefit from the flexibility of SDN to quickly setup countermeasures to security threats, such as DDoS, generally by dynamically enforcing simple forwarding rules.⁴ Likewise, legacy security equipment manufacturers have already started developing and selling commercial products based on SDN.⁵ However, by following the classical SDN approach that promotes the separation between the control plane and data plane functionalities—all the network intelligence is concentrated at the controller, with the switches having the simple task of enforcing *stateless* forwarding rules as instructed by the controller—most of state-of-the-art approaches rely completely on the controller to perform threat protection processes. In this case, network controllers have to deal with both stateful forwarding decisions and application processing, thus risking being overloaded and failing consequently with catastrophic consequences.⁶

Recently, a novel approach called *stateful SDN* has been proposed to improve reactivity and to offload both the controller and the control plane channel by delegating the execution of local treatments to be performed directly on the data plane (ie, at the switches).⁷ In stateful SDN, the programmable logic is modeled as a sequence of multiple finite state machines (FSMs) implemented into switches as *state tables* associated with preconfigured flow tables in the forwarding pipeline. This model has been implemented and is publicly available as open-source software.⁸

In this article, we exploit the stateful SDN concept, as defined in Bianchi et al,⁷ by designing a novel DDoS detection and mitigation strategy named *StateSec* showcasing the benefits of such an approach. *StateSec* exploits the efficient monitoring capabilities offered by in-switch programmability, namely, the delegation of local counting of packet features to switches, to reduce the burden on the controller, and achieving more accurate detection levels compared with state-of-the-art controller-centric strategies. *StateSec* is designed as a highly configurable approach based on stateful in-switch processing capabilities to detect and mitigate DDoS attacks. Similarly to many SDN-based approaches, *StateSec* consists of three main steps, as shown in Figure 1, and has the following properties:

1. Efficient monitoring of pertinent traffic features (eg, IP src, IP dst, port src, and port dst) is handled directly on the data plane inside the switch by taking advantage of stateful in-switch programming, thus allowing scalable and very precise monitoring of traffic to feed the detection algorithm.
2. Anomaly detection is based on an entropy-based algorithm.⁹ The detection algorithm could be implemented both at the switch or at the controller depending on its complexity (as of today, its placement represents a trade-off between detection accuracy and the amount of information exchanged over the control channel). In this article, we evaluate the performance of the detection algorithm deployed at the controller.* By feeding this algorithm with exact information about traffic metrics, *StateSec* manages to detect and differentiate several types of attacks having high accuracy (eg, DoS, DDoS, and Port Scan).

*Anomaly detection may be integrated directly into the switch by exploiting the extended finite state machine (EFSM) packet processor.¹⁰

3. 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 (eg, rate-limiting, forwarding suspicious traffic towards a blackhole or a deep packet inspection engine).

To validate the benefits of *StateSec*, we compared its performance against native OpenFlow monitoring and sFlow,¹¹ which is a well-known approach for traffic monitoring in SDN. Evaluations are conducted initially in a controlled environment that mimics a company-wide network deployment, and experimental results confirm the effectiveness of *StateSec* both in reducing the control plane overhead and in quickly detecting and mitigating ongoing DDoS attacks. Moreover, we verified the memory footprint of *StateSec* under stress confirming the feasibility of the approach. By performing a real-world deployment, we extracted also several useful information to tune finely the parameters of the detection algorithm.

The remainder of this paper is structured as follows. *StateSec*'s design considerations are presented in Section 2, and details on current implementation are given in Section 3. Section 4 presents the evaluation of *StateSec* in terms of detection performance, overhead, and memory use. Also, the lesson learned from a real-world deployment is presented. In Section 5, we discuss the related work. Finally, Section 6 draws the conclusions and points out future directions of work.

2 | STATESEC'S DESIGN BASICS

We have developed *StateSec* with the overall objective of protecting the communication end points from various security threats by using SDN concepts. While throughout the paper we focus on (D)DoS attacks, *StateSec* can be easily used to detect and mitigate other security threats such as port scans and Internet Control Message Protocol (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 Figure 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.⁷

2.1 | Monitoring and detection: distributed vs centralized

As it emerges from Figure 2, most of the related works perform all the three stages of the security management function at the controller. Following the classical approach, an SDN switch can be seen as a “dumb” device, with all the network intelligence located at the controller. This implies that, to monitor the traffic, the controller has to retrieve the entire flow table entries of any controlled switch. As we will see in Section 3.1, this approach is neither efficient nor scalable.

FIGURE 2 Classical software-defined networking-based distributed denial of service detection and mitigation approaches are controller centric; all three phases are performed at the controller with the switch only performing stateless operations (left). *StateSec* exploits stateful principles by delegating monitoring and detection functions to the switch since they are performed on local information (right)

FIGURE 3 Stateful table architecture: For each stateful stage, a state table is added to the flow table. The state table permits to associate a state to each packet. The flow table can match on both the packet headers and its state

Instead, *StateSec* delegates monitoring and, eventually, 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 (eg, the controller). When an anomaly is detected, the switch notifies the controller so that countermeasures 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 network management (eg, by orchestrating mitigation actions when needed). On the other hand, switches have to become smarter 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.

2.2 | In-switch processing abstraction with OpenState

StateSec relies on *OpenState*,⁸ an open-source implementation of the stateful SDN concept originally proposed in Bianchi et al.^{7,†} In-switch processing benefits from three additional components at the switch, as detailed in Figure 3:

1. *State table*: linked to a *flow table* configured as stateful, the *State table* associates a configurable key (eg, 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 (eg, a `src IP`) inside the *state table*. In *StateSec*, this action could be used to increase the counter associated with a given `src IP` key.

At the foundation of *StateSec* resides the 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 (eg, to get the state table entries).

[†]*OpenState* programming abstractions are available for the *Ryu* controller and are ported to the accelerated version of the *ofsoftswitch13* software switch.¹²

2.3 | Future evolutions of *StateSec*

The current *OpenState* implementation allows the switch to run only FSMs, where transitions are limited to a change of state. Even if this abstraction is already powerful (eg, other examples of applications are proposed in previous studies⁸), FSMs do not allow the switch to perform comparisons or complex computations autonomously. As a consequence, the proposed detection process, which consists in computing an entropy value and comparing against its running average, cannot be implemented completely at the switch in the current stage of development, needing to resort to the controller. However, authors of *OpenState* are currently working on an extended abstraction—the extended finite state machine (EFSM) packet processor—enabling simple computations and memory operations during state transitions.¹⁰ *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 EFSM processing abstraction will be available.

3 | *STATESEC*'S IMPLEMENTATION

Following the design decisions explained before, this section provides 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*. To offer efficient, fast, and reliable protection, *StateSec* targets delegating both the monitoring and detection processes to the switch, while the controller elaborates and orchestrates the mitigation actions.

3.1 | Monitoring

Information about traffic flows must be gathered to feed the detection algorithm. For DDoS protection, *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`, and `src port`, along with the information on the type of transport layer protocol used (eg, Transmission Control Protocol [TCP], User Datagram Protocol [UDP], and ICMP). These features are used in the detection process, so the switch needs to monitor them all. However, virtually any protocol field can be matched and counted, as in classic OpenFlow, providing a high degree of flexibility in designing the protection solution.

Several methods already exist in SDN to monitor traffic. In the following, we will list these approaches, 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* (eg, destination IP addresses) and counting how many times they appear during a time interval. On the contrary, *StateSec* makes the tracking of *traffic features* quick and efficient, by exploiting the stand-alone reconfiguration of state tables offered by stateful in-switch processing.

3.1.1 | Native OpenFlow monitoring

OpenFlow protocol defines the messages exchanged between the controller and a switch.¹³ The controller can natively gather the content of flow tables, including packet and byte counters associated with 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 (eg, 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 (eg, 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 the forwarding logic.

3.1.2 | sFlow sampling

sFlow is a well-known monitoring approach relying on packet sampling.¹¹ An sFlow agent integrated into the switch is in charge of both sampling packets at a configurable rate¹⁴ 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 with the native OpenFlow approach, since the monitoring functionality is totally decoupled from forwarding; ie, it is not necessary to add flow rules to ensure both forwarding and monitoring of flows. On the basis of 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,¹⁵ as we will show in Section 4. 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 trade-off between detection accuracy and overhead on the control channel.

3.1.3 | *StateSec's state-based monitoring*

StateSec 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*, eg, `src IP`, the controller initializes the following elements in the switch (cf Figure 3):

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

Finally, we extended *OpenState* to perform incremental state updates. Indeed, the original *OpenState* implementation offered a single function to update a state, namely, `set_state(newState)`, which requires to preconfigure the value of the *newState* value in the flow table. This function is adapted to FSMs where states and transitions are known in advance. For traffic monitoring applications, 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 with 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.[‡]

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.

3.2 | Detection

To validate the benefits of the in-switch monitoring offered by *StateSec*, we adapted a simple statistical-based algorithm.⁹ One advantage of this strategy is that it is directly applicable to the *traffic features* that are counted by the switch. Moreover, once EFSM will be available, the 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.

While in the following we focus mainly on the detection and mitigation of *flooding* and *port scan* attacks, it is worth noting that other strategies involving state variables can be engineered to detect different types of attack. For instance, *SYN-ACK Flood* attacks can be detected by maintaining a global state variable to count packets carrying the TCP SYN flag. In case of abnormal values, the mitigation might challenge the correctness of any suspicious TCP three-way handshake connection by dropping the first TCP SYN request and waiting for its retransmission (typically, attackers do not follow the

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

TCP state machine). Likewise, an IP fragmentation attack may be detected and avoided by associating each fragmented IP packet with its expected fragment offset and checking its consistency before any forwarding action.

3.2.1 | 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 used 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.

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

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.^{4,9} By identifying significant changes in the randomness of consecutive *traffic features* distributions, this statistical approach can detect several types of attacks, including (D)DoS, with better accuracy than methods based on volume metrics.⁹

Next, we correlate different *traffic features* entropy variations to identify the type of attack. Table 1 shows that monitoring the entropy of four *traffic features* distributions (dst IP, dst port, src IP, and src port) allows differentiating between multiple types of attacks. For instance, a significant decrease in entropy for both the dst IP and 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 used to detect spoofing attacks.

Figure 4 serves to illustrate the variations in entropy induced on the dst IP *traffic feature* while a UDP DoS flooding attack occurs around $t = 90$ seconds. 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, eg, servers providing very popular services. In this case, a

TABLE 1 By correlating the entropy variations of multiple features, it is possible to identify different types of attacks

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

Abbreviations: DDoS, distributed DoS; DoS, denial of service; ICMP, Internet Control Message Protocol.

FIGURE 4 Entropy variations for the dst IP *traffic feature* seen by a single switch: A User Datagram Protocol flooding attack happens between 90 and 95 s. This generates a significant drop in the entropy, which returns back to normal levels after detection and mitigation of the attack (traffic from malicious hosts is discarded)

hefty drop in entropy for the `dst IP` traffic feature might be related to certain properties of legitimate traffic and not to malicious actions (eg, unexpected flash crowds arising after major events). Hence, selecting a good detection threshold and implementing additional protection mechanisms are paramount to avoid detecting potential false positives.

3.2.2 | Detection thresholds and sensitivity

Sensitivity in detection is one of the key elements allowing to identify threats with a limited number of false positives. We used 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 ($m = 3$).

3.2.3 | Outliers protection

A key step consists in identifying the victim(s) (eg, target hosts and service ports) and, if possible, the attacker(s), (eg, hosts and source ports) that generate the attack. No matter the value chosen for m , to reduce further the impact of false positives, we enforce an additional mechanism to protect outliers. In effect, in any network, some legitimate hosts (eg, IP addresses) or services (eg, ports) are much more solicited than others (eg, 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 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 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 (eg, IP addresses that are much more solicited than the others, in a nonentropy-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 $\mu_d \pm m \cdot \sigma_d$.

3.2.4 | Detection process primer

The detection process starts by computing the entropy of each monitored *traffic feature* (`src addr`, `src port`, `dst addr`, and `dst port`), based on the list of symbols and their associated counters. Then, the algorithm computes the upper and lower thresholds, evaluating 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 1), and the mitigation process takes place consequently.

3.3 | Mitigation

Finally, mitigation actions are elaborated to protect legitimate users. Once a violation is detected, new flow rules are installed into the switch with a high priority to match suspicious packets. The effectiveness 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 via standard OpenFlow functionalities. Existing actions allow to drop, queue, prioritize, blackhole, or even forward traffic towards an intrusion detection system (IDS). We can

FIGURE 5 Test bed topology: Host *h1* replays legitimate traffic from the trace “BigFlows”¹⁷ using *tcpreplay*. Each attacker host generates a denial of service (DoS) attack towards the HTTP/FTP server

also imagine many more complex actions: For instance, suspicious traffic could be forwarded towards a deep packet inspection device (or virtual network function) running somewhere in the network for a further analysis conditioning a final remediation decision.

4 | STATESEC'S EVALUATION

We evaluated *StateSec*'s performance in terms of false positives, detection accuracy, control plane overhead, and memory consumption in a controlled test bed. Since the core contribution relates to the efficiency of the monitoring phase, we have compared *StateSec*'s results with those obtained with the same entropy-based detection strategy coupled with *sFlow* for monitoring. In particular, we identified the existing trade-off between the cost of a precise monitoring and the detection efficiency. We also highlighted the relation between the entropy and line rate of the traffic seen at the switch and the memory consumption, the control plane overhead, and the number of states to be stored at the switch. These measures proved crucial to assess an important number of hypotheses related to the suitability of our proposal in real deployments. Finally, we present the take away that were learnt by deploying *StateSec* into a real network.

4.1 | Experimental test bed setup

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

Moreover, on the basis of the test bed setup, we have developed also a *StateSec* virtual network function on the basis of the microservice architecture and running on Docker containers. This type of packaging is much easier to deploy and integrate as network function in OpenStack-like Network Function Virtualization [NFV] infrastructures. The Docker-based test bed has also been showcased during several live demo running on top of ordinary workstations.¹⁸

[§]We used OVS for experiments with *sFlow* because this software switch can integrate an *sFlow* agent while *ofsoftswitch13* cannot.

4.2 | Scenario and parameters

In VM-based experiments, host *h1* replays legitimate traffic taken either from the “BigFlows” trace or from synthetic traces with fixed entropy. By using the `tcpreplay` tool, this approach makes it possible to mimic a much more complex topology than the one presented in Figure 5.

BigFlows trace: The “BigFlows” trace (available at previous literature¹⁷) contains 50 000 packets and has been chosen to include a significant number of hosts (679) that exchange diverse types of legitimate traffic (9605, 40 249, and 118 packets for UDP, TCP, and ICMP, respectively); the trace captures a real network traffic on a busy private network’s access point to the Internet. The trace is indefinitely replayed at the speed of 3.3 Mb/s.

Fixed entropy traces: We crafted several synthetic traces having fixed entropy values using the Python library `scapy`. As in the “BigFlows” trace, each synthetic trace is composed of $N = 50000$ packets all of them having the same size (1024 B), with fixed values for `src IP`, `dst IP`, and `src port`. The only variable feature is the `dst port`, which can take values in $[1 \dots N]$ following a geometric distribution having a target normalized entropy.

In effect, we can derive the entropy of a geometric distribution with $p_{\text{geo}}(X = k) = (1 - p)^{k-1} \cdot p$, where $k \in \mathbb{Z}^+$ as follows:

$$\begin{aligned} H_{\text{geo}}(X) &= -\sum_{k=1}^{+\infty} (1-p)^{k-1} \cdot p \cdot \log_2((1-p)^{k-1} \cdot p) \\ &= -p \cdot \log_2(p) \sum_{k=1}^{+\infty} (1-p)^{k-1} - p \cdot \log_2(1-p) \sum_{k=1}^{+\infty} (k-1)(1-p)^{k-1} \\ &= -\log_2(p) - \frac{(1-p)\log_2(1-p)}{p}. \end{aligned} \quad (2)$$

It is then possible to generate a geometric distribution having the desired *normalized* entropy by fixing the number of packets N that we want to generate (to have the normalization parameter) and then computing the value of the success probability p that corresponds to the desired entropy. By extracting the `dst port` from this probability distribution, we can create a trace having the desired entropy.

Detection parameters: To test the detection capabilities of *StateSec*, the attacker(s) generate a (D)DoS flooding attack towards the server listening on ports 69 (TFTP) and/or 80 (HTTP), by using the tool¹⁹ *hping3* (1 pkt/1200 μ s). 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. For each set of parameters, we ran the experiments 30 times to smooth out effects due to the particular trace configuration.

We evaluate the performance of *StateSec* and *sFlow* for different values of the following 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, and 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 (for both *StateSec* and *sFlow*), as described in Section 3.2. A fine tuning of this parameter is required 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 rate¹⁴ is configured with values ranging from 1 (high—a sample for each packet) to 100 (low—a sample each 100 packets) in our experiments.

4.3 | DoS experiments results

4.3.1 | False positives

The fine tuning of detection thresholds is particularly critical to reduce as much as possible the number of false positives, thus preventing artificial DoS to legitimate users while preserving sensitivity to real attacks.

We analyze the impact of the detection sensitivity value m on the amount of false positives at stationary regime without attacks, by replaying the “BigFlows” traffic trace and counting (false) detections. Figure 6 gives an idea of the values at stake (note the logarithmic y-axis). As a result, 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. Figure 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

FIGURE 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

FIGURE 7 Control plane channel load during one Time Window

(*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 an increased control overhead.

4.3.2 | Control plane load

By performing advanced packet processing (in this case the counting of packet features) directly on the data plane, *StateSec* offers a very compact way of transferring information to controller, reducing the overhead due to control traffic.

We prove this observation by depicting in Figure 7 the average traffic flowing on the control plane channel during one Time Window. The overhead for *StateSec* includes any message exchanged to gather counters from the state tables, whose size depends on the variety of traffic flowing through the switch. Instead, with *sFlow*, the overhead includes the packet samples sent from the agent to the collector. In this case, the control traffic depends rather on the data traffic throughput at the switch. We note that with frequent sampling (eg, sampling rate set to 1), *sFlow* overloads the control plane. This is the price to pay for using *sFlow* at the same precision level offered by *StateSec*, which in turns provides always exact counters. *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 (eg, a sample every 50 or 100 packets) and accepting a lowered detection performance, *StateSec* results more efficient: It generates less overhead maintaining very precise monitoring information.

Also, to understand better the connection between the profile of the traffic flowing through the switch and the load on the control plane, we depict in Figure 8 the relationship between entropy and control plane load. In this analysis, we made use of the synthetic traces with fixed entropy. To isolate the effects of entropy on control plane load, we focus the analysis on the single varying feature of the traces (`dst port`). Figure 8 displays both the control plane load due to the retrieval at the controller of the stateful table (with `dst port` statistics) and the number of states stored at the switch in the same table during one Time Window (fixed at 5 s).

FIGURE 8 Average control plane channel load and number of states during one Time Window. The single `dst port` feature is considered. Traces are replayed at 8 Mbps

FIGURE 9 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 amount of states to take care of at the switch depends on the entropy of traffic—by fixing the entropy levels, their number depends on the amount of different destination ports seen at the switch. Low entropy levels mean that most of the traffic has the same destination ports, so low number of symbols with high counters, while high levels of entropy imply a high variability in the destination port, so high number of symbols having, in general, low counters. It is important to note that by increasing the entropy of traffic, the number of states to take care increases exponentially. This characteristic has to be taken into account when deploying *StateSec* in real high-speed networks, because the switch must be able to store a large amount of states inside its stateful tables. Traffic exchanged on the control channel presents a similar shape, only with a steeper slope. This stems from the fact that state tables are exchanged using a TCP connection, which inevitably adds some overhead. The effect of these inefficiencies is particularly evident for larger values of entropy.

4.3.3 | Detection rate

By feeding the detection algorithm with always accurate counters, *StateSec* shows the most consistent results in terms of accuracy and timely detection of possible menaces.

Ultimately, the detection rate is the most meaningful parameter for any credible 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, Figure 9 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. We ran the experiments with a fixed initial time (26 s) of the (D)DoS attack to highlight the particular interactions between the sampling rate value and the detection accuracy.

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 detection 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 (eg, compare 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 Figure 9 for Time Window 5 s and sampling rate 50).

4.3.4 | Memory consumption of the software switch

To gain insights on the capability to run *StateSec* over real-world deployment, we measured the actual memory requirements of the software switch during the monitoring phase. This analysis is even more meaningful with the use of a high-performance hardware platform as the target for the software switch. We identified the main factors driving the memory consumption and the order of magnitude required for running the software switch. We used the *Valgrind Massif* tool to profile precisely the RAM use (both stack and heap allocations) of the switch when running on the VM-based test bed.

In general, for the *ofsoftswitch13* used as base switch, we have two main processes running: *ofdatapath* and *ofprotocol*. The *ofdatapath* process is the user space implementation of an OpenFlow-compliant datapath. It monitors two or more network interfaces, forwarding packets between them according to the entries in the flow table that it maintains. It is also responsible for the stateful processing as explained in Section 3.1. *Ofprotocol* is the process in charge of handling the communications between the local datapath and the remote controller. We found that most of the memory resource consumption in our system is attributable to the *ofdatapath* process and the line rate seen at the switch.

Figure 10 depicts the evolution of the allocated (heap and stack) memory to run the switch in the VM-test bed for different line rates of the same “BigFlows” traffic trace. As hinted by the figure, when the process is launched, we assist to a rapid increase in memory consumption—around 11 MB for any line rate under test. From this point on, further increases in memory consumption depend to a large extent on the supported rate and on the need to store and update the state values in the state tables. Higher line rates translate into an increased number of states to maintain simultaneously, implying that this value constitutes the principal responsible for the differences existing in memory consumption. We tested also the impact of the number of activated stateful stages on aggregate memory consumption, finding it less critical. Table 2 represents the maximum memory consumption of the switch for different number of stateful tables activated. We discovered that, by fixing the line rate, the impact of adding a state table is not so decisive. Although the number of state tables slightly increases the memory footprint, it does not have such a visible effect as the line rate.

FIGURE 10 Evolution of switch memory use for different line rates

TABLE 2 Memory consumption vs number of stateful stages

No. of stateful tables	1	2	3	4	5	6	7
Memory consumption, MB	13.60	13.79	13.89	14.01	14.17	14.19	14.20

Note. Line rate set at 100 Mbps.

The memory footprint of *StateSec* is relatively small compared with the resources available on board of general purpose servers and cloud platforms (typically proposing several GB of memory). On the other hand, the values at stake must be carefully evaluated when the target is a high-performance hardware platform, like the NetFPGA board or custom ASICs, typically providing much lower amount of memory to use as forwarding and state table.

4.4 | 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 with 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, ie, it is generated by a number of attackers comparable with the number of hosts sending legitimate traffic, *StateSec* still can detect the attack without being able always to identify attackers. To evaluate this, we replayed the same traffic trace as for the DoS evaluation, this time using 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.²⁰ 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 victim's 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 evaluations, we considered $n = 4$). If there is a significant decrease in the current entropy value compared with 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 three 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/10 000 μ s). Figure 11 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.

4.5 | *StateSec* deployed over a real network

In the context of the H2020 BEBA project,²¹ we had the possibility to deploy and run *StateSec* on one end point of the link connecting Prague (CZ) to Amsterdam (NL) and belonging to the academic network operated by CESNET (the operator of the network infrastructure dedicated to science, research, and education in Czech Republic) for more than one day. The network has more than 400 000 users and the line rate of the link under investigation is up to 2 Gbps. Traffic was mirrored

FIGURE 11 Slow distributed denial of service detection rates for a Time Window of 5 s and $n = 4$

FIGURE 12 Average entropy values detected on the CESNET academic network for one day of deployment

at the switch and anonymized to guarantee the privacy of data. Because of traffic mirroring, *StateSec* could not directly intervene on the real network, but it was capable of monitoring all the traffic flowing from/to the end point, logging the measured entropy levels of traffic. Retrieved data proved then very useful for the analysis of the behavior of entropy in a real scenario.

Figure 12 presents the logged entropies for the four observed features (*src IP*, *dst IP*, *src port*, and *dst port*) during one day of monitoring. Looking at the figure, it is evident that the time-of-day has a large influence on the entropy values. In general, at nighttime, there is a reduction in the average traffic levels and the variability of such traffic increases lowering thus the entropy value. This is directly visible on the entropy values. Also, the variability of entropy is much larger during nighttime for three features out of four (the exception being the *dst port* feature). Another aspect that has to be taken into account is the average value of the entropy, which is always higher for IP address (source/destination) than ports (source/destination).

A simple way to exploit these results during the deployment of *StateSec* to improve the detection algorithm is to use different values of the sensitivity parameter m depending on the time-of-day. For instance, during daytime, the sensitivity parameter should be increased, while relaxed during nighttime following the pattern of Figure 12. Also, the sensitivity parameter m used for IP addresses and ports should be different and independent.

5 | RELATED WORK

Different types of DDoS attacks exist.²² Their impact can be significant: They are able to generate such a huge amount of traffic (up to 600 GB/s) causing targets to crash and the associated service(s) to become partially or totally unavailable. Detecting and mitigating DDoS attacks are 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.²⁰

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 (eg, related to the flow tables exhaustion at switches and the controller overload), focusing on the protection of the network itself.^{23,24} Others deal more with the protection of communication end points (user terminals and application servers).^{4,25-29} Similarly, in this paper, we focus on the protection of the communication end points 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 (eg, OpenFlow¹³). 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 describes 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.²⁸ 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 (eg, throughput decrease and latency increase). Also, the logical centralization at the network controller of management could be catastrophic when the controller is saturated with attack traffic.⁶ Another option is to replicate the traffic towards an IDS.²⁶ Threat identification can be performed by correlating IDS alerts through attack graphs at the controller. Simpler yet powerful options are to use statistical tests³⁰ or entropy-based algorithms for anomaly detection.^{9,31}

Regardless of the detection strategy used, 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.⁴ Performance can be improved by using the sampling techniques of sFlow.¹¹ However, sFlow must be carefully configured because the gains on the control plane load trade off the monitoring precision.¹⁵ 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.^{25,29,32,33} An emerging approach in this sense is stateful SDN,⁷ which implies faster reaction times and less load over the control plane.³⁴ *StateSec* implements a stateful monitoring process running into the switch to gather very precise information and feed an entropy-based detection algorithm.

The set of packet matching rules and the associated actions can be defined programmatically via the P4 language,³⁵ which is then compiled to a target platform—either a hardware or software implementation of the SDN switch. In principle, P4 allows describing stateful functions; nevertheless, limitations in its concurrency model reduce their field of applicability.³⁶ For instance, PISCES represents a high-performance software switch implementation built with the P4 language.³⁷ PISCES does not support any stateful processing, thus not being able to implement the functionalities required by *StateSec*. Another approach, named SoftFlow, addresses stateful processing by extending OVS and integrating arbitrarily complex stateful functions.³⁸ Differently from OpenState, which offloads the stateful operations directly to the switch, a SoftFlow action is a black box approach, acting similarly to a VM attached to the switch.

6 | CONCLUSION AND PERSPECTIVES

In this paper, we described *StateSec*, a novel approach based on stateful SDN to protect communication end points from DDoS attacks. *StateSec* relies on advanced in-switch processing capabilities to delegate the traffic monitoring phase to the switch, thus reducing the control plane load and improving detection accuracy. We evaluated its performance on both a controlled test bed and a real-world deployment. Comparisons against other state-of-the-art SDN-based solutions have confirmed that *StateSec* is efficient in terms of control plane occupation, guaranteeing better monitoring and detection accuracy. We also investigated the influence that line rate and traffic entropy values have on the performance of the switch. Furthermore, we highlighted a number of preliminary results extracted from the real-world deployment.

Our work will continue in the following directions. Firstly, we will explore the integration of *StateSec* with the extended in-switch processing abstraction proposed by EFSM, to include a better version of the entropy-based calculation into the switch and thus offload as much as possible the controller from the heavy task of traffic monitoring and anomaly detection. Secondly, we will continue to deploy the *StateSec* in even more realistic scenarios to validate further the capabilities of the stateful approach. Furthermore, *StateSec* is very flexible in terms of monitored feature and detection strategies making it a perfect candidate to be adapted to different threats (eg, SYN-Flood attacks).

ACKNOWLEDGMENTS

This is an extended version of the work appeared³⁹ in IEEE NetSoft 2017. The authors thank Victor Pus and Pavel Benacek from CESNET for their assistance during the real-world deployment. This work has been partly funded by the EU in the context of the H2020 BEBA project (grant agreement 644122) and Celtic-Plus SENDATE-Tandem.

ORCID

Filippo Rebecchi <http://orcid.org/0000-0002-3946-9047>

REFERENCES

1. Khalimonenko A, Kupreev O, Ilganaev K. Kaspersky lab: DDoS attacks in Q3 2017. <https://securelist.com/ddos-attacks-in-q3-2017/83041/>; 2017.
2. Khandelwal S. World's largest 1 Tbps DDoS attack launched from 152,000 hacked smart devices. 2016. <http://thehackernews.com/2016/09/ddos-attack-iot.html>. Accessed on 26/07/2018.
3. York K. Dyn statement on 10/21/2016 DDoS attack. 2016. <http://dyn.com/blog/dyn-statement-on-10212016-ddos-attack/>. Accessed on 26/07/2018.
4. Giotis K, Argyropoulos C, Androulidakis G, Kalogeras D, Maglaris V. Combining OpenFlow and sFlow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments. *Comput Networks*. 2014;62:122-136.
5. RADWARE. DefenseFlow—SDN based network DDoS, application DoS and APT protection. In: ONS'14; 2014. <http://opennetsummit.org/archives/mar14/site/pdf/2014/sdn-idol/Radware-SDN-Idol-Proposal.pdf>. Accessed on 26/07/2018.
6. Kalkan K, Gur G, Alagoz F. Defense mechanisms against DDoS attacks in SDN environment. *IEEE Commun Mag*. 2017;55(9):175-179.
7. Bianchi G, Bonola M, Capone A, Cascone C. OpenState: programming platform-independent stateful OpenFlow applications inside the switch. *ACM SIGCOMM Computer Communication Review*. 2014;44(2):44-51.
8. OpenState SDN project. 2016. <http://openstate-sdn.org>. Accessed on 26/07/2018.
9. Lakhina A, Crowella M, Diot C. Mining anomalies using traffic feature distributions. *ACM SIGCOMM Comput Commun Rev*. 2005;35(4):217-228.
10. Bianchi G, Bonola M, Pontarelli S, Sanvito D, Capone A, Cascone C. Open packet processor: a programmable architecture for wire speed platform-independent stateful in-network processing. 2016. <http://arxiv.org/pdf/1605.01977.pdf>.
11. Phaal P. sFlow specification version 5. 2014. http://www.sflow.org/sflow_version_5.txt. Accessed on 26/07/2018.
12. Bonelli N, Prociassi G, Sanvito D, Bifulco R. The acceleration of OfSoftSwitch. In: IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN); 2017; Berlin, Germany. 1-6.
13. Mckeown N, Anderson T, Balakrishnan H, et al. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Comput Commun Rev*. 2008;38(2):69.
14. sFlow blog. sFlow sampling rates. 2009. <http://blog.sflow.com/2009/06/sampling-rates.html>. Accessed on 26/07/2018.
15. Mann V, Vishnoi A, Bidkar S. Living on the edge: monitoring network flows at the edge in cloud data centers. In: Fifth International Conference on Communication Systems and Networks (COMSNETS); 2013; Bangalore, India. 1-9.
16. Mininet Team. Mininet. 2016. <http://mininet.org>. Accessed on 26/07/2018.
17. Tcpreplay sample captures. <http://tcpreplay.appneta.com/wiki/captures.html>. Accessed on 26/07/2018.
18. Rebecchi F, Boite J, Nardin P-A, Bouet M, Conan V. Traffic monitoring and DDoS detection using stateful SDN. In: IEEE Conference on Network Softwarization (NetSoft); 2017; Bologna, Italy. 1-2.
19. Sanfilippo S. Hping. 2006. <http://www.hping.org/>. Accessed on 26/07/2018.
20. Park J, Iwai K, Tanaka H, Kurokawa T. Analysis of slow read DoS attack. In: International Symposium on Information Theory and its Applications, (ISITA); 2014; Melbourne, Australia. 60-64.
21. BEBA project. 2016. <http://www.beba-project.eu>. Accessed on 26/07/2018.
22. Douligeris C, Mitrokotsa A. DDoS attacks and defense mechanisms: classification and state-of-the-art. *Comput Networks*. 2004;44(5):643-666.
23. Dharma NG, Muthohar MF, Prayuda JDA, Priagung K, Choi D. Time-based DDoS detection and mitigation for SDN controller. In: 2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS); 2015; Busan, South Korea. 550-553.
24. Dhawan M, Poddar R, Mahajan K, Mann V. SPHINX: detecting security attacks in software-defined networks. In: 22nd Annual Network and Distributed System Security Symposium (NDSS); 2015; San Diego, California, USA. 8-11.
25. Chin T, Mountrouidou X, Li X, Xiong K. An SDN-supported collaborative approach for DDoS flooding detection and containment. In: IEEE Military Communications Conference (MILCOM); 2015; Tampa, Florida, USA. 659-664.
26. Chung C-J, Khatkar P, Xing T, Lee J, Huang D. NICE: network intrusion detection and countermeasure selection in virtual network systems. *IEEE Trans Dependable Secure Comput*. 2013;10(4):198-211.
27. Lin Y-H, Kuo J-J, Yang D-N, Chen W-T. A cost-effective shuffling-based defense against HTTP DDoS attacks with SDN/NFV. In: IEEE International Conference on Communications (ICC); 2017; Paris, France. 1-7.
28. Oktian Y, Lee S, Lee H. Mitigating denial of service (DoS) attacks in OpenFlow networks. In: International Conference on Information and Communication Technology Convergence, (ICTC); 2014; Busan, South Korea. 325-330.
29. Shin S, Yegneswaran V, Porras P, Gu G. AVANT-GUARD: scalable and vigilant switch flow management in software-defined networks. In: ACM Conference on Computer and Communications Security, (CCS); 2013; Berlin, Germany. 413-424.
30. Soldo F, Metwally A. Traffic anomaly detection based on the IP size distribution. In: Proceedings of the IEEE INFOCOM 2012; 2012; Orlando, FL, USA. 2005-2013.

31. Huang X, Du X, Song B. An effective DDoS defense scheme for SDN. In: IEEE International Conference on Communications (ICC); 2017; Paris, France. 1-6.
32. Afek Y, Bremner-Barr A, Shafir L. Network anti-spoofing with SDN data plane. In: IEEE INFOCOM 2017 - IEEE Conference on Computer Communications; 2017; Atlanta, GA, USA. 1-9.
33. Liu Z, Manousis A, Vorsanger G, Sekar V, Braverman V. One sketch to rule them all: rethinking network flow monitoring with UnivMon. In: Proceedings of the ACM SIGCOMM 2016 Conference; 2016; Florianopolis, Brazil. 101-114.
34. Bifulco R, Boite J, Bouet M, Schneider F. Improving SDN with InSPIred switches. In: Proceedings of the Second ACM Symposium on SDN Research, ACM SOSR; 2016; New York, NY, USA. 11:1-11:12.
35. Bosshart P, Daly D, Gibb G, et al. P4: programming protocol-independent packet processors. *ACM SIGCOMM Comput Commun Rev.* 2014;44(3):87-95.
36. Petrucci L, Bonelli N, Bonola M, et al. Towards a stateful forwarding abstraction to implement scalable network functions in software and hardware. *arXiv preprint arXiv:1611.02853*. 2016.
37. Shahbaz M, Choi S, Pfaff B, et al. Pisces: a programmable, protocol-independent software switch. In: Proceedings of the ACM SIGCOMM 2016 Conference; 2016; Florianopolis, Brazil. 525-538.
38. Jackson EJ, Walls M, Panda A, et al. SoftFlow: a middlebox architecture for Open vSwitch. In: USENIX Annual Technical Conference; 2016; Denver, CO, USA. 15-28.
39. Boite J, Nardin P-A, Rebecchi F, Bouet M, Conan V. StateSec: stateful monitoring for DDoS protection in software defined networks. In: IEEE Conference on Network Softwarization (NetSoft); 2017; Bologna, Italy. 1-9.

Filippo Rebecchi received BSc and MSc in Telecommunications Engineering from University of Parma, Italy. He then received the PhD degree from Sorbonne University (formerly UPMC—Paris VI) in 2015 working on mobile data offloading solutions. He works as an R&D engineer with Thales in the Networking Laboratory, Advanced Studies Department. His current research interests include mobile next-generation networks, security, and network virtualization.

Julien Boite obtained in 2009 both Engineering and Master degrees in Information Systems and Telecommunications after a 3-year course at the University of Technology of Troyes (France). He then achieved a PhD in a research laboratory at Thales Communications & Security (Paris, France), dealing with adaptive forwarding of flows in heterogeneous networks. Employed as a research engineer at Thales Communications & Security after his thesis, he mainly worked on mobile and mesh networks (LTE, offloading), and more recently on software-defined networking (SDN), contributing to international research projects in those fields. He moved to Toulouse (France) in 2016 to work as a system and network architect at Sigfox, a French startup deploying a worldwide network dedicated to Internet of Things (IoT) use.

Pierre-Alexis Nardin graduated from both the university and an engineering school (ENSEIRB-MATMECA) of Bordeaux with Master degrees in Telecommunications and Networking. He has worked on hardware-software isolation against DDoS attacks with the University of Arkansas and on detection of DDoS attacks on SDN in Thales. He then worked for 2 years as a Security Engineer in Thales's main CSOC and is now working as a cyber security expert for Thales by being a part of the CERT-IST team.

Mathieu Bouet received the PhD degree in Computer Science and the Habilitation degree from Sorbonne University (formerly UPMC—Paris VI) in 2009 and 2017, respectively. He is a Research Expert in networking and communications with Thales, France, where he currently manages research activities on network softwarization with the Networking Laboratory, Advanced Studies Department. His research interests are mainly focused on network virtualization and network optimization.

Vania Conan received the Engineering and PhD degrees in Computer Science from Mines ParisTech in 1990 and 1996, respectively, and the Habilitation degree from Sorbonne University, Paris in 2012. He is a Senior Research Expert in networking and communications with Thales, France. He is currently the Head of the Networking Laboratory, Advanced Studies Department in Thales. He has been conducting research in the fields of software-defined communications and wireless networking. He has published over 100 international conference and journal papers and holds 10 patents in networking technologies. His current research topics include mobile network protocols and virtualized network design.

How to cite this article: Rebecchi F, Boite J, Nardin P-A, Bouet M, Conan V. DDoS protection with stateful software-defined networking. *Int J Network Mgmt*. 2019;29:e2042. <https://doi.org/10.1002/nem.2042>