INFAS: In-Network Flow mAnagement Scheme for SDN Control Plane Protection

Tao Li[1], Hani Salah[1], Xin Ding[1], Thorsten Strufe[1], Frank H. P. Fitzek[1], and Silvia Santini[2]

[1]Technical University of Dresden, Germany
[2]Universitá della Svizzera Italiana (USI), Switzerland
{*tao.li, hani.salah, xin.ding, thorsten.strufe, frank.fitzek*}@tu-dresden.de, silvia.santini@usi.ch

***Abstract*—The work that we present in this paper is motivated by a systematic vulnerability of SDN, a current technology that is expected to dominate the Internet. In particular, we focus on the Control Plane Saturation (CPS) attack, a very harmful, yet easy to implement, DoS attack. In CPS, the adversary generates a massive amount of flow packets that will not match switches' flow rules. As a result, the switches flood the control channels and the controller with malicious control packets. Previously proposed solutions mainly rely on the controller-side detection and filtering, thus still consume the control plane bandwidth resources and cannot achieve quick response due to the switch-controller delay.**

We present INFAS, a system that runs on commodity servers installed near network devices, for protecting SDN against CPS. The switches send flow packets that do not match concrete flow rules in their flow tables to INFAS, which is tasked to analyze the packets and to subsequently decide on sending them back to the switches or not. This results in reducing the number of generated control packets by up to 80%, which we show through extensive evaluations.

***Index Terms*—Software-Defined Networking; Security; Control Plane Saturation; Denial-of-Service; Flow Management**

I. Introduction

Traditional IP networks, being based on distributed protocols running inside network devices, are complex and hard to manage. In particular, administrators of these networks convey network-wide policies by configuring each network device individually using vendor-specified commands. They also need to reconfigure the devices in case of faults and changes. Furthermore, each device in the network is tasked both (i) to take switching (or routing)[1] decisions in the control plane and (ii) to forward incoming traffic accordingly in the data plane. Such coupling between the control plane and the data plane inside the network devices reduces flexibility, and it handicaps large-scale deployments and evaluations of new network protocols and architectures.

Software Defined Networking (SDN) has been gaining popularity over the past few years, and it is widely considered today as a key player in modern and future networking. SDN promises to address the aforementioned limitations of traditional IP networks by decoupling the control plane from the data plane. In particular, the switches in SDN become dummy devices being responsible only for forwarding traffic according to the decisions taken by a (logically) centralized network controller. Such a design simplifies network management, and it also facilitates innovation and evolution [1].

While SDN benefits can be mainly attributed to the data-control planes separation, the same feature can represent a systematic vulnerability [2]. A very harmful, yet easy to implement, Denial-of-Service (DoS) attack that misuses such a separation is the *Control Plane Saturation (CPS) attack*. In CPS, the adversary exploits the fact that SDN switches send control packets to the controller whenever their flow tables miss rules matching incoming flow packets. In particular, the adversary generates a large number of such packets rapidly. This will trigger the switches to flood both the controller and the control channels with malicious control packets. The malicious control packets will consume the computation resources of the controller as well as the control plane bandwidth. As a consequence, the communication between the switches and the controller will be disrupted, and the legitimate flow packets will not be (timely) handled.

There has been a considerable amount of research on CPS and similar attacks. The proposed solutions are commonly implemented as protection modules at the controller side (e.g. see [3]–[5] and the references therein). The protection module receives incoming control packets, before other controller modules, and analyzes them to identify potential adversaries (those generating flow packets incurring large amounts of control packets). Once potential adversaries are identified, the protection module carries out some mitigation actions (e.g. installing flow rules in the switches to block traffic coming from the identified hosts [4]). While these solutions are effective in protecting the computational resources of the controller, they do not alleviate the bandwidth saturation in the control plane. This is because all the control packets still have to be sent from the switches to the controller till they are handled by the protection module.

Our main contribution in this paper is **I**n-**N**etwork **F**low m**A**nagement **S**cheme (INFAS), an efficient solution for protecting SDN networks against CPS. INFAS addresses the aforementioned drawbacks of prior solutions by a *self-contained in-network* module to handle the malicious data flows from a source, before they saturate the control plane. INFAS is designed as a network function running on com-

[1] From now onward, the term *switch* will represent both types of devices.

978-3-903176-15-7 © 2019 IFIP

modity servers installed near the switches. Such in-network resources are already available in the current, rapidly increasing, networks that support Network Functionality Virtualization (NFV). The switches send flow packets that do not match any rule in their flow tables firstly to INFAS for evaluation. INFAS, in turn, employs a novel threshold-based algorithm to determine the probability of allowing the received flow packets to return to the switches and trigger the corresponding control packets. To reduce the delay caused by this additional processing step, we build INFAS using the Data Plane Development Kit (DPDK) [6].

We evaluate the effectiveness of INFAS extensively through representative prototype and network emulations. The results show the high effectiveness of INFAS. In particular, INFAS reduces the generation of malicious control packets by up to 80%, depending on the parameters configuration of our algorithm. In addition, compared to the approach of directly handling control packets, the switch throughput is improved with INFAS by about 26%.

The remainder of the paper is structured as follows: we give an overview of the CPS attacks and defence mechanisms on the SDN in Section II. Next, we describe INFAS and evaluate it in Section III and Section IV, respectively. Lastly, we conclude the paper in Section V.

II. Background and related work

In SDN, Denial-of-Service (DoS) attacks to the control plane mainly target the controller and control channels. In this paper, we deal with a specific DoS attack called *Control Plane Saturation (CPS)*. The basic principle of CPS is to generate a large number of flow packets that will not match any flow rule in the flow tables of the receiving switches. As a consequence, each unmatched flow packet results in two control packets: (i) `packet_in` message sent from the switch to the controller and (ii) `ofp_flow_mod` or `packet_out` messages sent in the opposite direction. Floods of these control packets, in turn, result in consuming the computational resources of the controller [4] as well as the control plane bandwidth. Subsequently, the legitimate flow packets will be either dropped or delayed [7].

The majority of the CPS defense mechanisms are controller-based, which consists of a mitigation module implemented as a controller application. For example, FloodDefender [3] is a network control framework for protecting the resources in the data and control planes. It implements a packet filtering module in the controller to identify the malicious flows based on the arrival rates of `packet_in` messages. SDN-Guard [8] is a controller application designed to mitigate CPS-like attacks. It manages the flow packets according to the information it receives from an intrusion detection system. These information includes the threat probability of each flow. SDN-Guard reroutes potentially malicious flow packets through the least utilized links. The mitigation module of FlowRanger [9] uses a trust management system to prioritize the incoming `packet_in` messages, and stores them in a queue. The higher the priority of the message, the faster it will delivered to other controller modules, such as the routing module.

The aforementioned controller-based solutions do not alleviate the bandwidth consumption caused by the exchanged control packets. That is, the control requests (i.e. `packet_in` messages) will be sent from the switches to the controller, and their responses will be sent oppositely, until the protection module deals with them. Some prior works also implement the in-network approach, to some extent. For instance, FloodGuard [10] employs symbolic executions to pre-generate flow rules to increase the responsiveness of the controller. It further uses in-network packet queues to cache all unmatched flow packets. FloodGuard translates the cached flow packets into control packets in a round-robin way based on protocol types, and sends them to a migration agent running inside the controller. AVANT-GUARD [11] aims to prevent TCP-based DoS attacks, using additional modules introduced into the design of the switch architecture. Its principle idea is to allow only the flow packets arriving from a source that can complete a TCP handshake to trigger `packet_in` messages.

The above discussed in-network solutions either still implement the mitigation logic in the controller (like [10], [12]), or are hard to implement (like [11] which requires to change the switch architecture). In contrast, INFAS implements the mitigation logic directly and locally in the in-network module, to reduce the control plane traffic under DoS. In addition, it is easy to implement, does not require to change the switch architecture, and can be easily integrated into NFV platforms.

III. Our Solution:INFAS

In this section, we describe INFAS, our solution for protecting SDN against the CPS and similar DoS attacks.

A. Architecture design

Fig. 1 depicts a SDN network deployed with INFAS. For each switch, we deploy an INFAS instance on a connected server. In this paper, we do not consider cooperation, thus there is no communication, among INFAS instances. Each INFAS instance consists of three components: (i) flow management module, (ii) query module, and (iii) switch statistic proxy. In the following, we describe each of these components, and how they interact with each other.

The *flow management module* accepts the flow packets that do not hit any flow rule. It includes an attack detection and mitigation algorithm performing analysis over statistics, collected from both unmatched flow packets and the query module. It determines the severity of control plane saturation caused by the packets from a flow source. Accordingly, the flow management module tunes action parameters for each suspicious entity, e.g. host or port, to drop portions of the corresponding unmatched flow packets. Other unmatched flow packets are considered legitimate, and they are sent back to the switch via another port and will trigger `packet_in` messages.

The *query module* is responsible for collecting the information that cannot be directly derived by the flow management

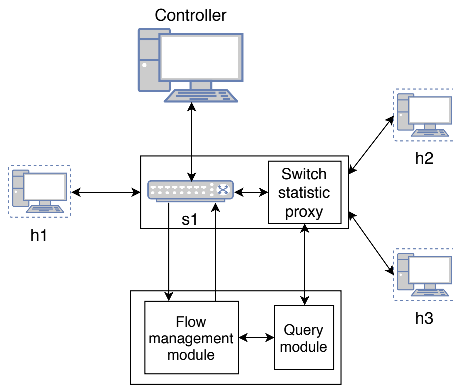


Fig. 1: System architecture of INFAS and testbed setup

module. These information include basic flows statistics, such as flow packet counters. The query module requests them, at regular time intervals, from the switch statistic proxy. Once the information is collected, the query module sends them to the flow management module. The flow management module is separated from the query module. That is, they work asynchronously, because the former is tasked to perform in-network packet processing at a high speed, while the latter involves slow I/O operations, like socket communication.

The *switch statistic proxy* is a small piece of code that runs in the switch to bridge its associated switch with the two other INFAS components. As described above, the switch statistic proxy receives inquiries from the query module asking for some statistics. To answer these inquiries, the switch statistic proxy runs basic switch commands, aggregates the returned results, and lastly sends the outputs back to the query module.

B. Flow rule design

To support the detection and mitigation algorithm, INFAS defines three categories of flow rules: (i) concrete flow rule, (ii) redirection flow rule, and (iii) monitoring flow rule. We illustrate the roles of these categories through an example. In the example, there is a switch `s1` with four ports: `port 0` is an egress port used to send flow packets to the flow management module, `port 1` is an ingress port used to receive data from the same module, `port 2` is connecting `s1` with a host `h1` holding the IP address 10.0.0.1, and `port 3` which we mention through the example. Table I shows four exemplary flow rules following INFAS design.

The *concrete flow rules* perform exact matching for the packet flows, to achieve the actual goal of some network control logic (e.g. routing). The controller is responsible for installing these rules in the switches' flow tables. In the example, `rule 1` is a concrete flow rule that specifies the output `port 3` for the flow packets having the source IP address 10.0.0.1 and the destination IP address 10.0.0.2. From now onward, we will use the terms *matched packets* and *unmatched packets* to respectively refer to the flow packets that match and those that do not match concrete flow rules.

The purpose of the *redirection flow rules* is to avoid sending `packet_in` messages from the switches to the controller in the case of unmatched packets. In the current INFAS design, unmatched packets are simply forwarded to the flow management module. `Rule 2` is an exemplary redirection flow rule that specifies the output `port 0` for the unmatched packets that arrive through `port 2` and are destined to 10.0.0.2. Note that it is possible to select a portion of unmatched packets using more specific matching fields, depending on the concrete flow rules. Redirection flow rules always have lower priority than concrete flow rules, which assures that flow packets first obey the network control logic.

The *monitoring flow rules* are intended to help to obtain basic statistics, like the number of matched packets received from a host or through a port. These rules are usually installed in the flow table (e.g. flow table 1) following the one containing concrete flow rules and redirection flow rules (e.g. flow table 0). In the example, `rule 3` is a monitoring flow rule that counts the total number of matched packets coming from `h1`, since all packets matching `rule 1` are forced to go through the flow rule table 1.

In the example, `rule 4` is the default flow rule. A flow packet that is permitted by INFAS to return to the switch will not match any flow rule belonging to the above three categories. With the default flow rule, it ultimately will incur a real table-miss event.

C. Flow management algorithm

The majority of DoS mitigation algorithms in SDN, which we are aware of, tend to clearly distinguish between malicious flows and legitimate flows. They subsequently block the sources of potentially malicious flows. A widely used approach is to use the amount of triggered control packets as a detection parameter. However, we argue that this approach can be inaccurate. This is because a large number of control packets can be attributed to legitimate flow packets originating from a source during a workload peak. Instead, we propose a threshold-based flow management algorithm that does not block a network entity completely.

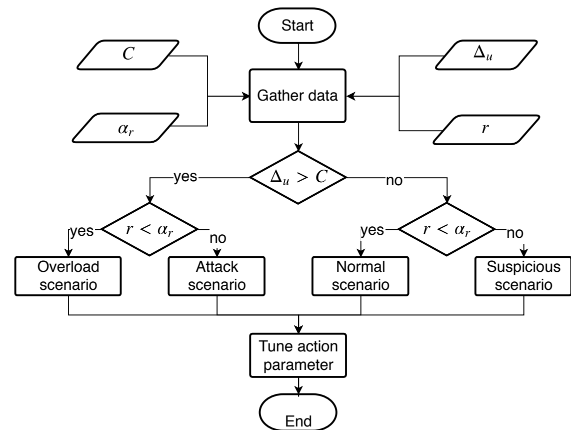


Fig. 2: INFAS flow management algorithem

As shown in Fig. 2, the algorithm identifies four different control plane's saturation severity levels. Accordingly, the al-

Flow rule	Category	Table ID	Source IP	Destination IP	Priority	In_port	Action
Rule 1	Concrete	0	10.0.0.1	10.0.0.2	2	2 (h1 – s1)	Output: 3 & GOTO Table 1
Rule 2	Redirection	0	*	*	1	2 (h1 – s1)	Output: 0
Rule 3	Monitoring	1	10.0.0.1	*	1	2 (h1 – s1)	None
Rule 4	Implicit	*	*	*	1	*	Controller

TABLE I: Exemplary INFAS flow rules

gorithm applies distinct mitigation strategies on incoming unmatched packets. The algorithm executes in a periodic manner, and it uses two input thresholds: (i) the `packet_in` budget C and (ii) the threshold of unmatched packets proportion α_r. The first threshold defines, within a time slot, the maximum number of flow packets permitted to trigger `packet_in` messages. It highly depends on the capacity of the controller and the expected number of networking entities sending packet flows. A simple method to determine the budget value is to divide the controller capacity among the network entities. The second threshold specifies the maximum allowed percentage of unmatched flow packets received from a host or port. To measure the unmatched packet proportion, the algorithm uses the unmatched and matched packet statistics, Δ_u and Δ_m, respectively collected from the flow management module and the query module. The unmatched packet proportion for a network entity is: $r = \Delta_u/(\Delta_u + \Delta_m)$.

The algorithm uses the above-described input values to calculate the *acceptance probability* p, i.e. the probability to return an unmatched packet to the switch. In principle, the more severe the saturation caused by a network entity, the smaller the acceptance probability p for the corresponding flow packets. In the following, we describe the four severity levels, and the corresponding categorization conditions and p values:

- *Normal case* ($\Delta_u < C$, $r < \alpha_r$): Here, the amount of unmatched packets is less than the `packet_in` budget, and the most of flow packets received from a network entity can hit the concrete flow rules. The algorithm allows all unmatched packets to return to the switch and trigger table-miss events. The acceptance probability p in this case is set to 1.
- *Suspicious case* ($\Delta_u < C$, $r > \alpha_r$): We consider this case as suspicious because these packets do not bring much workload to the control plane, although a relatively large portion of flow packets received from the network entity trigger table-miss events. In this case, the algorithm introduces a small penalty, according to which only a small portion of the unmatched packets are dropped: $p = 1 - tanh(r * 2)$. $tanh$ is a hyperbolic tangent function.
- *Overload case* ($\Delta_u > C$, $r < \alpha_r$): Under these conditions, the control plane is considered overloaded, because the number of unmatched packets exceeds the `packet_in` budget. Meanwhile, the majority of the flow packets can match the concrete flow rules. This can be interpreted as a workload peak. In this case, the algorithm simply regulates the rate of unmatched packets to be the same as the budget value: $p = \Delta_u/C$.
- *Attack case* ($\Delta_u > C$, $r > \alpha_r$): This is the most severe case. More precisely, the amount of unmatched packets exceeds the `packet_in` budget, and the network entity sends a large amount of flow packets that will trigger `packet_in` messages. This case is very likely caused by an attack. To mitigate the aggressive impact of the attack on the control plane, the algorithm dramatically decreases the acceptance probability: $p = \Delta_u/C*(1-tanh(r*10))$.

IV. Implementation and Evaluation

In this section, we detail our implementation and evaluation.

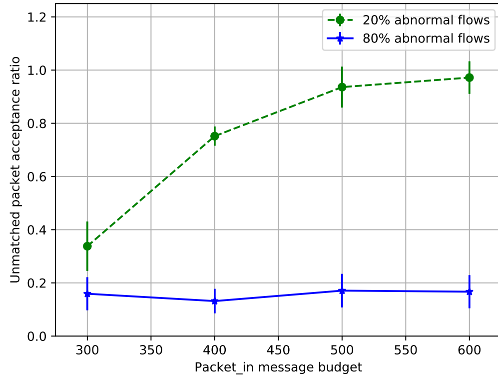
A. Prototype implementation

We implement the flow management module as a Data Plane Development Kit (DPDK) application. This implementation choice provides the required performance guarantees for fast in-network packet processing [13]. It also enables to integrate INFAS into other DPDK-based NFV platforms, like openNetVM [14], [15]. The query module is also implemented as a DPDK application. It exchanges information with the flow management module through a high-speed ring buffer provided by DPDK. It also communicates with the switch statistic proxy using standard TCP/IP sockets. Both the flow management module and the query module run on a Netgate DPDK box [16], containing a Quad Core Intel(R) Atom(TM) E3845 1.91 GHz CPU and 2GB RAM. We use one Open vSwtich (OvS) instance [17] as the SDN switch in the experiment. As for the switch statistic proxy, it is implemented as a Python script running on the same host as the OvS. To query the flow statstics, the switch statistic proxy interacts with the OvS using OvS-Python APIs [18].

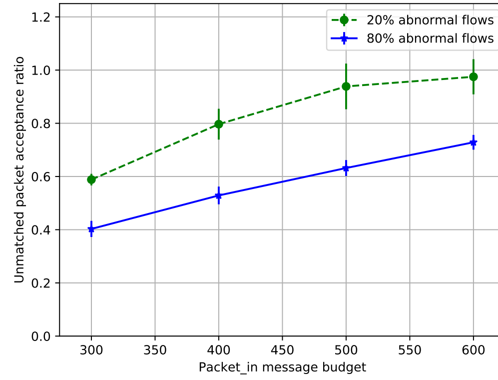
B. Evaluation setup

*1) **Testbed scenario**:* Our evaluations are based on a testbed emulating the functionality of a SDN-based server workload balancer. As illustrated in Fig. 1, the testbed consists of a Floodlight [19] controller, an OvS `s1` protected by an INFAS instance, and three hosts {`h1`, `h2`, `h3`} running as Docker containers [20]. The host `h1` has the IP address 192.168.0.1, which represents an external entity. The hosts `h2` and `h3` are internal servers assigned the IP addresses 10.0.0.2 and 10.0.0.3, respectively. To expose the service to the outside world, `h2` and `h3` also share an external IP address 192.168.0.20. The OvS and the three containers run on a machine equipped with a four-core Intel(R) Core(TM) i5-6500 CPU and 32GB RAM.

In the experiments, `h1` sends packet flows, from its socket ports, to different ports belonging to 192.168.0.20. The role of `s1` is to evenly distribute the flow packets from `h1` to the two servers {`h2`, `h3`}, by mapping their IP addresses and ports.

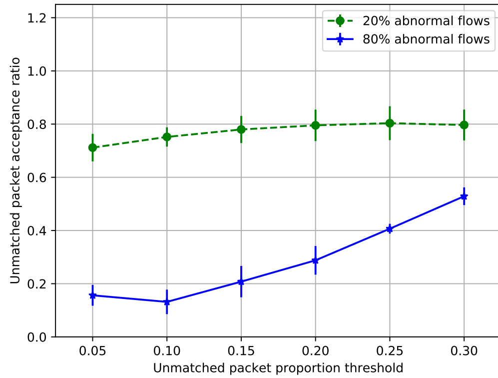


(a) Unmatched packet proportion threshold $\alpha_r = 0.1$

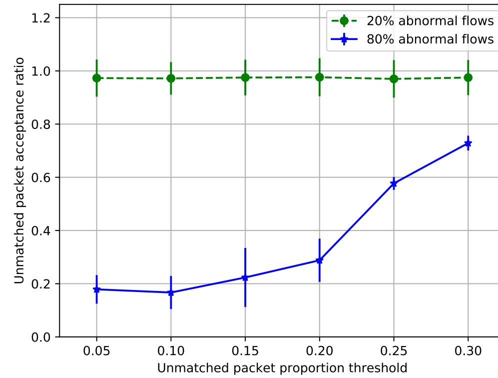


(b) Unmatched packet proportion threshold $\alpha_r = 0.3$

Fig. 3: INFAS performance under varying `packet_in` message budget C



(a) `packet_in` budget $C = 400$



(b) `packet_in` budget $C = 600$

Fig. 4: INFAS performance under varying unmatched packet proportion thresholds α_r

`s1` performs a four-tuple {Source IP, Source port, Destination IP, Destination port} matching. For example, when `h1` sends packets to 192.168.0.20:1, according to the concrete flow rule in `s1`, the destination IP address 192.168.0.20 is converted to an internal IP address (10.0.0.2 or 10.0.0.3), and the port number 1 is mapped to a new port number (e.g. 10). After this conversion, the packets are sent to the corresponding server. When an internal server replies with flow packets to `h1`, the source IP address is converted back to 192.168.0.20.

The above-described mapping procedure is managed by the controller. In particular, the controller decides which server and which port are mapped for a flow coming from `h1`, upon receiving the `packet_in` messages. For each flow, the controller installs flow rules specifying an output port, and rewrites the packet's destination IP address and destination port. Each concrete flow rule is configured with an idle timeout of two seconds, to reduce the number of concrete flow rules.

*2) **Launching the CPS attack**:* An aggressive CPS attack is emulated by a script running in `h1`. More precisely, the script generates a large number of unmatched packets from `h1` to 192.168.0.20 using 2000 different destination ports. As we already described in Section II, if the packet inter-arrival time of a flow exceeds the idle timeout, a large number of `packet_in` messages are generated. Particularly, we define two types of flows, namely *normal flows* and *abnormal flows*. The packet intervals of both flow types follow the Gaussian distribution, but the normal flow has a mean value of 0.5 and the abnormal flow has a mean value of 2.5. By varying the ratio between the abnormal and normal flows, we could emulate different severity of the CPS attacks. In the experiments, we call this parameter as the *abnormal flow percentage*.

*3) **Evaluation metrics and focused parameters**:* We use two system performance metrics: (i) the unmatched packet acceptance ratio β and (ii) the switch throughput. The first metric enables to evaluate the mitigation effectiveness of INFAS against the CPS attack. $\beta = A_c/A_i$, where A_c is the number of unmatched packets that finally trigger generation of `packet_in` messages, and A_i is the number of unmatched

packets forwarded to INFAS. As for the switch throughput, it is obtained by measuring the experienced bandwidth between two hosts connected with the protected switch. We use this metric to show that INFAS is able to reduce the workload of the protected switch. In order to evaluate the impact of the system parameters, we experiment with varying `packet_in` budget C values and unmatched packet proportion thresholds α_r.

C. Results

*1) **Unmatched packet acceptance ratio**:* We conduct two groups of experiments to measure the unmatched packet acceptance ratio. In the first group, we fix the unmatched packet proportion threshold $\alpha_r \in \{0.1, 0.3\}$, and vary the `packet_in` budget C from 300 to 600. In the second group, we fix $C \in \{400, 600\}$, and vary α_r from 0.05 to 0.3. In both groups of experiments, we experiment with two different CPS attack severity levels: (i) light CPS with 20% abnormal flows and (ii) severe CPS with 80% abnormal flows.

Fig. 3 depicts the impact of the `packet_in` budget C. In particular, Fig. 3a shows INFAS performance under different `packet_in` budgets, when α_r is fixed to 0.1. We can see that, in the light CPS case, β stays small when only a small number of `packet_in` messages are allowed to be generated within a time slot. After that, it increases almost proportionally to the budget C. As for the severe CPS case, the acceptance ratio β always remains below 0.2, no matter how C changes. This means that INFAS blocks the majority of unmatched packets from `h1`.

Fig. 3b shows the system performance when α_r is fixed to 0.3. We can see that the acceptance ratio β increases with C, under both attack levels, but with smaller values in the case of the severe attack. The reason is that both attack levels are classified as an *overload case* or *suspicious case* by the flow management algorithm, when the unmatched packet proportion threshold α_r is relatively large.

These results can be interpreted as follows: in the light CPS case, the proportion of unmatched packets is relatively small and just a bit lower than the threshold α_r. The flow management algorithm treats it as an *overload case*, thus enforces a light penalty. However, in the severe CPS case, a large portion of packets from `h1` cannot match concrete flow rules due to their long inter-arrival time. The flow management algorithm classifies it as an *attack case*, thus enforces a heavy penalty on the acceptance probability p.

Fig. 4 depicts the impact of the unmatched packet proportion threshold α_r on INFAS performance. Fig. 4a shows the performance change under different values of α_r, when C is fixed to 400. In general, we can see that increasing α_r allows more unmatched packets to trigger table-miss events. In the light CPS case, β is relatively small (around 0.7) when α_r is set to 0.05, and it starts to become constant once α_r reaches a certain value. The reason is that when α_r is large enough, the light CPS case will be treated as an *overload case* by the flow management algorithm. In this case, the unmatched packet acceptance ratio β is only determined by the `packet_in` budget C and the total number of unmatched packets. This point can be also observed in Fig. 4b.

With the above presented results, we confirm that INFAS can effectively block malicious flow packets that deliberately trigger table-miss events in SDN networks. By this, INFAS significantly mitigates CPS, depending on the severity level of the attack. Furthermore, the message budget C and the unmatched packet proportion threshold a_r have impacts on INFAS performance. In addition, choosing proper values for C and α_r in real systems requires to measure the controller capacity and to estimate the traffic patterns. When this is achieved, INFAS parameters can be tuned in an adaptive way.

*2) **Switch throughput**:* We measure the switch throughput by connecting an additional pair of Docker containers to `s1`, and configuring several static concrete flow rules to allow their mutual communication. In our evaluation, we treat their experienced bandwidth as the throughput of the switch. The standard tool `iperf3` [21] is used to test the bandwidth under three scenarios: (i) attack-free system, (ii) a system under a severe CPS attack with INFAS enabled, and (iii) a system under a severe CPS attack without INFAS. Note that, due to the fact that OvS operates in the kernel mode, its throughput can reach about 47Gbit/s on the used hardware. Under the severe CPS attack without protection, the switch throughput drops to around 35Gbit/s. When enabling INFAS, we achieve roughly 44Gbit/s. Such an improvement (about 26%) confirms that INFAS effectively blocks a large amounts of unmatched packets before they trigger the generation of `packet_in` messages. That is to say, the workload of the switch CPU and the netlink channel connecting the OvS kernel module and OpenFlow daemon is dramatically reduced, which contributes to the improved switch performance.

V. Conclusion and Future Work

The Control Plane Saturation (CPS) is a DoS attack being capable to significantly disrupt the operation of SDN. The adversary floods the data plane with flow packets not matching the stored flow rules. As a consequence, floods of control packets are exchanged between the switches and the controller. We presented INFAS, a defense scheme for protecting SDN against CPS. INFAS is installed on the rapidly increasing in-network commodity servers, which are used in modern networks mainly to support NFV. The switch forwards the flow packets that do not match any of its concrete flow rules to a nearby INFAS module. INFAS evaluates these packets, and accordingly decides either to return them to the corresponding switches or to drop them.

In the future, we plan to improve the current design of INFAS in several ways. In particular, we will investigate approaches to adjust INFAS parameters in an adaptive way, for different types of SDN networks. Another idea for investigation is to support cooperation among multiple INFAS modules to improve detection and mitigation of distributed DoS attacks.

ACKNOWLEDGMENT

This work was supported by the German Research Foundation (DFG) as part of the A12 and A08 projects within the Collaborative Research Center (CRC) 912 – HAEC. It was also supported by the German Federal Ministry of Education and Research (BMBF) as part of the project "fast robotics" under grant 03ZZ0528E.

REFERENCES

[1] O. N. Fundation, "Software-defined networking: The new norm for networks," *ONF White Paper*, vol. 2, pp. 2–6, 2012.

[2] S. Scott-Hayward, S. Natarajan, and S. Sezer, "A survey of security in software defined networks," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 623–654, 2016.

[3] G. Shang, P. Zhe, X. Bin, H. Aiqun, and R. Kui, "Flooddefender: Protecting data and control plane resources under sdn-aimed dos attacks," in *Proceedings of the 36th IEEE Conference on Computer Communications (INFOCOM)*, 2017, pp. 1–9.

[4] S. Wang, K. G. Chavez, and S. Kandeepan, "Seco: Sdn secure controller algorithm for detecting and defending denial of service attacks," in *Proceedings of the IEEE Information and Communication Technology (ICoIC7)*, 2017, pp. 1–6.

[5] N.-N. Dao, J. Park, M. Park, S. Cho *et al.*, "A feasible method to combat against ddos attack in sdn network," in *Proceedings of the IEEE International Conference on Information Networking (ICOIN)*, 2015, pp. 309–311.

[6] D. Intel, "Data plane development kit," 2014.

[7] S. Shin and G. Gu, "Attacking software-defined networks: A first feasibility study," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013, pp. 165–166.

[8] L. Dridi and M. F. Zhani, "Sdn-guard: Dos attacks mitigation in sdn networks," in *Proceedings of the IEEE 5th International Conference on Cloud Networking (Cloudnet)*, 2016, pp. 212–217.

[9] L. Wei and C. Fung, "Flowranger: A request prioritizing algorithm for controller dos attacks in software defined networks," in *Proceedings of the IEEE International Conference on Communications (ICC)*, 2015, pp. 5254–5259.

[10] H. Wang, L. Xu, and G. Gu, "Floodguard: A dos attack prevention extension in software-defined networks," *Proceeding of the IEEE/IFIP Dependable Systems and Networks (DSN)*, 2015.

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

[12] A. M. Bahaa-Eldin, E. E.-E. ElDessouky, and H. Dağ, "Protecting openflow switches against denial of service attacks," in *Proceedings of the IEEE International Conference on Computer Engineering and Systems (ICCES)*, 2017, pp. 479–484.

[13] M.-A. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal, "Enhancing vnf performance by exploiting sr-iov and dpdk packet processing acceleration," in *Proceedings of the IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, 2015, pp. 74–78.

[14] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[15] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "Opennetvm: Flexible, high performance nfv," in *Proceedings of the IEEE NetSoft Conference and Workshops (NetSoft)*, 2016, pp. 359–360.

[16] "MinnowBoard Turbot Dual Ethernet Quad Core System," https://store.netgate.com/MBT-4220-system.aspx [Accessed 05-September-2018].

[17] "Open vSwitch," https://www.openvswitch.org [Accessed 05-September-2018].

[18] "ovs python api," https://github.com/PinaeCloud/ovs-api [Accessed 05-September-2018].

[19] "Floodlight SDN controller," http://www.projectfloodlight.org/floodlight/ [Accessed 05-September-2018].

[20] "Docker container," https://www.docker.com [Accessed 05-September-2018].

[21] "iperf3," https://iperf.fr/iperf-download.php [Accessed 05-September-2018].