Balancing the Detection of Malicious Traffic in SDN Context

Bruno Salgado Machado
Centro Algoritmi &
Universidade do Minho,
4710-057 Braga, Portugal
Email: a74941@alunos.uminho.pt

João Marco C. Silva
INESC TEC,
HASLab & Universidade do Minho,
4710-057 Braga, Portugal
Email: joao.marco@inesctec.pt

Solange Rito Lima, Paulo Carvalho
Centro Algoritmi &
Universidade do Minho,
4710-057 Braga, Portugal
Email: {solange@, pmc@ }di.uminho.pt

***Abstract*—Huge efforts and resources are spent every year on prevention and recovery of cyberattacks targeting users, services and network infrastructures. Software-Defined Networking (SDN) is a technology providing advances to the field of security with the ability of programming the network, promoting high-performance solutions and efficient resource utilization at low costs, as the use of specialized hardware is avoided. The present paper aims at exploring the SDN paradigm to develop an SDN-based framework for prevention and mitigation of malicious attacks throuhgt the network. The framework design and proposal has concerns regarding the efficient use of network and computational resources, distributing the inspection of suspicious flows by distinct Intrusion Detection Systems. For this purpose, a load-balancing strategy for traffic inspection is devised, allowing to balance both the usage of resources and the analysis of traffic flows. In this way, this paper also sheds light on the usage of OpenFlow messages to build distributed SDN-based applications with the mentioned properties.**

I. Introduction

The Internet has led to the establishment of a digital society where everything is connected and accessible from anywhere. However, the current Internet architecture is complex and hard to manage due to its ubiquity and heterogeneity. In particular, security is introduced via expensive, specialized and hard-to-configure equipment (middleboxes) such as Intrusion Detection System (IDS), Intrusion Prevention System (IPS), firewalls, among others. In this context, the Software-Defined Networking (SDN) concept arises as an appealing solution to this problem, by providing the ability to program the network through a centralized network control and by decoupling the control and data planes. The centralized and cost-effective architecture inherent to SDN improves network visibility, helping to achieve efficient resource utilization and high performance. In addition, it is possible to recreate middleboxes used in standard networks with software, making their adoption and integration easy and affordable. Based on programming, network security can be enhanced by quickly diverting or analyzing malicious traffic.

OpenFlow is an SDN standard network communication protocol that allows the SDN controller to interact with the forwarding plane of network devices, such as switches and routers. An OpenFlow switch has one or more flow tables composed of various rules called flows. Each flow matches a specific set of packets and performs actions on them, such as forward, drop, adjust. With the ability to interact directly with the network resources through OpenFlow, a real-time overview of the network can be obtained at any point. This, coupled with the ability to program the network, can be integrated with standard security and load balancing software to obtain a robust solution regarding network efficiency, resource utilization, and security.

Striving to create an efficient solution to improve network security, this paper explores the above concepts to propose a scalable solution using SDN. Taking advantage of distributed IDSs, the proposed security framework, totally based on software, integrates load balancing algorithms and severity alerting priorities to improve scalability and security efficiency. The proof-of-concept resorts to distinct Mininet network schemes to analyze and attest the impact of the framework on improving network security and resource utilization. In addition, this paper clarifies the relevance and usage of *ofp_packet_in, ofp_flow_mods, ofp_group_mod* OpenFlow messages to build distributed SDN-based applications.

This paper is organized as follows: Section II presents relevant OpenFlow concepts to fully understand the proposed framework implementation; Section III discusses the related work regarding security frameworks and load-balance strategies; Section IV presents the proposed SDN-based security framework, its design goals, architecture and main components; Section V provides the proof-of-concept evaluating the framework performance; and Section VI concludes this work.

II. Background Concepts

SDN relies on a network architecture where the forwarding functionalities are removed from the network devices. Instead, these functions are handed out to a remote control, decoupling the control and data plane [1]. The controller is capable of handling the entire traffic stream and individually deciding on routing, flow forwarding, and packet filtering through predefined instructions [2]. Although initially explored for packet forwarding, network security has also emerged as a promising field for SDN, as discussed in Section III.

A. OpenFlow

OpenFlow is a southbound Application Programming Interface (API) and is considered one of the first SDN standards,

created at the University of Stanford in 2008 [3]. OpenFlow is a non-proprietary protocol that defines an API for communication between the controller and network devices, filling the gap regarding the lack of flexible network programmability [4], and allowing for high-performance and low-cost implementations. An OpenFlow switch consists of one or more flow tables, a group table which perform packet lookups and forwarding, and an OpenFlow channel to an external controller. A flow table contains a list of flow entries. Each flow entry contains a set of fields that may change according to the OpenFlow version. The fields that remain constant among all OpenFlow versions are: (i) *Match fields*: against which a packet is matched; (ii) *Counters*: updated when a packet match occurs; and (iii) *Instructions*: attached to a flow entry to describe OpenFlow processing when a packet matches that entry. As regards OpenFlow messages, the following are particularly relevant:

Packet In - allows interaction between the datapath and the controller. This message (*ofp_packet_in*) is used in three situations: i) when an action defined in a flow asks for this message; ii) when there is no flow matching the packet (table miss); or iii) as result of a Time to Live (TTL) error [3].

Flow Mod - allows changing the state of a flow in a switch, namely delete, add, or modify. All *Flow Mod* messages begin with the standard OpenFlow header, containing the appropriate version and type values, followed by the message structure. One import parameter of *ofp_flow_ mod*, relevant for this work, is the hard timeout that represents the number of seconds a flow is active, regardless of its activity duration.

Group Mod allows managing groups (create, modify, and delete) in the group table. A group is an abstraction that eases complex and specialized packet operations that cannot be easily performed through a flow table entry. Examples of these operations are packet mirroring, link redundancy for failure prevention, among others. Each group receives packets as input and performs any OpenFlow actions on these packets.

These OpenFlow messages will be explored in the concretization of the SDN security framework, being their usage highlighted in Section IV.

III. Related Work

As reconfigurability is a key property associated with SDN, this concept has been explored to devise SDN-based defense solutions to detect and prevent network intrusion attacks. In [5], a network framework able to forward or mirror incoming packets to an off-path processing unit via OpenFlow switch is proposed. This processing unit includes Deep Packet Inspection (DPI), IDS, Distributed Denial of Service (DDoS) units, and firewall. After arriving at the processing unit, a packet is passed to the DPI unit for analysis, being the controller notified. Then, the policy creation module of the controller creates proper policies and notifies the processing unit. In case of detection of malicious traffic, the processing unit can react in 3 separate ways: alert, quarantine, and block. In case of an alert action, the forwarding rules are not changed, and the controller is notified with an alert, whereas under quarantine, the entire traffic will be logged. Finally, if a block action is fired, a message is sent to the controller to block specific traffic.

In [6], the authors proposed a security framework consisting of an SDN controller, a clustering node and Detection as a Service (DaaS) nodes. In this framework, all network traffic is mirrored and sent to the clustering node. The clustering node acts as a load-balancing unit, deciding to which of the DaaS nodes the traffic will be sent to. A DaaS acts similarly to an IDS, analyzing and detecting threats, and deciding whether the traffic is malicious or benign. If the traffic is considered malicious, the DaaS will inform an SDN application running on the controller that the flow should be blocked. Next, the controller will insert the blocking entry on the switch, blocking further traffic.

To detect anomaly-based attacks with the help of SDN, in [7], a graphical approach has been proposed that uses OpenFlow switches to discover the source of attacks, finding paths susceptible to an anomalous attack. With SDN, collaborative detection can be implemented where each switch or host reports its attack detection to the controller. Then, the controller can decide to consider an attack if just one, the majority, or all the devices report an attack, enhancing the performance of anomaly detection [2].

Regarding the detection, prevention, and recovering from DDoS attacks through SDN, in [8], the authors suggest the detection of DDoS attacks using flow volume and flow rate asymmetry feature. The approach uses a sequential and concurrent method to change flow monitoring granularity on all switches to quickly locate the potential victims and suspicious attackers.

After evaluating existing works in network security that leverage the use of SDN, the gap of multilevel load balancing strategies, determinant to reach a scalable and flexible solution, is evident. This gap concerns both the distribution of load between network resources and the requirements of traffic analysis, as existing solutions always rely on examining all packets in the network. With this open issue in mind, this work proposes a robust solution for both network security and load-balancing traffic analysis, allowing to save resources and increase network usage when there is no need for a complete breakdown of all traffic flowing in the network.

IV. Proposed Approach

This section presents the design goals and the architecture of the devised SDN-based security framework. The main components and protocol interactions which allow a proper load-balancing among switches and IDSs are also described.

A. Design Goals

The objective of this work is to create an efficient security framework sustained by the SDN capabilities. In this way, it is necessary to define the relevant design goals to accomplish this purpose. Taking advantage of SDN features, and considering security, flexibility, scalability, and load balancing as target properties, the framework design goals are as follows:

- enhance security without jeopardizing network performance, resorting to an SDN architecture with off-path traffic analysis;
- distribute traffic load among switches and IDS;
- maximize malicious traffic detection, without analyzing all packets;
- identify and block traffic from malicious sources;
- be compatible with OpenFlow-enabled devices.

B. System Architecture

The spacial representation of the proposed architecture is illustrated in Figure 1. As shown, an SDN controller interacts and configures network switches and IDSs to allow a load-balanced inspection of existing network flows.

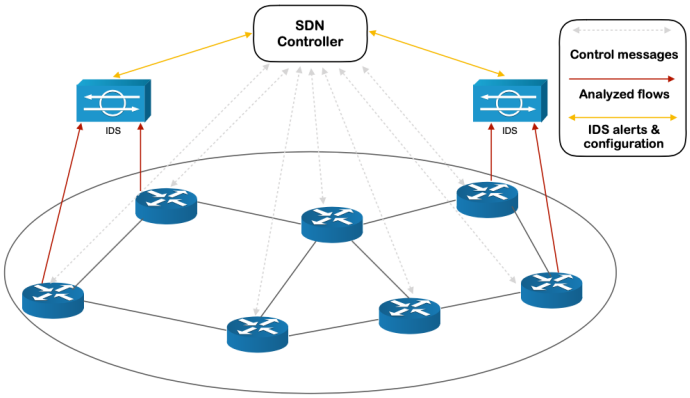


Fig. 1. System Architecture

As presented in Figure 2, in operational terms, the system is composed of a virtual network emulated by Mininet [9] that creates virtual OpenFlow switches with the help of Open VSwitch (OVS). Then, as a southbound protocol, OpenFlow establishes bidirectional communication between the SDN controller (Floodlight) [10] and the OpenFlow switches. When a flow is mirrored, it is sent to Snort [11] to find out if the flow is malicious. Snort is executed with the option to output the alerts to a UnixSocket called *snort_alert*. The component UnixClient.py reads all alerts sent by Snort to the Unix socket and redirects them to Floodlight. This component acts as a client, establishing a connection with a server running as a thread in the module written in Floodlight. Each of these entities possesses a certificate for the establishment of a Transport Layer Security (TLS) 1.3 connection.

The proposed framework mirrors incoming traffic based on flow distribution to off-path IDS, which will infer the maliciousness within the traffic. The IDSss produce alerts in the presence of malicious traffic which are then sent to the UnixSocket unit. An independent component will read the incoming alerts and establish a TLS [12] connection with the SDN controller to transmit the security alerts safely. The SDN controller will take proper network/switches configuration measures according to the received feedback. Depending on the flow analysis outcome, monitoring will continue as before

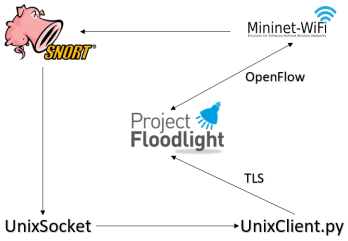


Fig. 2. System Components

or the incoming flow is blocked. If a flow is considered benign, that flow stops being examined regularly. To improve scalability, packets mirroring and their corresponding inspection are distributed among the switches and IDSs within the network, as discussed in Section IV-C.

C. Main Components

The proposed framework is composed by a set of software components running in different platforms (Floodlight, Snort, Mininet and the UnixClient.py, as shown in Figure 2). Considering the scope of this work, an additional module was developed and integrated into Floodlight platform. This module is composed of several threads of type *packet_in* handler, namely, a flow stats thread for each switch in the topology, a round-robin thread, and a Snort server thread. The behavior of these threads are explained below, followed by the description of the proposed load-balancing algorithm.

1) Packet_in Handler: In SDN, when a packet reaches a switch without matching any existing flow, an *ofp_packet_in* is sent from the switch to the controller. The *Packet_in* handler processes these messages and decides which switch and IDS will be responsible for mirroring the traffic. In this decision, it is assumed that all switches and IDSs have possible paths to destination hosts in the network. The flow diagram in Figure 3 resumes the main features of the *Packet_in* thread.

The *Packet_in* handler has two types of behavior depending on the existence of a match. In the case of there is a match for the incoming packet, the switch limits itself to perform the actions present in the instructions installed for the respective flow (Step 2). Otherwise, Step 3 is applied. A structure called *groups* defined within the new Floodlight module keeps all information of the *Group Mods* created to mirror the traffic. From now on, each entry will be referred as a *group*[1].

Case 1 - There are two reasons for a nonexistent group match (Step 3.1 check): the flow was not created, or the flow is no longer active because the hard timeout associated with the flow has already expired. If the group does not exist, Steps 3.2 to 3.9 are applied. Then, it is chosen the best switch and IDS that are going to be responsible for duplicating and analyzing the duplicated flow, respectively. Once this is carried out, the interface connecting the switch and the IDS is determined, and a group is created with all the relevant information including

[1]Unlike the temporary nature of *Flow Mods*, *Group Mods*, once created, are persistent even if no *Flow Mods* are applied for *Group Mods* actions.

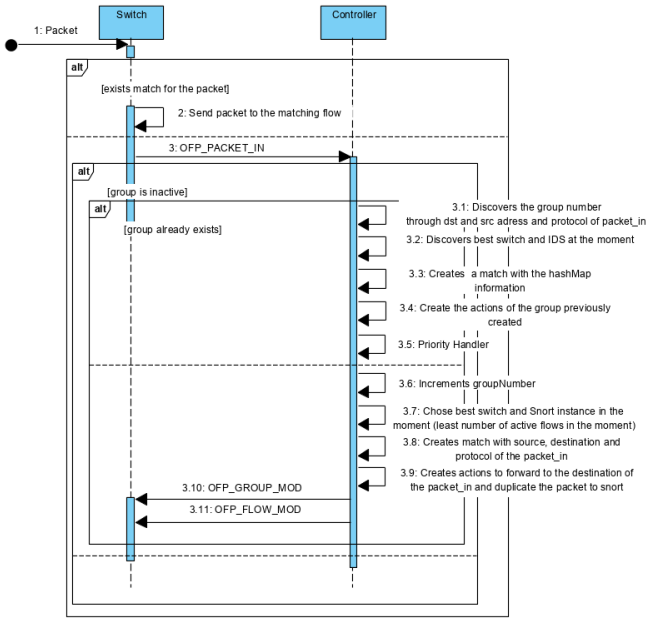


Fig. 3. *Packet_in* handler

source/destination IP, switch, and IDS instance's Media Access Control (MAC) addresses. Next, a match to the flow is created, taking into consideration the transport protocol as well.

The actions are then built, creating a bucket with two lists of actions: one for normal forwarding and other that will change the MAC address of the packet that matches the MAC interface determined before, mirroring it to the respective IDS port, also determined previously. Finally, a *Group Mod* is created with the match and actions generated in the previous steps, and sent to the respective switch (Steps 3.10 and 3.11). A *Flow Mod* is also created to perform the actions of *Group Mod*. At the end, the group number is increased so that all groups have a distinct identifier. As last step, the number of active flows in each switch is updated.

Case 2 - When the group already exists, but there is no flow applying to the *Group Mod* actions, the group number is determined based on the source and destination IP addresses and protocol in the *groups* data structure. At this point, the group priority is checked, as it can influence the definition of actions, as discussed next. Similarly to the case of there was no group created, the best switch and IDS are determined, and the respective actions and match fields created.

Group Priority and Quarantine - The priority represents the severity of the alerts found in a flow/group. The priority values range from 0 to 4, being 1 the most dangerous alert. For example, while an *Indicator-scan UPnP service discover attempt* has priority 3, an *Indicator-shellcode x86 inc ebx NOOP* has priority 1. The value 0 is reserved for no alerts.

Flows with priority 0 will suffer no action as they will be handled by the round-robin thread and *benignGroups* structure discussed below. For priorities between 2 and 4, it is checked if the newly elected switch and IDS are the same as the ones currently in use for this flow/group. If at least one is different, the *Group Mod* is deleted as the actions will not match. If the priority is 4 a new *Group Mod* is created and installed in the respective switch, and a new *Flow Mod* is created with double default hard timeout. This hard timeout value (quarantine time) was selected to assure that a flow exhibiting signs of ill-intent is fully analyzed. The same behavior applies if the priority is 3, however, the *Flow Mod* is now created with the quadruple of the default hard timeout. As this priority represents a higher potential for dangerous behavior, the time of analysis must be prolonged. In case of priority 2, the corresponding *Group Mod* diverts all traffic to the IDS, without sending any traffic to the intended destination. In this case, a *Flow Mod* with a quadruple default hard timeout is also created and sent to its respective switch. Finally, when the priority is 1, the installed *Group Mod* is deleted and a *Flow Mods* is created so that all packets matching dangerous flows are dropped. These *Flow Mods* are permanent (have no hard timeout), and sent to all switches in the network to proceed accordingly. Every time a *Flow Mod* is sent, the number of active flows in the respective switch and IDS is updated.

2) Flow Stats: The flow stats thread controls which flows are active per switch, and also monitors the number of flows per switch and IDS (used as input for the load-balancing algorithm). An OVS script prints the active flows in each switch, obtaining the flow's data. The information on active flows is then compared with the information in the *groups* structure to find which groups are no longer active. When a flow becomes inactive and has zero priority (no alerts) it is added to a new data structure called *beningGroups*, composed of no malicious flows. The flow stats thread runs on each switch for a complete gathering information across all topology, updating data every 5 seconds.

3) Round Robin: The round-robin thread aims at reducing the number of examined flows, without degrading the security level. This thread starts by waiting until the *benignGroups* structure has elements, and then performs a round-robin cycle to periodically activate new flows for analysis. When a group is selected for activation, the best current switch and IDS to analyse it is recalculated.

4) Unix Client and Snort Server: The UnixClient starts by establishing a TLS 1.3 connection between the Snort Server and the SDN controller. After that, the alerts from the *snort_alert* Unix Socket are read and sent to the Snort Server. When the Snort Server receives an alert, the protocol, source, and destination IP are checked for a match to identify alert's origin. Once the flow and priority corresponding to the alert are found, the procedure described in the *Packet In* section takes place.

5) Load-Balancing Algorithm: The load-balancing algorithm is used to determine the best switch and IDS to handle incoming flows. Taking into account the discussion in Section III, the algorithm selected in this work follows the same principles of the least connections [13], called the least number of active flows. The advantages of this algorithm are its simplicity and low-time of execution, which are pivotal

characteristics for an algorithm highly dependable on scalability. However, the performance of this algorithm decreases when the flows differ significantly in the number of packets, which impacts the amount of traffic being balanced between network components. Thus, packet counts must be considered as an additional input to enhance the balancing strategy. At present, the load-balancing algorithm starts by determining the switch/IDS with the least number of active flows. In case of a tie, these components are chosen randomly.

V. Tests and Results

In order to evaluate the proposed framework, a set of tests was performed using experimental topology scenarios configured in Mininet. The purpose was to assess the flexibility and efficiency of load distribution among switches and IDSs, and the capacity to detect anomalous traffic.

A. Experimental Setup

The experimental analysis resorts to two types of traffic traces. The first one consists of traffic resulting from a vulnerability scan using the full Nessus' database [14]. Thus, this trace contains hundreds of flows expected to be identified as anomalous by the IDS. The second trace includes benign traffic resulting from video streaming sessions. These traces are then injected into the experimental topologies depicted in Figure 4.

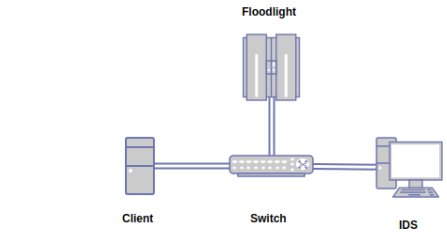
Topology 1 (illustrated in Fig. 4(a)) is a simple environment comprising a single IDS, SDN-capable switch, and SDN controller, used uniquely as a test baseline. Topology 2 (Fig. 4(b)), used to assess the efficiency of the distributed traffic analysis, includes two IDSs, three SDN-capable switches and one SDN controller responsible for load balancing and result analysis.

B. Evaluation Tests and Results

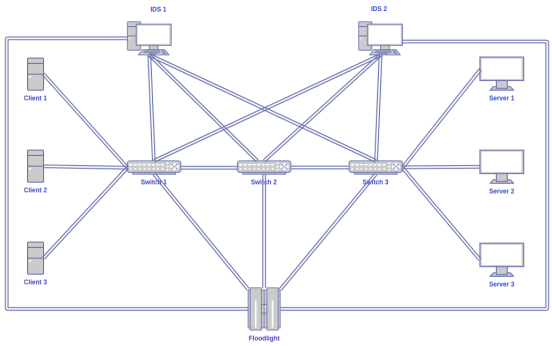
The first test aims at analysing the framework's efficiency in detecting malicious traffic and taking action on it. To do so, the malign trace was injected in Topology 1 (Fig. 4(a)) having the developed Floodlight module deactivated, allowing the traffic to flow naturally in the network. As a result, the IDS generated 156 alerts of 17 anomaly types.

Then, the same trace was injected in Topology 2 (Fig. 4(b)) with the security module activated and configured to drop all traffic related to the generated alerts. In this case, the *Flow Mod* created to drop packets was configured without *hard timeout*. As consequence, the system generated 24 alerts from 4 different types. This means that after being identified for the first time by any of the IDSs, all traffic of the same type was blocked by the SDN controller, which demonstrates the overall system accuracy.

Combined with the adopted load-balancing algorithm, the *timeout* defined for each flow type may affect the number of packets analyzed in each IDS. In this way, while assessing the ability to distribute traffic between IDSs, three sets of *hard timeouts* are comparatively evaluated. As shown in Table I, the default and configurable *timeouts* for IDS alerts with priority 4, priorities 3 and 2, and priority 1 are defined as explained in Section IV-C (*Group Priority and Quarantine*).



(a) Topology 1



(b) Topology 2

Fig. 4. Evaluation scenarios

TABLE I
Timeouts per type of alert.

	Set 1	Set 2	Set 3
DEFAULT	20 sec	60 sec	90 sec
PRIORITY_4	40 sec	120 sec	180 sec
PRIORITY_3, PRIORITY_2	80 sec	240 sec	360 sec
PRIORITY_1	drop	drop	drop

Resorting to Topology 2, different test scenarios combine the analyzed traces and *timeout* configurations to assess the efficiency of load balancing. For the malicious traffic trace, Table II summarizes the resulting load measured as the number of packets mirrored by each switch to the IDS.

The results shown that, although using a load-balancing algorithm based on the number of active flows being analysed by each IDS (without considering flow size), the framework is able to fairly distribute traffic between the detection systems. The difference in the total of packets in each scenario is due to three factors: (i) acknowledgment packets exchanged between the hosts and the SDN controller; (ii) packet broadcasting when the forwarding module of Floodlight does not know yet a path to a given host; and (iii) control packets (e.g. ICMP and LLDP) used by the topology module of Floodlight to discover active hosts.

Table II also presents the results for the test scenarios using benign traffic and Topology 2. A first remark is the absence of IDS alerts, showing the framework overall correctness regarding false positives. Here, the distribution of traffic between IDSs is similar to scenarios 1-3. Comparing the results across all scenarios, it is clear that under malicious traffic, the number of analyzed packets increases. In fact, when

TABLE II
OVERALL RESULTS

Malicious traffic									
Tests	Scenario 1			Scenario 2			Scenario 3		
Timeouts	Set1			Set2			Set 3		
Nº Alerts	138			399			261		
	IDS 1	IDS 2	Total	IDS 1	IDS 2	Total	IDS 1	IDS 2	Total
Switch 1	71794	38261	110055	87659	28431	116090	115059	47320	162379
Switch 2	71802	55158	126960	163980	90159	254139	76075	97874	173949
Switch 3	95788	87680	183468	171717	82069	253786	65129	49634	114763
Total	239384	181099	420483	423356	200659	624015	256263	198828	451091
Benign traffic									
Tests	Scenario 4			Scenario 5			Scenario 6		
Timeouts	Set1			Set2			Set 3		
	IDS 1	IDS 2	Total	IDS 1	IDS 2	Total	IDS 1	IDS 2	Total
Switch 1	10852	14303	25155	69078	25760	94838	31148	45130	76278
Switch 2	57623	32168	89791	72127	32488	104615	74807	67905	142712
Switch 3	9229	12733	21962	28825	75046	103871	111644	14338	125882
Total	77704	59204	136911	170030	133294	303324	217599	127373	344972

a flow is considered malicious, it will always be analyzed, and never enters the round-robin process. In scenarios 1-3, as almost every flow in Nessus trace is deemed malicious by Snort, all flows at a given time were tagged as malicious. On the contrary, benign traffic only yields benign flows. Upon becoming inactive, flows are inserted in the round-robin process, and will only be analyzed again sporadically in the future. This decreases the number of packets under analysis dramatically.

Looking at the results collected for malicious traffic, it is possible to conclude that the amount of analyzed traffic is not directly proportional to the increase of the *hard timeout*. The framework behavior strongly depends on the number of alerts generated, as well as their severity. On the other side, in presence of benign traffic, the amount of traffic analyzed is directly proportional to the *hard timeout* condition.

The test with the best performance using malicious traffic was Scenario 2 (default *hard timeout* of 60 seconds), where the higher number of alerts was found, also aligned with the larger amount of traffic examined, for all tests. Under benign traffic, Scenario 4, which corresponds to the smaller *hard timeout*, i.e., 20 seconds, led to the best results.

VI. CONCLUSIONS

In this work, IDSs were coupled with the programmability of SDN to create a reliable and scalable framework to improve network security. This framework is composed by an SDN controller, the underneath topology with IDS nodes, and an SDN application. The SDN application is responsible for managing the packets that need to be mirrored for analysis, determining actions for malicious traffic, as well as balancing traffic between each switch and IDS in the topology. The proof-of-concept attests that the framework detects and acts efficiently on malicious traffic. This was attained from the number of alerts found and the distribution of traffic load by all switches and IDSs. Although the results are encouraging, the load-balancing algorithm may be enhanced, namely accounting for flows comprising a significantly divergent number of packets. Considering that some attacks are performed by combining multiple flows, e.g., DDoS attacks, distributing the analyzed traffic across different IDSs without affecting the overall detection efficiency is also a future research topic.

Acknowledgements

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020 and by FCT – Fundação para a Ciência e Tecnologia – within the R&D Units Project Scope: UIDB/00319/2020

REFERENCES

[1] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):14–76, jan 2015.

[2] Danda B. Rawat and Swetha R. Reddy. Software Defined Networking Architecture, Security and Energy Efficiency: A Survey. *IEEE Communications Surveys & Tutorials*, 19(1):325–346, 2017.

[3] OpenFlow Switch Specification v.1.5.1, ONF, . URL: https://opennetworking.org/software-defined-standards/specifications/, Accessed: April 2021.

[4] David R. Teixeira, João Marco C. Silva, and Solange Rito Lima. Deploying time-based sampling techniques in software-defined networking. *2018 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6, 2018.

[5] S. Veena and R. Manju. Detection and mitigation of security attacks using real time SDN analytics. In *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, volume 2, pages 87–93, April 2017.

[6] Mehrnoosh Monshizadeh, Vikramajeet Khatri, and Raimo Kantola. Detection as a service: An SDN application. *International Conference on Advanced Communication Technology, ICACT*, pages 285–290, 2017.

[7] J. Francois and O. Festor. anomaly traceback using software defined networking. In *2014 IEEE International Workshop on Information Forensics and Security (WIFS)*.

[8] Yang Xu and Yong Liu. DDoS attack detection under SDN context. *Proceedings - IEEE INFOCOM*, 2016-July, 2016.

[9] Bob Lantz and Brian O'Connor. Mininet. URL: http://mininet.org/, Accessed: April 2021.

[10] Floodlight. URL: http://https://floodlight.atlassian.net/wiki/home/, Accessed: April 2021.

[11] Cisco. Snort. URL: https://www.snort.org, Accessed: April 2021.

[12] RFC 8446, TLS 1.3. URL: https://tools.ietf.org/html/rfc8446, IETF Accessed: April 2021.

[13] Anish Ghosh and Mrs Manoranjitham. A study on load balancing techniques in SDN. *Int. Journal of Eng. & Technology*, 7:174, 03 2018.

[14] Nessus. URL: https://www.tenable.com/products/nessus, Accessed: April 2021.