Implementation of a SDN architecture observer: detection of failure, distributed denial-of-service and unauthorized intrusion

Loïc Desgeorges, Jean-Philippe Georges, Thierry Divoux

Security and Communication Networks

Research Article

Implementation of a SDN architecture observer: detection of failure, distributed denial-of-service and unauthorized intrusion.

Loïc Desgeorges[1], Jean-Philippe Georges[1] and Thierry Divoux[1]

[1]Université de Lorraine, CNRS, CRAN, F-54000 Nancy, France

Correspondence should be addressed to Loïc Desgeorges: loic.desgeorges@univ-lorraine.fr

Abstract

Software-defined networking was recently introduced and proposed to separate the control from the data plane. This architecture introduces new challenges, particularly with regard to security and safety. To address the safety challenges, it is necessary to set up a multi controller architecture to provide redundancy. In addition, the second controller can have a security benefit because it can be used to validate the decisions taken by the first controller. However, communication between the controllers is necessary in these architectures, which may be exploited by an attacker to spread across the controllers, resulting in a security issue. This study aims to develop a multi controller architecture without communication between controllers. The control is executed by the nominal controller, which performs the data plane computation, whereas the second controller is in charge of verifying the consistency of the controller's decisions, i.e., the management traffic. We first formulated the activity of the command and then provided conditions to determine a consistent control. These conditions include a time boundary, which corresponds to the tolerance for a delay in the response time of the controller, and structural properties to verify the consistency of the path setup. Moreover, we proposed a detection algorithm that is divided into two parts: first, a learning phase that aims to learn the consistent path set up by the controller, and second, a running phase which aims to verify that the controller sets up paths that are similar to the learned path. This algorithm was evaluated in terms of its reactivity, precision, and recall. To evaluate this, we considered three use cases: a distributed denial of service (DDOS) attack, an attack to send malicious packets on the network, and a failure of the controller.

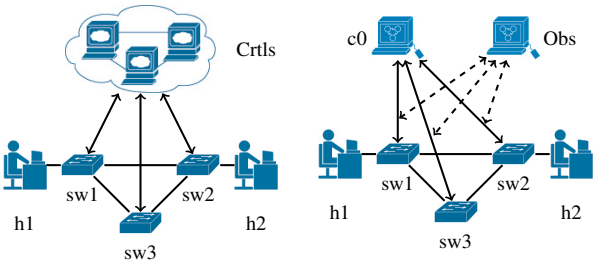
Keywords: Software-defined networking; security; safety; denial of service; detection; observation.

Introduction

Software-defined networking (SDN) [1] was introduced as a new paradigm in the networking field by moving network functions from the hardware environment to the software environment. This enables the management of various application requirements and dynamic networks [2]. SDN architecture is defined by two characteristics: decoupling of the control and data planes and programmability of the control plane [3]. Hence, such an architecture simplifies network management, facilitates network recovery [4] and permits to improve the security in the network as in [5] or [6] and summarized in [7]. The classical SDN control architecture comprises a single controller that controls the entire network. However, this architecture has two main drawbacks, scalability and robustness [8]. As a solution to these limitations, control architecture has been extended to consider

distributed control [9] [10]. A multi-controller proposal was introduced to improve the efficiency and availability of high-size networks [9]. The multiplicity of controllers provides the possibility of distributing the load between them and also provides active redundancy. However, multi-controller architectures face specific challenges in terms of consistency, reliability, load balancing, and security [10] [11].

This study focuses on the security and safety challenges of the control plane. SDN specifications can be used to protect against traditional attacks [7] [12]; however, these features also introduce new vulnerabilities. SDN architecture has introduced its own security challenges and each plane of the architecture has its own weaknesses [13] [14] [15]. In particular, as the controller is the brain of the network, the control plane is one of the most attractive targets in the case of an attack. An attack to take the control of the controller leads to access of the entire network. In addition, an attack to consume the resources of the controller (by a DDoS) leads to network collapse as presented in [16]. Similarly, in terms of the safety threat, a failure of the controller causes the network to be paralysed. Several mechanisms have been proposed in the literature to address the security and safety challenges of the control plane [17]. The first line of development has been to enforce the controller as in [18], with the aim of proposing a robust architecture that will enable it to be more robust to the considered threat or in [19] which proposes to set up a security module in the controller to analyze its received traffic using the Support Vector Machine (SVM) classifier to detect a DDoS attack. However, with such a proposition, the control architecture does not provide a controller redundancy and so there is still a single point of failure. Hence, to consider safety aspect these solutions are not applicable. The next idea is to enforce the control plane architecture by proposing a multi-controller approach, as in [20] or [21]: the aim is to combine several controllers to create a more robust global architecture by providing an, active or not, redundancy of the controller. These other controllers may have several roles regarding the security and safety of control. It can be to detect anomalies in the behavior of the others, as in [20], or it can be to validate the decisions of the other controllers, as in a BlockChain [21]. However, as far as we know, in all the current control plane architecture proposed an interface of communication between them (east-west interface) must be installed. Nevertheless, this interface is also a security threat because an attack may spread through it [22]. Besides, all the multi controller architectures introduced rely on a communication between the controllers. This interface is a threat as an attacker may spread malicious information thought it [22]



(a) Classical mutli controller architecture

(b) The proposed multi controller architecture

Figure 1: Differences between a classical multi controller architecture and our proposal

The objective of this study is to introduce a multi controller approach without an east–west interface, as shown in Fig. 1. The first controller $c0$ is responsible for controlling the network and an observer, the second controller, is responsible for detecting any anomaly (safety or security) in control. We propose to estimate the internal variables of the nominal controller only by observing the activity of the control and without information from the controller itself or from traffic on the data plane (as both might be corrupted). In this context, attention has been paid to intrusion detection systems (IDS), particularly the specification-based approach. Furthermore, this study focuses on the data plane computation case using well-established deterministic algorithms (such as routing using the Dijkstra algorithm). Hence, the main contribution of this paper is first to propose a detection algorithm of the anomalies (due to a safety or security threats) of the control. Second, to implement it in order to evaluate its performance through simulations.

Related works are introduced in Section Related Works. A motivating example is presented in Section System and Threat Model. The proposed architecture is elaborated in Section Proposed architecture and the performance of the proposed detection algorithm is evaluated in Section Performance evaluation. Finally, future works are presented in the conclusion.

Related Works

This section presents related works and compares them with our proposed method. We will present the solutions described in the literature concerning threats on the control plane [23]: failure of the controller, denial of service, and a malicious controller (by controlling the device using a Kali Linux exploit, which we will be included in other sections of this paper). A comparison of these approaches with our proposed method is presented in Table 1.

Failure of the controller and Denial of Service

The heart of an SDN network is the control plane. All decisions are managed by the controller. Thus, the failure of the control plane impacts the entire network. To avoid the challenge of a single point of failure, the use of multiple controllers is suggested, as in [20] and [24]. The architecture is based on a primary backup mechanism. This is passive redundancy, which implies the detection of a failure. This detection was set using the communication between the controllers.

Similarly, it is possible to affect the performance of the controller with a Denial of Service (DoS) or Distributed-DoS (DDoS). The motivation for a DoS or a DDoS is to flood the resources of an intended user to hinder or stop the service [36]. Moreover, SDN switches have no control over incoming packets, and do not spend any time processing them. This means that the entity targeted by such attacks is the controller, and the controller resource saturation is an SDN DDoS threat [37]. Some solutions have been proposed for detecting such attacks [37] [16]. They can be divided into two categories: the spreadest, which aims to develop a self-statistical-policy-based defense controller, such as the reinforcing anti-DDoS actions in real time (RADAR) proposed in [38] or [39]. It introduced an extension module of SDN named DosDefender. As previously mentioned, we focus on the second category, which is the multi controller approach, to provide a solution in case of failure and/or attacks.

Using a distributed control architecture, a safe-guard scheme [25] was proposed for the protection of the control plane against DDoS attacks. They performed a two-stage defence procedure: anomaly traffic detection at the data plane and controller dynamic defence in the control plane. This defence includes controller remapping and access control to mitigate the DDoS attacks. Similarly, [20] introduced a recovery mechanism called CPRevovery which is a primary back-up mechanism that offers resilience in the case of DDoS attacks against the controller. PATMOS [26] introduced a set of procedures to mitigate DDoS attack effects on SDN controllers. The procedure is divided into three phases: identification of the overloaded controllers by an attack, selection of a master controller which will coordinate the clustering process, and finding a configuration. In addition, a multi-controller architecture based on blockchain has been proposed in the literature to guard against a single point of failure and denial of service threats, as in [27]. They proposed an analysis of traffic to recognize patterns that correspond to DDoS or TCP flooding. Blockchain has also been considered as a solution against DDoS attacks which targets the control plane as in [28]. They propose a blockchain-based SDN-targeted DDoS defense framework (BSD-Guard) which provides a cooperative detection and mitigation mechanism to protect SDN controllers. BSD-Guard introduces a blockchain-based secure middle plane at the interface of the control and data plane. This middle plane calculates the suspect rate of new flows based on the collected packet information and reports suspect lists of blockchain for immutably storing and sharing.

However, a DDoS attack may still spread through the E/W interface between the controllers. These methods are not concerned with the possibility of a malicious controller in the architecture and are susceptible to this threat.

Malicious controller in SDN

The network is sensitive to perturbations that affect the controller, particularly in the case of an attack that can lead to a malicious controller. Such an attack allows the attacker to have access to the entire network and to redefine, for example, the previously configured routing policy [40]. A multiple-controller architecture was proposed in [29] to prevent unsecured behaviour. Specifically, a decision-making security architecture is proposed. The aim is to prevent an attack by allowing all the controllers to observe each other. In particular, it consists of determining if the rules coming from a controller are valid. In this objective, there is a vote between all the other controllers which limits the influence

Table 1: Classification of the methods proposed to ensure a secure and safe control plane

	Threats on the control plane				
Related Works	Safety	DDOS	Malicious controller	Role of the Second Controller	East-West Communication ?
[20]	√	√		Detection	Yes
[24]	√			Detection	Yes
[25]		√		Detection	Yes
[26]	√	√		Detection	Yes
[27]	√	√		Detection	Yes
[28]		√		Validation	Yes
[29]			√	Validation	Yes
[30]			√	Validation	Yes
[31]	√		√	Validation	Yes
[21]	√		√	Validation	Yes
[32]	√		√	Validation	Yes
[33]			√	Validation	Yes
[34]	√		√	Detection	Yes
[35]	√		√	Detection	Yes
Our proposal	√	√	√	Detection	No

of a controller attack. Each controller participates in determining which controller is infected by validating its production. In addition, [30] proposed configuring a filter that validated the commands sent by the controller. A second controller, which played the role of a filter, was added for receiving the commands of the controller to the switches to validate them. Recently, blockchain has also been considered as an option to secure the control layer, particularly, the communication interface between the controllers, as in [31], [21] or [32]. The controllers' decisions are the subject of a vote among all other controllers to ensure the consistency of the decisions.

However, these observations were achieved through communication between the controllers. This communication is typical for a multiple controller approach and is established through an east-west interface. Such an interface is a weakness because a malicious controller can spread incorrect information through it. To address this issue, [33] proposed the introduction of a private key generator to the control plane to encrypt communications between controllers. The aim is to secure the communication channel used for the east-west interface. Similarly, [34] proposed a secure global architecture and focused on developing a secure communication mechanism between the controllers. They used an indirect method of communication by using "inter-domain agent flow" which constituted a secure communication tunnel. Additionally, a multigranularity approach to the security and safety challenges of the controller was proposed in [34].

Another solution proposed in the literature is the use of moving target defence, such as in [35] with the introduction of the notion of shadow controllers. Furthermore, in the case of detection of probing traffic, a part of the shadow controllers are randomly selected to respond to the traffic.

In summary, the multi-controller approach permits the handling of safe and secure threats by adding backup, which may have a specific role in the security and safety treatments, as in the aforementioned method. However, this implies a new security issue: the east-west interface. Therefore, we propose the introduction of a multi-controller architecture without an east-west interface.

In summary, the presented methods are classified as shown in Fig. 1. It can be observed that a first difference with the other works is that we consider both safety and security threats of control. But the main originality is that we propose a multi controller approach without East-West interface, without interface of communication between the controllers. This interface is a security threat and is avoided in our work.

Intrusion Detection System

We propose the development of the architecture shown in Fig. 1. To develop the detection logic of the observer, the IDS theory has been considered. According to [41], IDS proposals can be divided into two categories: focusing on the attack behaviour or the unfaulty behaviour of the system. The attack behaviour implies some assumptions regarding the considered attacks which is not the case here.

The second category was introduced to detect unknown attacks by focusing on unfaulty system behaviours. The principle is to compare the *normal* behaviour of the system with that of the running system. There are two techniques for formalizing unfaulty behaviour: anomaly detection techniques, which are based on a model of the unfaulty behaviour of the system, or specification-based techniques, which are based on a specification directly from documentation.

In addition, these two techniques may be combined, as in [42]. There are two steps: A specification is determined offline, and some statistical properties linked to the specification model are learned online. Here, a similar approach is proposed. The chosen specification formalism is a template, as proposed by [43], that expresses the causality between the requests from the switches and the commands of the controller. Moreover, this specification evolves online according to the estimation of the internal variables of the controller by observing the activity of the command. However, contrary to others' works using the notion of Template, we do not have any assumptions on the order of the events. Indeed, they are considering multi instance process but the event has an order which defines a precise sequence. In our case, we do not have any assumptions about the transmission order of the command (for instance, a controller may set up the forwarding entries by contacting the switches on the path in a random order). As a consequence, in response to a request (like the installation of a route between two nodes), we do not know exactly which command is expected (is it necessary to change the forwarding plane? which path will be selected? which switch will be contacted first?).

System and Threat Model

The objective of this section is to present the unwanted behaviour monitored by the observer. First, the generally considered threats are presented, followed by specific scenarios.

Threat model

The aforementioned attack is an example; however, SDN introduces several challenges. The points of attack in SDN architecture were enumerated in [44] [22]. Each layer of a SDN architecture has its own weakness; in this study, we consider only attacks which concern the activity of control.

Furthermore, [23] proposed classifying the threats of the control plane into three different categories, as shown in Fig. 2. First, we consider the threats from the application layer. Because the lack of a mechanism to ensure trust in communication between the controller and the application layer, as SDN controllers translate application information into configuration commands for the underlying infrastructure, a malicious application may be a source of damage. For example, unauthorized access to internal storage may offer the possibility of an attacker introducing contradictory network policies [18] [45]. Similarly, a malicious application may inject authorized but forged flows in the direction of the OpenFlow switches to either overwrite or flush existing rules in the flow tables of the switches [46] [47]. In addition, malicious applications can manipulate control packet handlers to execute service disruption by discarding the packets [45] or poison controller information and the topology view [48].

Second, regarding the threats because of scalability and the risk of DoS attacks which attempt to render a network unusable to the original users [49]. There are several scenarios for setting up a DoS attack on the controller. Here, we consider an outsider for a network attacker. As the communication between the controller and switches is established through a TCP connection, an outsider only requires the IP address and the port number of the controller to establish the communication. From this perspective, the attacker may flood the controller with a large number of requests, as demonstrated in [46].

Finally, multiple controllers can be deployed to control an SDN network and manage a large number of devices. However, this division of network control implies an information aggregation challenge between controllers, which is a threat. Furthermore, the communication interface between the controllers is a threat as soon

as a malicious controller may spread malicious information through it. Hence, a malicious controller can drive the entire network.

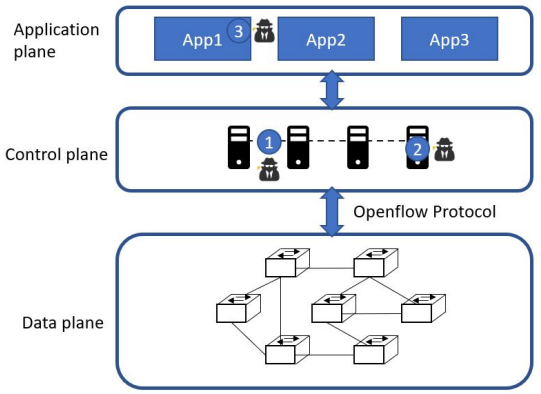


Figure 2: SDN main threats of the activity of the control

Motivating example

A controller in charge of routing using the Dijkstra algorithm is considered.

The controller chosen is ONOS, [50], which is supported by the Open Networking Foundation. It has been developed for experimenting with SDN. Also, ONOS is one of the most well-known SDN controller [51]. The ONOS controller [50] loads using a deterministic routing application. The controller communicates with the switches using OpenFlow v1.3 [52]. In addition, the controller discovers the topology using two applications, the host provider and LLDP to determine the topology of the network. Finally, proxyARP was used by the controller. Regarding the routing policy, we used the deterministic application *fwd* which uses the Dijkstra algorithm.

The topology in Fig. 3 is considered, and the following section presents an attack scenario.

Attack Scenario

As described in [17] and [22], new threats are introduced because of the nature of the centralized control of SDN. This study is focused on the command, and only attacks that impact the controller decisions are considered. For example, we consider internal storage abuse as described in [46] [47]. This corresponds to manipulating the flow table of a switch by using a malicious application. This malicious application accesses the internal storage of the controller and can alter the internal variables of the controller, such as the topology information. Such modifications have an impact on all peer applications which use topology information to derive flows and install a path over the network. In addition, modifying the internal variables of the controller may be used to violate the security policy of other applications, as described in [18].

An internal storage misuse attack on the ONOS controller of the topology is considered. Here, Fig. 3 is considered. The malicious application modifies the network topology data by exchanging the position of host 10.0.1.4 and host 10.0.1.5 in the controller topology storage. Consequently, if a host attempts to join host 10.0.1.4 then the data plane computed by the controller reaches 10.0.1.5 and not 10.0.1.4.

DDoS Scenario

There are several ways of attacking the controller using a DDoS: saturation of the computation resource of the controller, the storage resource, or the Southbound or Northbound bandwidth resource. We consider the resource saturation of the controller because of an external attacker. The attacker floods the controller with requests to open a TCP connection. The aim of this attack is to delay the response of the controller and render the controller unreachable.

Failure Scenario

The failure of the controller is also considered. A failure of the controller leads to the absence of control and decisions, and this compromises the entire network because all applications and services depend on it. In addition, this failure might be because of a distributed denial of service (DDOS) attack.

Problem statement

This section aims to introduce briefly the problem statement of this paper. The aim of the proposal is to implement an observer in charge of the detection of security and safety threats on the control. To set up such detection approach, we propose a novel architecture given at the right of Fig. 1. As mentioned in the section. Related Works, the main originality of our control plane is the

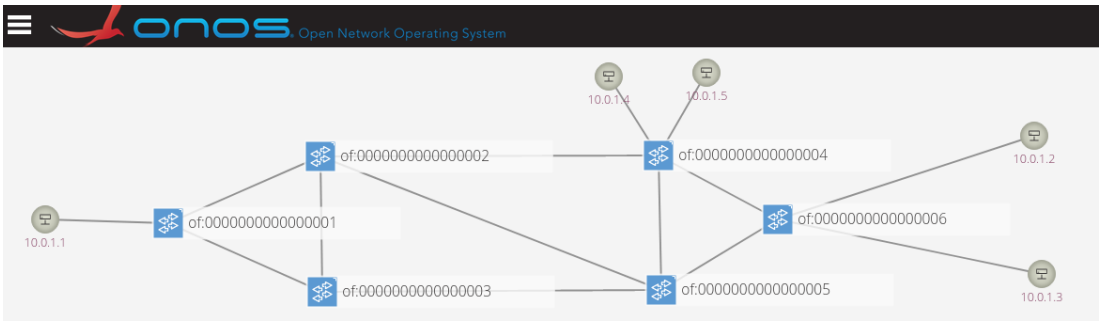


Figure 3: Network topology considered

absence of East-West communication between the controller and the observer. This choice is motivated by the fact that this interface is a security threat according to [22] and a consequence of it is that the observer does not have access to the internal states of the controller. The observer just has access to the activity of the control. The problem appears here: how to determine if the actions emitted by the controller correspond to the emanation of a healthy execution of the network function, without knowing the internal states of the controller. It is therefore necessary to define a behavioral model of the controller as well as structural rules concerning the actions implemented. This model, fed by the only observed packets at the Southbound interface, will have to be able to detect any dysfunction of the controller. Our work is based on the template formalism, which puts into the light the causality link between the events. However, contrary to others' works, we do not have any assumptions on the order of the events. In such a case, it is not possible to determine if an event is clearly expected or not. Here, we assume that in response to an event (a request for the infrastructure), a set of commands might be required. As a consequence, the sequence of events observed after a request has to be analyzed and structural properties of this set of command need to be defined so that the consistency is guaranteed. The architecture and detection approach is developed here after.

Proposed architecture

To respond to these challenges and threats, we propose the use of a multi-controller architecture without an east-west interface, as introduced in [53] and [54] and presented in Fig. 1. There is one nominal controller in charge of the network, whereas the second controller is an observer that detects safety or security threats without sending any flows on the network. It observes the exchange of packets at the southbound interface.

Owing to the absence of the east-west interface, the observer has no information from the nominal controller and its internal states contrary to other propositions as presented in section Related Works. Hence, the proposed method is based on some *a priori* knowledge of the control logic and the observation of its activity in the network. We formalized the activity of control in the next section.

Modelling of the set of exchanges

This section describes the communication between the controller and the infrastructure. We consider the data-plane computation in a graph topology $\mathcal{T} = \{N, L\} \in \mathcal{N} \times \mathcal{N}^2$ with the following:

- $N \in \mathcal{N}$: the set of nodes of the topology.
- $L \in \mathcal{N} \times \mathcal{N}$: the set of links (between two nodes) in the topology.

Set of exchanges

The packets exchanged for the data plane computation were sent through the southbound interface, which is the interface between the switches and the controller. This interface was normalized using the OpenFlow protocol [52].

According to the protocol, the messages are the requests from the switches, commands for the switches, notifications of a port status, and statistics from the switches, $\Sigma = \Sigma_{In} \cup \Sigma_{Out} \cup \Sigma_{Ps} \cup \Sigma_{Mp}$.

The first set, Σ_{In}, corresponds to the "Packet_In" messages, named pin, which are requests from the

switches in the direction of the controller. $\forall pin \in \Sigma_{In} \subset \mathscr{M}_{1,5}$, $pin = (p, S, b, src, dst)$ with the following:

- $p \in \mathbb{N}$: the in-port p.
- $S \in \mathcal{N}$: the switch S.
- $b \in \mathbb{N}$: an identifier named Buffer IDentifier which is tagged to the original packet by the switch.
- src: the IP source address of the packet.
- dst: the IP destination address of the packet.

In response to these requests, a second type of event occurs: commands sent by the controller. These were divided into two categories. First, "Packet_Out", denoted by $pout$, which is used only once. (The switches do not retain the information and will have to ask again to the controller for further actions.) Second, "Flow_Mod", denoted by $fmod$, which is permanent: the switch adds this command to its flow table. Thus, $\Sigma_{Out} = \Sigma_{PO} \cup \Sigma_{FMOD}$ with $\forall pout \in \Sigma_{PO} \subset \mathscr{M}_{1,3}$ we define $pout = (act, S, b)$:

- $act \in \mathbb{N}$: the action ordered, here the port of transmission.
- $S \in \mathcal{N}$: the switch destination of the command.
- $b \in \mathbb{N}$: the buffer ID of the packet.

$\forall fmod \in \Sigma_{FMOD} \subset \mathscr{M}_{1,7}$ then $fmod = (act, S, b, src, dst, idle, type_{fmod})$:

- act, S and b are similar to $pout$.
- src: the IP source address of the packet.
- dst: the IP destination address of the packet.
- $idle \in \mathbb{R}^+$: the storage time of the order by the switch.
- $type_{fmod} \in Add, Delete, Modify$: the type of the instruction.

Because we consider only the computation of the data plane, the actions are limited to the port of transmission of the packets by the switches for the associated flow.

"Port_Status", denoted by ps, is a notification from the switches about the state of their ports. This concerns the evolution of the network topology (at the infrastructure level). Then, $\forall ps \in \Sigma_{Ps} \subset \mathscr{M}_{1,3}$ and thus $ps = (reason, p, S)$.

- $reason \in Add, Delete, Modify$: the reason of the message: Add to notify that the port was added, $Delete$ if the port was removed, and $Modify$ for a modification of the port state.
- $p \in \mathbb{N}$: the considered port.
- $S \in \mathcal{N}$: the switch source of the packet.

Finally, "MultiPart," denoted by mp, is the statistics of the switches sent to the controller. These statistics are sent in response to a request of the controller though "MultiPartRequest", denoted by mq. According to [52] there are several kinds of statistics provided by the switch; in this study, we consider only the statistics related to the flow because they are used by the controller ONOS to remove the path by hand (and thus the controller does not use the $idle$ time parameter of the $fmod$ packet). Then $\forall mp \in \Sigma_{MP} \subset \mathscr{M}_{1,4}$, $mp = (byte, S, src, dst)$:

- $byte \in \mathbb{R}^+$: the number of bytes transmitted for the flow.
- $S \in \mathcal{N}$: the switch source the packet.
- src: the source of the flow.
- dst: the destination of the flow.

A description of the set of exchanges is presented in Table 2.

Path Properties

The consistency of the path set up by the controller was evaluated by the observer according to the three criteria defined below. This corresponds to the structural properties of the controller behaviour defined in [54]. Such properties permit to unify the set of command. As a consequence, in response to a request we do not expect a set of ordered commands but a set of unordered commands which satisfy the structural properties defined just below. A path r installed by a set of n commands $r = (pout_{i,j})_{i \in [1,n], j \in [1,7]} \in \mathscr{M}_{n,7}$ is considered consistent if:

1. There is no loop. $\forall i \in [1,n] \nexists j \ in [1,n], i \neq j | pout_{i,3} = pout_{j,3}$
2. There is no dead node. $\forall i \in [1,n] \exists j \ in [1,n], i \neq j | \mathcal{T}(pout_{i,3}, pout_{i,1}) = pout_{j,3}$

Table 2: Summary of the set of exchanges at the Southbound Interface.

Sender	OpenFlow	Notation	Description
Switch	Packet_In	pin	Request from a switch
	Port_Status	ps	Notification to the controller that the port status of the switch is changed
	MultipartReply	mp	Statistics sent to the controller
Controller	Packet_Out	$pout$	A command sent to the switch
	Flow_Mod	$fmod$	A command which modifies the flow table of the switch
	MultipartRequest	mq	Request for the statistics of a switch by the controller

3. The destination is reached. $\exists i \in [1,n] \mathcal{T}(pout_{i,3}, pout_{i,1}) = pin_{1,4} \& \nexists j \in [1,n], i \neq j | pout_{i,3} = pout_{j,3}$

Impacts of the threats

In case of an attack or failure, the control algorithm returns a biased command. The threat may have several origins in the SDN architecture, as explained in [44] or [23]; however, in this study, we do not consider the isolation of the fault. Thus, we propose to synthesize these different threats into one bias, named $bias$, which leads to an affine-biased Packet_Out $biaspout \in \Sigma_{Out}$ defined as:

$$biaspout = pout + bias \tag{1}$$

With:

- $biaspout \in \Sigma_{Out}$: the biased packet.
- $pout \in \Sigma_{Out}$: the original packet.
- $bias \in \Sigma_{Out}$: the bias.
- $+$ is the operator defined as: $\forall i \in [1, lengh(pout)]\ biaspout_{1,i} = pout_{1,i} + bias_{1,i}$

Temporal notion

The disappearance of command packets is a particular case of bias. This means that no command is sent by the controller. This corresponds mainly to the case of failure. Thus, we propose to introduce a tolerance for the response time of the controller; therefore, we define a time boundary. However, in contrast to [53] and [54] we do not determine a fixed boundary. Here, the boundary evolves in real time to follow the evolution of the traffic, and thus the evolution of the response time of the controller (owing to an increase in the number of requests from the data plane, for example).

In addition, the response times of the controller are recorded in a list $Lt_{Response}$ which is updated at each observation of a path setup. The length of this list is fixed to consider the observed values, which is suitable for the situation and not being polluted by another context (such as an acceleration of the requests of the switches or, an absence of a request, for example). Moreover, the response times of the controller which are longer than the fixed tolerance are ambiguous for the observer because it does not know if they correspond to an attack or the evolution of the traffic. Consequently, such values are not added to $Lt_{Response}$ (and thus, the outliers).

Therefore, to determine the value of the boundary, we propose to analyse the set of previous values and define a tolerance over the worst observed case. We introduce t_{WC}, as follows:

$$t_{WC} \in Lt_{Response} | \forall t \in Lt_{Response} : t \leq t_{WC} \tag{2}$$

Based on this, a boundary is fixed with a tolerance on the worst case observed, and we introduce the tolerance factor $\beta \in \mathbb{R}^+$ as the following:

$$t_{boundary} = \beta \times t_{WC} \tag{3}$$

This parameter determines the tolerance to the potential delay of the controller. This delay might be owing to a DDoS attack; however, if this attack has a limited impact which permits the controller to achieve its task within the tolerance, then no fault is declared by the observer because the activity of the command is satisfied as it respects the tolerance. Consequently, because of the attack, the boundary might increase owing to the evolution of the worst case, and tends toward infinity. Therefore, we propose introducing a constraint formalized as an arbitrarily fixed limit t_{limit}, which corresponds to the physical limit of the evolution of the boundary.

$$timeout = \min(t_{limit}, t_{boundary}) \tag{4}$$

Consequently, we assume that for route r that was installed at time t_r,

$$r \text{ is consistent } \Rightarrow t_r < timeout \quad (5)$$

Observation Logic

The observer's task was to detect the aforementioned bias. Indeed, as a reminder, as we do not consider communication between the observer and the controller we do not evaluate the internal states of the controller but its activity which corresponds to the packets exchanged at its Southbound interface. Hence, the observer is expected to declare that the decisions of the controller are faulty if they do not fulfil the constraints defined in Sections Path Properties and Temporal notion or if they are subjected to an unexpected change (as we consider a deterministic algorithm). Therefore, the observer reacts to the packets sent by the switches and verifies that the responses of the controller are consistent (which corresponds to the packets described in Table 2). Thus, an attack on the controller without a request from the switches will not be detected, as there is no reconsideration of the traffic.

Here, we introduce the following notations: r is a path, $\mathcal{R}$ is the set of paths learned by the observer, and $\mathcal{R}_{Set}$ is the set of paths currently up.

The detection logic is shown in Algorithm 1. The input of the logic is the OpenFlow packets described in Section Set of exchanges. At the observation of a Packet_In, the learning phase starts from line 3: the updates of the data plane are stored until the end of the timeout, as defined in Section Temporal notion, or the path is assumed to be consistent, according to Section Path Properties. The three constraints are resumed in one Boolean variable $cons(r)$ which means that if r satisfies the three constraints, then $cons(r) = true$; otherwise, it is $false$. This learning phase is mandatory as the observer does not have access to the internal states of the controller. It is used for the next phase to verify the consistency of the control and so infer over the states of the control. Indeed, if the route asked at the request is part of $\mathcal{R}$ then the observer verifies that the controller sets a similar route (line 15). In any case, when an observed path is assumed consistent, it is added to the set of paths currently up $\mathcal{R}_{Set}$.

However, as soon as a link evolution (i.e. a link failure) between switches is notified by Port_Status, the previously set up path might be impacted and the global view of the controller permits to install new rules (and a recovery solution) from a central position [4]. There are two possibilities: if the path is up to the observation of Port_Status, part of $\mathcal{R}_{Set}$, then the controller must update the path as soon as possible by uninstalling the previous rules to install a new one. This implies that the observer must restart a learning phase, line 26, and let the controller install a new path from the previous one. The commands might be the deletion of previous rules, line 29, or the installation of a new one, line 31, or the modification of a previous one, line 33. At the end of the timeout, the consistency of the route set up through the observed commands is checked (Line 34). The other possibility concerns the paths learned but are not up anymore. The controller does not react to this packet but to the observation of a request which should lead to the installation of the path by setting up a new one. Hence, the observer restarts the learning phase, which means that for this step, it updates its estimation of the routing table of the controller (line 40).

Moreover, the path installed by the controller expires. In this study, the controller deletes the paths by hand (and does not fix idle_time) when the requested traffic is finished. In addition, the controller requires statistics periodically, through $MultiPart$ packets, to the switches to determine the number of bytes transmitted for each flow. If there is no evolution, the controller removes the path by sending a Flow_Mod packet to delete previously installed rules. Hence, the observer checks the evolution of the number of transmitted bytes (line 37); if there is no evolution, the observer waits for the deletion from the controller and removes these commands from $\mathcal{R}_{Set}$.

In addition, the observer verifies that the duty of the controller is performed in time by controlling the controller with respect to the tolerance time, $timeout$. This tolerance is not the same for each task; therefore, we introduce $timeout_{pin}$, $timeout_{ps}$ and $timeout_{mp}$ which correspond to the reaction time tolerated in the case of a packet in, port status, or multipart.

Algorithm 1: Observer Logic

Input: An OpenFlow packet p.
Data: $\mathcal{R}$: the set of routes.

$p = wait(packet)$;
if $p \in \Sigma_{In}$ **then**
  if $p \notin \mathcal{R}$ **then**
    $r = \emptyset$;
    while $\overline{cons(r)}\&timeout_{pin}$ **do**
      $f = wait(fmod)\&r = r \cup (p, fmod)$;
    if $cons(r)$ **then**
      $\mathcal{R} = \mathcal{R} \cup r$ & $\mathcal{R}_{Set} = \mathcal{R}_{Set} \cup r$;
    else
      return $Fault$;
  else
    $r_{learn} = r | r \in \mathcal{R}$ & $pin(r) = p$ & $r = r_{learn}$;
    while $path(r_{learn}) \neq \emptyset$ & $timeout_{pin}$ **do**
      $f = wait(fmod)$;
      if $f \notin path(r_{learn})$ **then**
        return $Fault$;
      else
        $path(r_{learn}) = path(r_{learn}) \setminus fmod$;
    if $path(r_{learn}) \neq \emptyset$ **then**
      return $Fault$;
    else
      $\mathcal{R}_{Set} = \mathcal{R}_{Set} \cup r$
else if $p \in \Sigma_{Ps}$ **then**
  for $r_{learn} \in \mathcal{R} \mid \exists fmod \in cmd(r_{learn}) \mid port(fmod) = port(p)$ **do**
    $r'_{learn} = r_{learn}$;
    if $r_{learn} \in \mathcal{R}_{Set}$ **then**
      while $timeout_{ps}$ **do**
        $f = wait(fmod)$;
        if $type(f) = delete$ **then**
          $r'_{learn} = r'_{learn} \setminus f'$;
        else if $type(f) = add$ **then**
          $r'_{learn} = r'_{learn} \cup f$;
        else if $type(f) = modify$ **then**
          $r'_{learn} = r'_{learn} \setminus f' \& r'_{learn} = r'_{learn} \cup f$;
      if $\overline{cons(r'_{learn})} || r'_{learn} = r_{learn}$ **then**
        return $Fault$;
      else
        $\mathcal{R} = \mathcal{R} \setminus r_{learn} \& \mathcal{R} = \mathcal{R} \cup r'_{learn}$;
    else
      $\mathcal{R} = \mathcal{R} \setminus r_{learn}$
else if $p \in \Sigma_{MP}$ **then**
  if $byte_{t-2}(p) = byte_{t-1}(p) = byte(p)$ **then**
    while $timeout_{mp}$ **do**
      $f = wait(fmod)$;
      if $fmod \in \mathcal{R}_{Set}$ **then**
        $\mathcal{R}_{Set} = \mathcal{R}_{Set} \setminus fmod$
      else
        return $Fault$

Recover of the attack

The proposed algorithm determines whether the controller is attacked. Following the alarm, the observer must take the lead over from the main controller. Several techniques can be used as the recovery phase presented in [20]: stop the communication between the switches and the faulty controller, and then the switches search to contact the observer as a controller.

However, the techniques which use communication between the controllers, such as in [35], cannot be applied because we consider that the controller is faulty after the alarm. In addition, some methods cannot be applied as the methods proposed in [25] use non-Openflow packets named "Remap" or "Remap-Begin" to lead over the faulty controller.

Performance evaluation

This section evaluates the aforementioned algorithm. First, the metrics used are introduced, the experimental setup is described, and finally, case studies are presented.

Metrics

First, to evaluate the relevance of our detection method, we propose using the number of true positives TP, false positives FP, true negatives TN, and false negatives FN. These values are used to determine some properties of the systems: the precision P and recall R. Precision corresponds to the number of correct alarms (TP) compared to the total number of alarms ($TP + FP$), whereas recall is defined by the number of correct alarms (TP) with respect to the number of alarms that have to be considered ($TP + FN$). Moreover, we also consider the harmonic means of these two metrics, named "F-Measure" and note Fm.

Mathematically, these are computed as follows:

$$P = \frac{TP}{TP + FP} \tag{6}$$

$$R = \frac{TP}{TP + FN} \tag{7}$$

$$Fm = \frac{2 * P * R}{P + R} \tag{8}$$

Second, the reactivity of the detection algorithm is evaluated. We introduced the reactivity of the algorithm

as the time of reaction of the observer to a request from the switches. The objective of our algorithm is not to determine the state of the controller but to verify that the controller responds correctly to the request. In other words, an attack on the controller without traffic in the data plane will not be detected because the service of the controller will not be called. Therefore, we are not attempting to be reactive to an attack, but to a lack of response to a request from a switch.

Experimental set up

The physical topology is illustrated in Fig. 4. The network was simulated using Mininet [55] and the considered topology is presented in Fig. 3. The ONOS controller [50] is on the first machine, whereas the attacker is on another machine that uses the Kali Linux tool. The observer was implemented on the same machine as Mininet and the communication between the controller and switches was captured using the Scapy tool. Notably, the observer did not send packets to the network.

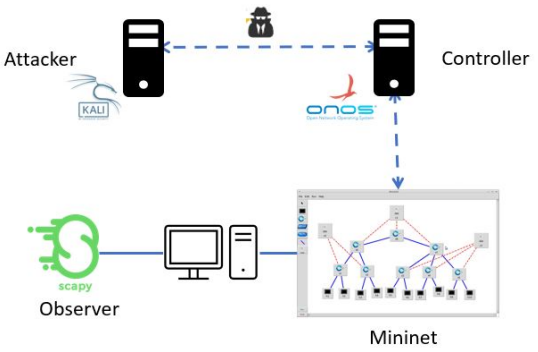


Figure 4: Physical topology

Nominal behaviour of the controller

The observer's reaction when the controller was unfaulty is shown in Fig. 5. In the figure, the decisions of the observer are represented by the following points: zero if there is no alarm and one otherwise. In addition, during the experiment, two failures occurred in the link between the switches. The times of these failures are represented by vertical lines in Fig. 5.

Learning Phase

During the learning phase, the observer observes how the controller reacts and verifies the consistency. In Fig. 5, the learning phase corresponds to the first 10 points, until $t = 105$ s. We develop the learning of the path between 10.0.1.1 and 10.0.1.2. The exchange of packets related to this traffic, captured using Wireshark, is represented on the left of Fig. 6.

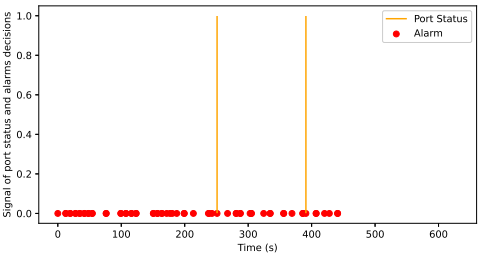


Figure 5: Alarms in case of an unfaulty behaviour.

The flow request, packet 2241, for the route from the host 10.0.1.1 (denoted as pin_{2241}) is observed. Based on this observation, the observer expected a path from the controller. The evolution of the state of the observer is shown on the right side of Fig. 6.

Packet number 2243 is observed and corresponds to $fmod$, denoted by $fmod_{2243}$. These commands are related to the path requested by pin_{2241} and are added to the current observed route $r_{pin,2241}$ through line 6 of the algorithm. Then, the consistency of the new path is verified by line 5. Because the current observed path is not consistent, no decision is finalized, and the observer is still waiting for other commands. This process is the same for frames 2247 and 2251.

Concerning the frame number 2255, denoted as $fmod_{2255}$, the same process is expected to satisfy the three conditions introduced in section Path Properties which means that the condition $cons(r_{pin,2241})$ is satisfied. This corresponds to the final command for setting up the path using the controller. Hence, there was no alarm, as shown in Fig. 5 (point at 0) and the path observed is stored in $\mathcal{R}$ and added to $\mathcal{R}_{Set}$.

Then, at the reception of request 2257, there will be a similar process for the route from the host 10.0.1.2.

In this example, we assumed that the controller was not attacked during the learning phase. Alternatively, the observer learns the wrong, but consistent path. The observer does not determine whether the paths correspond to the results of the control algorithm. However, there is a guarantee that even if the path is not optimal, it will forward traffic, as expected.

Controller

No.	Host Dst	Host src	Time	Switch	Port	Type
2241	10.0.1.2	10.0.1.1	40.564411885	0		OFPT_PACKET_IN
2243	10.0.1.2	10.0.1.1	40.588684529	0	3	OFPT_FLOW_MOD
2247	10.0.1.2	10.0.1.1	40.595801517	3	3	OFPT_FLOW_MOD
2251	10.0.1.2	10.0.1.1	40.606386603	5	4	OFPT_FLOW_MOD
2255	10.0.1.2	10.0.1.1	40.609537361	2	3	OFPT_FLOW_MOD
2257	10.0.1.1	10.0.1.2	40.609916096	2		OFPT_PACKET_IN
2259	10.0.1.1	10.0.1.2	40.616730027	2	2	OFPT_FLOW_MOD
2266	10.0.1.1	10.0.1.2	40.621113890	5	1	OFPT_FLOW_MOD
2270	10.0.1.1	10.0.1.2	40.626496161	4	1	OFPT_FLOW_MOD
2276	10.0.1.1	10.0.1.2	40.645798759	0	1	OFPT_FLOW_MOD

Observer

$\mathcal{R}$	$\mathcal{R}_{Set}$	Evolution of the Expectation	Observation
$\emptyset$	$\emptyset$	$\emptyset$	pin_{2241}
$\emptyset$	$\emptyset$	$r_{2241} = \{fmod_{2243}\}$	$fmod_{2243}$
$\emptyset$	$\emptyset$	$r_{2241} = \{fmod_{2243}, fmod_{2247}\}$	$fmod_{2247}$
$\emptyset$	$\emptyset$	$r_{2241} = \{fmod_{2243}, fmod_{2247}, fmod_{2251}\}$	$fmod_{2251}$
r_{2241}	r_{2241}	$r_{2241} = \{fmod_{2243}, fmod_{2247}, fmod_{2251}, fmod_{2255}\}$	$fmod_{2255}$
r_{2241}	r_{2241}	$\emptyset$	pin_{2257}
r_{2241}	r_{2241}	$r_{2257} = \{fmod_{2259}\}$	$fmod_{2259}$
r_{2241}	r_{2241}	$r_{2257} = \{fmod_{2259}, fmod_{2266}\}$	$fmod_{2266}$
r_{2241}	r_{2241}	$r_{2257} = \{fmod_{2259}, fmod_{2266}, fmod_{2270}\}$	$fmod_{2270}$
$r_{2241} \cup r_{2257}$	$r_{2241} \cup r_{2257}$	$r_{2257} = \{fmod_{2259}, fmod_{2266}, fmod_{2270}, fmod_{2276}\}$	$fmod_{2276}$

Figure 6: Exchanges of packets during traffic between 10.0.1.1 and 10.0.1.2. The packets received and sent by the controller (captured with Wireshark) are shown on the left side of the figure and the vision (understanding) of the observer is shown on the right side.

Running Phase

During this phase, the paths set by the controller are compared to those learned through lines 11–19 of the algorithm.

Moreover, the path set by the controller may evolve in the case of modification of the switch topology. As an example, we sent traffic from 10.0.1.1 in the direction of 10.0.1.2 and during the transmission, we set a link between switches 1 and 2. As observed during the learning phase, this link is part of the path that has been set up by the controller. Therefore, the controller must update it accordingly. The frames exchanged in this situation, as captured using Wireshark, are shown on the left of Fig. 7.

The first task of the observer is to check what learned paths are impacted by this evolution (line 22). The set of impacted paths is denoted by $\mathcal{R}_{Imp}$. Only one is up at the notification of this evolution: the one from 10.0.1.2 in the direction of 10.0.1.1 (the reverse route passes through switches 1, 3, 5, and 6). Therefore, a new route is expected. On the right side of Fig. 7 there is the evolution of the state of the observer.

At the end of the timeout, the consistency of the final route $r_{Ps} = \{fmod_{2259}, fmod_{2266}, fmod_{6329}, fmod_{6334}\}$ was checked. Here, r_{PS} is consistent, and thus we replace r_{2257} in $\mathcal{R}$ and $\mathcal{R}_{Set}$, line 38. Thus, there is no alarm, as shown in Fig. 5.

The observer deletes the other paths impacted by this evolution from its estimation of the routing table of the controller, line 39, as shown on the right side of Fig. 7: the estimation of the routes $\mathcal{R}$ has evolved into $\mathcal{R} \setminus \mathcal{R}_{Imp}$. Hence, the observer restarted the learning path.

In summary, it can be observed from Fig. 5 that there is no error from the observer. As we considered a deterministic algorithm, the observer did not make a mistake. The risk of error occurs during the learning phase to learn a consistent path, which does not correspond to the requirements of the request (for example, a non-optimized path which leads to the exceeding of the criteria of the waiting times).

Case of Attack

In this section, we launch different attacks on the controller and examine the reactions of the observer.

Case of Distributed-DOS

First, we consider a DDoS attack on the controller by saturation of the computational resources. To set up the attack, we used the tool "hping3" for an external attacker.

The consequence of the attack is a delay in the reaction of the controller. Therefore, the detection of such an attack is related to the tolerance of the controller response time and, more precisely, the parameter β defined in Section Temporal notion.

The aim of this study is to compare and analyse the precision and recall for seven different values of β: 1.2, 1.3, 1.5, 2, 3, 4, and 5.

Illustration: A first design of the experiments is shown in Fig. 8. There are three different DDOS with different duration: 10 s, 100 s, and 1000 s. The aim is to analyse the reaction of the observer faced with different types of DDoS. The impact of DDoS is related to its duration. First, the impact of a DDoS that lasted 10 s (respectively 100 s) was to multiply the response time of the controller by up to 100 (respectively 1000). Finally, for 1000 s, the controller became unreachable. The factor of proportionality introduced was indicative based on

Controller

No.	Time	Switch	Port	Type	Command
6283	115.0931…	0		OFPT_PORT_STATUS	
6287	115.0934…	4		OFPT_PORT_STATUS	
6316	115.1542…	4		OFPT_FLOW_MOD	OFPFC_DELETE_STRICT
6329	115.3402…	4	4	OFPT_FLOW_MOD	OFPFC_ADD
6334	115.3444…	3	1	OFPT_FLOW_MOD	OFPFC_ADD

Observer

$\mathcal{R}$	$\mathcal{R}_{Set}$	Evolution of the Expectation	Observation
$\mathcal{R} \setminus \mathcal{R}_{Imp}$	r_{2241}	$r_{\mathrm{PS}} = r_{2241}$	ps_{6283}
$\mathcal{R} \setminus \mathcal{R}_{Imp}$	r_{2241}	$r_{\mathrm{PS}} = r_{2241}$	ps_{6287}
$\mathcal{R} \setminus \mathcal{R}_{Imp}$	r_{2241}	$r_{\mathrm{PS}} = \{fmod_{2259}, fmod_{2266}, fmod_{2276}\}$	$fmod_{6316}$
$\mathcal{R} \setminus \mathcal{R}_{Imp}$	r_{2241}	$r_{\mathrm{PS}} = \{fmod_{2259}, fmod_{2266}, fmod_{2276}, fmod_{6329}\}$	$fmod_{6329}$
$r_{\mathrm{PS}} \cup \mathcal{R} \setminus \mathcal{R}_{Imp}$	r_{PS}	$r_{\mathrm{PS}} = \{fmod_{2259}, fmod_{2266}, fmod_{2276}, fmod_{6329}, fmod_{6334}\}$	$fmod_{6339}$

Figure 7: Exchanges of packet after a link failure between the switches 1 and 2. The packet received and sent by the controller (captured with Wireshark) are shown at the left side of the figure and the vision (understanding) of the observer is shown on the right side.

the experiments; however, we also observed a lower effect, and it should be mentioned that this delay appeared gradually.

```
while True:
        interarrival = randint(3, 15)
        time.sleep(interarrival)

        choice = random.uniform(1,100)
        if choice < 20:
                trafficlength = '126000'
        elif choice >95:
                trafficlength = '12600000'
        else:
                trafficlength = '1260000'

        hostSrc = random.choice(['h1', 'h2', 'h3', 'h4', 'h5'])
        hostDst = random.choice(['1', '2', '3', '4', '5'])

        net.get(hostSrc).cmd("iperf -c 10.0.1." + hostDst + " -n " + longInterA)
net.stop()
```

(a) Mininet parameter

```
#!/usr/bon/env bash

sleep 30

ping -c 1 192.168.56.1
hping3 --flood -k --rand-source -S 192.168.56.101 &
sleep 10
kill $!
ping -c 1 -s 10000 192.168.56.1

sleep 80
ping -c 1 192.168.56.1
hping3 --flood -k --rand-source -S 192.168.56.101 &
sleep 100
kill $!
ping -c 1 -s 10000 192.168.56.1

sleep 120
ping -c 1 192.168.56.1
hping3 --flood -k --rand-source -S 192.168.56.101 &
sleep 1000
kill $!
ping -c 1 -s 10000 192.168.56.1

echo "Fin"
```

(b) DDoS parameter

Figure 8: Design of experiment

The evolution of the alarm signal compared to the attack signal for one experiment is represented by $\beta = 1.2$ in Fig. 9, for $\beta = 2$ in Fig. 10, and for $\beta = 5$ in Fig. 11. In the figures, the decisions of the observer are represented by the following points: zero if there is no alarm and one alternatively, whereas the times of the attack are represented by the curve: zero if there is no attack and alternatively, one.

Precision and recall: A second design of experiments has been designed to analyse the performance of our proposal according to the defined metrics. The experimental design is shown in Fig. 12. First, regarding the data plane requests (left side of Fig. 12) we randomized the inter-arrival of the requests: a value is determined between 3 and 15 s through a uniform law. In addition, the length of the traffic is determined by

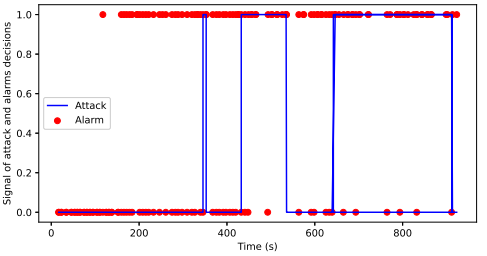


Figure 9: Alarm in case of the parameter is $\beta = 1.2$

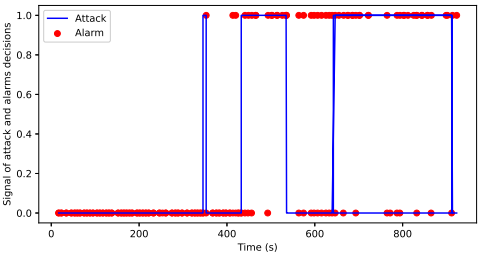


Figure 10: Alarm in case of the parameter is $\beta = 2$

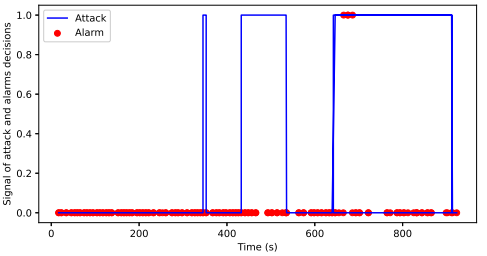


Figure 11: Alarm in case of the parameter is $\beta = 5$

the number of bytes to transmit, and is also random. We divided the case into three possibilities: short traffic (126000 bytes to transmit, 0.1 s is necessary for the transmission), medium traffic (1260000 bytes to transmit, 10 s is necessary for transmission), and longer traffic (12600000 bytes to transmit, 100 s is necessary for transmission). The aim was to determine the impact of an attack on different types of traffic.

With respect to the attack (on the right side of Fig. 12) there are different parameters. The first is the frequency of packet transmission. It was observed that attacks that do not have a "flood" flow rate have no impact on our controller. Therefore, we fixed the flow rate at "flood". However, the time of the attack has an impact, as mentioned previously. We propose to choose uniformly between a short and medium attack (between 20 and 200 s). Similarly, the inter-arrival is random, uniformly between 50 and 150 s.

```
while True:
        interarrival = randint(3, 15)
        time.sleep(interarrival)

        choice = random.uniform(1,100)
        if choice < 20:
                trafficlength = '126000'
        elif choice >95:
                trafficlength = '12600000'
        else:
                trafficlength = '1260000'

        hostSrc = random.choice(['h1', 'h2', 'h3', 'h4', 'h5'])
        hostDst = random.choice(['1', '2', '3', '4', '5'])

        net.get(hostSrc).cmd("iperf -c 10.0.1." + hostDst + " -n " + longInterA)
net.stop()
```

(a) Mininet parameter

```
while((TRUE))
do
        ((lengthtraffic = $((20 + RANDOM % 200))))

        hping3 --flood -k --rand-source -S 192.168.56.101 &
        sleep $lengthtraffic
        kill $!

        ((interarrival = $((50 + RANDOM % 150))))
        sleep $interarrival
done
```

(b) DDoS parameter

Figure 12: Design of experiment

The values of the metrics defined in Section Metrics are shown in Fig. 13.

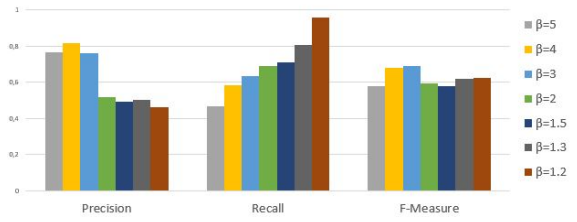


Figure 13: Value of the metrics for the different values of β.

First, we focus on precision. It can be observed that the precision increases with β as it can be observed by comparing Fig. 9 and Fig. 10. This means that the tolerance is extremely strict, with $\beta = 1.2$ compared with $\beta = 2$. Hence, compared to $\beta = 2$, $\beta = 1.2$ is sensitive to the evolution of traffic, and thus, the evolution of the response time of the controller. In general, if β increases, then the rate of false positives decreases owing to the increase in the tolerance. Also, increasing the value of β implies that attacks are not detected until the effect is higher than the tolerance. Furthermore, we consider an attack of 10 s; with $\beta = 5$ the attack is not detected, as can be observed in Fig. 11. This implies that an increase in β leads to a decrease in the rate of true positives.

Second, we focus on recall. Notably, this corresponds to the number of attacks detected compared to the number of attacks that should have been detected. As previously mentioned, the rate of false negatives decreases linearly as β increases. Then, the recall is inversely proportional to β as observed in Fig. 13. Consequently, it can be concluded that tolerance has an impact, and a larger value of β implies a decrease in the number of detected attacks.

The harmonic mean of these two metrics is determined and corresponds to the "F-Measure" part. It can be observed that the parameter β has no impact on the mean which means that the effect on precision is compensated by the effect on recall. Therefore, the choice of β has several consequences. As we have shown, for $\beta < 2$ there is a high rate of false positives. However, $\beta > 2$ implies the risk of not detecting shorter attacks. Moreover, tolerance has an impact on the reactivity of the method, as we will describe in the next sections.

Reactivity: The design of experiments proposed in Fig. 12 has been used. The aim of this section is to describe the influence of β on the reactivity of the observer.

Reactivity is defined as the reaction time of an observer to a request from a switch. This reaction involves observing a path from the controller or timeout. Thus, we considered all the aforementioned points of the experiments (1933) and determined the density of the reactivity of the controller. To determine the density, we split the interval $[0, 0.052]$ (into 52 intervals). The boundary of this interval, 0.052, was chosen because 99% of the value of the timeout was under 0.052 for $\beta = 1.2$, $\beta = 1.3$, and $\beta = 1.5$ and 90% of the value for $\beta = 2$.

The reactivity results are shown in Fig. 14. As expected, the reactivity increased with β. First, the reactivity with a tolerance factor of $\beta = 1.2$ is concentrated at 0.005 s owing to the lack of tolerance to any evolution of the traffic. Then, the reactivity expanded with tolerance (and thus β), even if there is the same rise until $t = 0.005$. This is because the packets are consistent with the tolerance defined with $\beta = 1.5$, thus, they are for the higher value of β. Subsequently, the curves evolve differently, and Poisson impact on the curve, depending on the parameter β, is observed. Furthermore, the amplitude of the curve decreases exponentially (e^{-x}), whereas the extent of the curve increases linearly. This is probably because the requests of the switches are sufficiently spaced, therefore the response time of the controller is independent of the time since the last event. Thus, the response time of the controller follows a Poisson distribution.

Moreover, the offset of the curves may be related to the false-negative rate. According to Fig. 3, the curve related to $\beta = 3$, $\beta = 4$ and $\beta = 5$ expands after 0.052 which means that the timeout increases to tolerate time which is higher than 0.052 and this time corresponds to 80% of the case without the attack. Thus, the tolerance permits tolerance of the case under attack and leads to false negatives.

However, it can also be seen that too much flexibility can lead to undesirable situations and the non-detection of low-intensity DDoS despite a non-negligible delay of the controller. For example, for $\beta = 5$, we can see that the observer is so tolerant that shorter DDoS are not detected. This tolerance is problematic as our worst case observation evolves and the delayed response time. This implies shifting the tolerance bound according to this delayed time. This will allow a larger delay to be covered.

This phenomenon can have undesirable consequences for future detections. Indeed, a delay due to an attack, but which is not detected, has the effect of increasing the uncertainty on the controller's response time. By the proposed approach, this uncertainty will have the effect of increasing the time limit left by the observer (i.e. increasing the value of the bound). Moreover, this increase implies the acceptance of greater delays and by a process of recurrence, this bound can tend towards $+\infty$ and thus prevent the detection of an attack that has an increasingly significant impact.

We will develop this phenomenon mathematically. Let us consider an unfaulty controller. We recall that the bound is calculated from a set of measurements of the controller's response times deemed valid Lt_{Rep}. Among this set, the bound is fixed as a proportion of the worst observed case $t_{Nominal} = max(Lt_{Rep})$ such that $t_{borne} = \beta \times t_{Nominal}$ where β is the protection factor. We will assume $\beta > 1$ for the following. This being said, let's consider a DDoS on the controller. The first delayed response time of the controller is $t_{Attack,1} = \beta \times t_{Nominal}$. It is within the limit and is thus judged as valid by the observer which implies the setting of a new bound $t_{borne} = \beta \times t_{Attack,1} = \beta^2 \times t_{Nominal}$. The second controller delay is again at the limit $t_{Attack,2} = \beta^2 \times t_{Nominal}$ which leads to the fixation $t_{borne} = \beta \times t_{Attack,2} = \beta^3 \times t_{Nominal}$. By recurrence, following the n-th delay the bound will be set to $t_{borne} = \beta^n \times t_{Nominal}$ and as $\beta > 1$, the geometric sequence β^n tends to $+\infty$. The case which has just been formalized corresponds to the worst case and as an illustration, Fig. 15 shows the evolution of the protection factor as a function of n for different values of β. Very clearly, the worst case described is true for any β, but the larger this factor is, the faster the uncertainty increases.

In summary, the parameter β has an impact on the precision, recall, and reactivity of the controller. However, a compromise must be found which depends on the constraints of the system because there is no value which maximizes each parameter. In the case of a precision of at least 0.8, a recall of 0.7, and a reactivity tolerance under 0.052, $\beta = 3$ is appropriate. However, not all criteria are completely fulfilled, and thus, some constraints must be prioritized.

Modification of the intern variables

The attack considered in this section is described in section System and Threat Model. The objective of the attack is to hijack the controller and then redirect the flows in the direction of a particular host. To hijack the controller, we used a Kali Linux tools named "Metasploit" and the vulnerabilities used is the vsftp v2.3.4 backdoor. The results of the attack detection are shown in Fig. 16. In the figure, the decisions of the observer are represented by the following points: zero if there is no alarm and one, alternatively, whereas the time of the attack is represented by the curve: zero if there is no attack and one, alternatively.

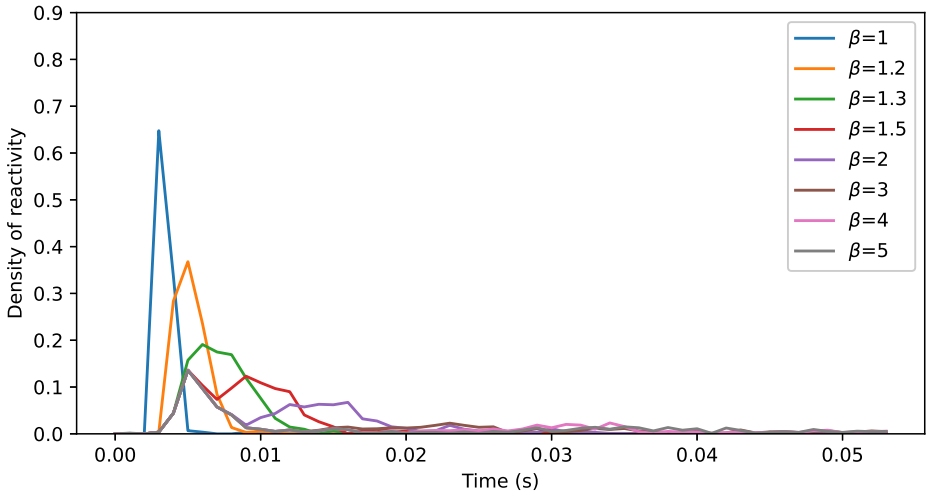


Figure 14: Density of the reactivity of the controller in case of a DDoS attack depending on β.

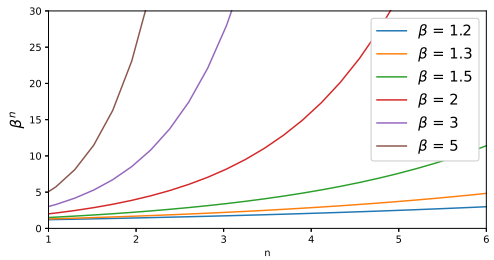


Figure 15: Evolution of β in the worst case.

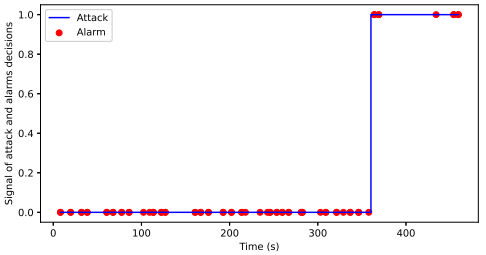


Figure 16: Alarm in case of an attack which is a modification of the intern variables of the controller as described.

Subsequently, we developed the detection of an attack. The exchange of packets captured using Wireshark is shown in Fig. 17.

Packet_In observed at frame number 38419 is denoted as pin_{38419}. We are in the running phase which implies that a similar data plane is learned during the learning phase. To ease the reading, the commands expected are synthesized through the field port and switch, as shown in Fig. 17. As $fmod_{38421}$, $fmod_{38430}$ and $fmod_{38431}$ are part of the expected data plane, the commands are assumed to be consistent through line 18 of the algorithm. Moreover, as there is no more command expected, the path is complete, and no fault is declared through line 22.

Regarding request pin_{38434} the same process runs as for pin_{38419}. The first packets observed are $fmod_{38439}$ and $fmod_{38443}$, which are parts of the expected data plane. Consequently, these decisions are assumed to be consistent and deleted from the set of commands expected through line 18 of the algorithm. Concerning $fmod_{38441}$, the command is part of the expected no-data plane, thus, it is assumed to be inconsistent and a fault is declared on line 16 of the Algorithm. 1 means that the bias has been detected. Therefore, each command bias, which can be formalized as in Section Proposed architecture, is detected.

The detection precision is 1 because we consider a deterministic command algorithm. In addition, the attacker was detected because after the attack, we considered the only path that implied the host 10.0.1.4 and 10.0.1.5. However, this is not the state of the controller, which is evaluated by our algorithm, but the command activity. Therefore, for all the paths requested which do not imply these hosts, the controller sets up a consistent path

Controller

No.	Host Dst	Host src	Time	Switch	Port	Type
38419	10.0.1.1	10.0.1.4	527.7262…	0		OFPT_PACKET_IN
38421	10.0.1.1	10.0.1.4	527.7488…	0	1	OFPT_FLOW_MOD
38430	10.0.1.1	10.0.1.4	527.7632…	1	1	OFPT_FLOW_MOD
38431	10.0.1.1	10.0.1.4	527.7636…	4	1	OFPT_FLOW_MOD
38434	10.0.1.4	10.0.1.1	527.7640…	1		OFPT_PACKET_IN
38439	10.0.1.4	10.0.1.1	527.7717…	4	3	OFPT_FLOW_MOD
38441	10.0.1.4	10.0.1.1	527.7721…	1	10	OFPT_FLOW_MOD

Observer

$\mathcal{R}_{Set}$	Evolution of the Expectation	Observation
$\emptyset$	$r_{\text{Expect}} = r_{0401} = \{p=1, sw=0\}, \{p=1, sw=4\}, \{p=1, sw=1\} \in \mathcal{R}$	pin_{38419}
$\emptyset$	$\{p=1, sw=4\}, \{p=1, sw=1\}$	$fmod_{38421}$
$\emptyset$	$\{p=1, sw=4\}$	$fmod_{38430}$
r_{0401}	$r_{\text{Expect}} = \emptyset$	$fmod_{38431}$
r_{0401}	$r_{\text{Expect}} = r_{0104} = \{p=2, sw=0\}, \{p=4, sw=1\}, \{p=3, sw=4\} \in \mathcal{R}$	pin_{38434}
r_{0401}	$\{p=2, sw=0\}, \{p=4, sw=1\}$	$fmod_{38439}$
r_{0401}	$fmod_{38441} \in r_{\text{Expect}}$	$fmod_{38441}$

Figure 17: Exchanges of packets during an attack as described. The packets received and sent by the controller are shown at the left side (captured with Wireshark) and the vision (understanding) of the observer shown at the right side.

which would then be evaluated as unfaulty by the observer.

Failure of the controller

This section illustrates how the observer reacts to controller failure. The injected traffic was the same as that presented in Fig. 12. Then, the controller was set down at a random moment. The operation was repeated 35 times. The detection results are shown in Fig. 18. In the figure, the decisions of the observer are represented by the following points: zero if there is no alarm and alternatively, one, whereas the time of the failure is represented by the curve: zero if there is no attack and alternatively, one.

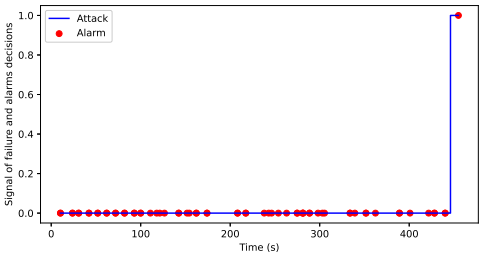


Figure 18: Alarm in case of a failure of the controller.

The failure was correctly detected. This is the case for any failure (because the delay tends to $+\infty$). Thus, the precision and recall of the experiment shown in Fig. 19, depends on the phase without attack, contrary to the DDoS case. Here, as the delay in the case of failure tends to $+\infty$, all the failures are detected, which means that the recall is equal to 1. Nevertheless, the precision increases with β owing to the increase in tolerance, as described in Section Case of Distributed-DOS.

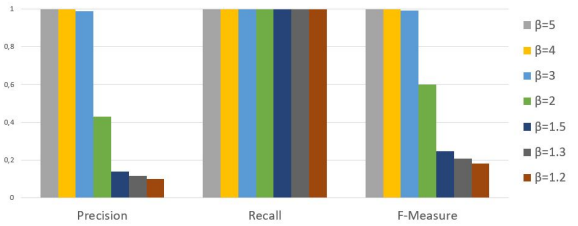


Figure 19: Value of the metrics for the different values of β in case of a failure.

However, the failure is more abrupt than the DDoS attack, which means that the reactivity of the controller evolves. These curves are similar to those shown in Fig. 14: a Poisson effect related to the parameter β can be observed. The difference lies in the standard deviation, which corresponds to the expansion of the curves. A comparison of the expansion, defined as the difference between the two abscissas t_1 and t_2 such that the integral between t_1 and t_2 is 0.90 in the case of failure or a DDoS can be observed in Fig. 3. Before the failure, there is no particular reason, which leads to a delay in the controller (contrary to DDoS which delays the controller), and thus, the timeout is not shifted. In addition, after the failure, the delay tends to $+\infty$; thus, the timeout is not shifted. Therefore, the evolution of the timeout is related to the evolution of the flow's traffic.

Additionally, for $\beta < 1.5$, as shown in Fig. 3, the expansion is similar to the failure and the DDoS. As mentioned in the section Case of Distributed-DOS, the expansion is because of the delay caused by attacks which are inside the boundary. However, for these values of β such delays are not tolerated, implying that the timeout does not expand. In addition, the expansion for $\beta > 3$ is more important in the case of a DDoS than failure, owing to the delays caused by the DDoS which are tolerated. However, the expansion of $\beta = 2$ is smaller for

a DDoS than for a failure because the slope of evolution of the delays of the controller owing to a DDoS is larger than in the case of an increase in the flow's traffic. Consequently, in the case of a DDoS, the delays are not tolerated for $\beta = 2$ whereas they are in the case of an increase in the flow's traffic, which implies that the timeout does not expand in the case of a DDoS. This is also related to the evolution of precision shown in Fig. 13 and Fig. 19. In the case of the DDoS, Fig. 13, the precision does not evolve significantly until $\beta = 3$ (because there is no expansion of the timeout) because of the non-tolerance over the packets delay under DDoS for $\beta < 3$. In the case of failure, as shown in the Fig. 19, there is an evolution between $\beta = 1.5$ and $\beta = 2$ because of the tolerance of flow's traffic evolution, which increases, and thus, the expansion of the timeout also increases.

Table 3: Comparison of the expansion in a case of a failure or a DdoS depending on the parameter β.

	Expansion	
β	Failure	DDoS
1	0.001	0.001
1.2	0.003	0.003
1.3	0.007	0.007
1.5	0.014	0.011
2	0.135	0.168
3	0.148	0.287
4	0.149	0.699
5	0.15	1.218

In addition, if the controller fails but is not requested, then there is no problem for the observer, as the evaluation is on the activity of the control and not on the state of the controller.

Conclusion and Perspectives

In summary, this study aims to introduce an SDN multi controller architecture without an east-west interface. This choice is motivated by the fact that this interface of communication is a security threat according to [22]. Hence, a controller cannot have access to the interns variable of the others. Hence, we proposed to study the observe the activity of the control in order to infer over the state of the control. An observer was added to the controller layer to analyse the activity of deterministic control. Our observer verified that the controller reacts correctly to a request from the switches and to a notification of topology evolution between the switches. In this objective, we specify the activity of the command: the protocol OpenFlow. To specify it we used a well-known formalism : a template defined in [43]. Such formalism puts into the light the causality link between the events. However, contrary to others' works using the notion of Template we do not have any assumptions on the order of the events. Here, we assume that in response to an event (a request for the infrastructure) a set of commands is expected and we do not have any assumptions about the order of transmission of the command. As a consequence, in response to a request we do not know exactly which command is expected. However, we defined structural properties on this set of command which has to be satisfied, by the set of commands, in order to be considered as consistent. It permits to verify that the set of commands observed corresponds to a consistent path. The performance of this method was evaluated in terms of precision, recall, and reactivity. To test our algorithm, we considered three scenarios: a DDoS, modification of the interns variables of the controller, and failure of the controller. Because we considered a deterministic control algorithm, we have observed that there is no error in the detection algorithm. However, the fixation of the time boundary has an impact on the performance: a compromise between the reactivity and the F-measure must be found depending on the time constraint of the considered system. This detection algorithm must be extended to a non-deterministic control algorithm.

Moreover, the future work will consist of completing the method to lead over the malicious controller by another controller. This is compatible with the methods proposed in the literature, such as the recovery phase presented in [20]: stop the communication between the switches and the faulty controller, and then the switches search to contact the observer as a controller. In this case, the observer becomes a second controller and provides instructions to the switches.

Acknowledgment

This work was supported in part by the French PIA project "Lorraine Université d'Excellence", reference ANR-15-IDEX-04-LUE.

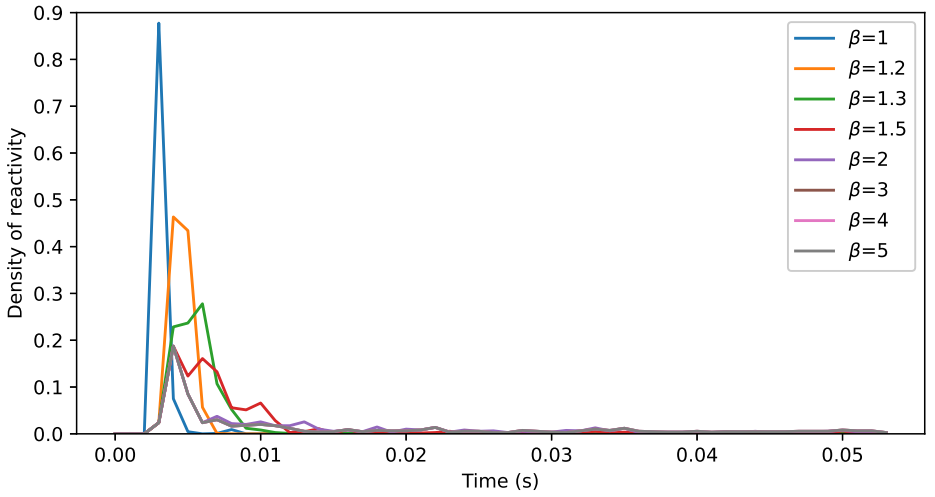


Figure 20: Reactivity of the controller in case of a failure.

References

[1] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo et al. "Software-defined networking: A comprehensive survey". *Proceedings of the IEEE*, vol. 103, no. 1, 14–76, 2015.

[2] Nick McKeown, Tom Anderson, Hari Balakrishnan et al. "Openflow: Enabling innovation in campus networks". *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, 69–74, mar 2008.

[3] Rahim Masoudi and Ali Ghaffari. "Software defined networks: A survey". *Journal of Network and Computer Applications*, vol. 67, 1–25, 2016.

[4] Jehad Ali, Gyu-min Lee, Byeong-hee Roh, Dong Kuk Ryu and Gyudong Park. "Software-defined networking approaches for link failure recovery: A survey". *Sustainability*, vol. 12, no. 10, 2020.

[5] Changhoon Yoon, Taejune Park, Seungsoo Lee et al. "Enabling security functions with sdn: A feasibility study". *Computer Networks*, vol. 85, 19–35, 2015.

[6] Marwan Ali Albahar. "Recurrent neural network model based on a new regularization technique for real-time intrusion detection in sdn environments". *Security and Communication Networks*, vol. 2019, 8939041, 2019.

[7] Seungwon Shin, Lei Xu, Sungmin Hong and Guofei Gu. "Enhancing network security through software defined networking (sdn)". In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, 2016.

[8] Surbhi Saraswat, Vishal Agarwal, Hari Prabhat Gupta et al. "Challenges and solutions in software defined networking: A survey". *Journal of Network and Computer Applications*, vol. 141, 23–58, 2019.

[9] Fetia Bannour, Sami Souihi and Abdelhamid Mellouk. "Distributed sdn control: Survey, taxonomy, and challenges". *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, 333–354, 2018.

[10] Yuan Zhang, Lin Cui, Wei Wang and Yuxiang Zhang. "A survey on software defined networking with multiple controllers". *Journal of Network and Computer Applications*, vol. 103, 101–118, 2018.

[11] Tao Hu, Zehua Guo, Peng Yi, Thar Baker and Julong Lan. "Multi-controller based software-defined networking: A survey". *IEEE Access*, vol. 6, 15980–15996, 2018.

[12] Arash Shaghaghi, Mohamed Ali Kaafar, Rajkumar Buyya and Sanjay Jha. *Software-Defined Network (SDN) Data Plane Security: Issues, Solutions, and Future Directions*, pages 341–387. Springer International Publishing, Cham, 2020.

[13] Sandra Scott-Hayward, Gemma O'Callaghan and Sakir Sezer. "Sdn security: A survey". In *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*, pages 1–7, 2013.

[14] Nada Mostafa Abd Elazim, Mohamed A. Sobh and Ayman M. Bahaa-Eldin. "Software defined networking: Attacks and countermeasures". In *2018 13th International Conference on Computer Engineering and Systems (ICCES)*, pages 555–567, 2018.

[15] Juan Camilo Correa Chica, Jenny Cuatindioy Imbachi and Juan Felipe Botero Vega. "Security in sdn: A comprehensive survey". *Journal of Network and Computer Applications*, vol. 159, 102595, 2020.

[16] Jagdeep Singh and Sunny Behal. "Detection and mitigation of ddos attacks in sdn: A comprehensive review, research challenges and future directions". *Computer Science Review*, vol. 37, 100279, 2020.

[17] Azath Mubarakali and Abdulrahman Saad Alqahtani. "A survey: Security threats and countermeasures in software defined networking". In *2019 IEEE 2nd International Conference on Information and Computer Technologies (ICICT)*, pages 180–185, 2019.

[18] Philip Porras, Seungwon Shin, Vinod Yegneswaran et al. "A security enforcement kernel for openflow networks". In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 121–126, New York, NY, USA, 2012. Association for Computing Machinery.

[19] Jehad Ali, Byeong-hee Roh, Byungkyu Lee, Jimyung Oh and Muhammad Adil. "A machine learning framework for prevention of software-defined networking controller from ddos attacks and dimensionality reduction of big data". In *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 515–519, 2020.

[20] Paulo Fonseca, Ricardo Bennesby, Edjard Mota and Alexandre Passito. "A replication component for resilient openflow-based networking". In *2012 IEEE Network Operations and Management Symposium*, pages 933–939, 2012.

[21] Abdelouahid Derhab, Mohamed Guerroumi, Mohamed Belaoued and Omar Cheikhrouhou. "Bmc-sdn: Blockchain-based multicontroller architecture for secure software-defined networks". *Wireless Communications and Mobile Computing*, vol. 2021, 9984666, 2021.

[22] Diego Kreutz, Fernando M.V. Ramos and Paulo Verissimo. "Towards secure and dependable software-defined networks". In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 55–60, New York, NY, USA, 2013. Association for Computing Machinery.

[23] Ijaz Ahmad, Suneth Namal, Mika Ylianttila and Andrei Gurtov. "Security in software defined networks: A survey". *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, 2317–2346, 2015.

[24] Rohit Kumar Das, Fabiola Hazel Pohrmen, Arnab Kumar Maji and Goutam Saha. "Ft-sdn: A fault-tolerant distributed architecture for software defined network". *Wireless Personal Communications*, vol. 114, no. 2, 1045–1066, 2020.

[25] Yang Wang, Tao Hu, Guangming Tang, Jichao Xie and Jie Lu. "Sgs: Safe-guard scheme for protecting control plane against ddos attacks in software-defined networking". *IEEE Access*, vol. 7, 34699–34710, 2019.

[26] Ricardo Macedo, Rafael de Castro, Aldri Santos, Yacine Ghamri-Doudane and Michele Nogueira. "Self-organized sdn controller cluster conformations against ddos attacks effects". In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2016.

[27] Shailendra Rathore, Byung Wook Kwon and Jong Hyuk Park. "Blockseciotnet: Blockchain-based decentralized security architecture for iot network". *Journal of Network and Computer Applications*, vol. 143, 167–177, 2019.

[28] Shanqing Jiang, Lin Yang, Xianming Gao et al. "Bsd-guard: A collaborative blockchain-based approach for detection and mitigation of sdn-targeted ddos attacks". *Security and Communication Networks*, vol. 2022, 1608689, 2022.

[29] Chao Qi, Jiangxing Wu, Hongchao Hu et al. "An intensive security architecture with multi-controller for sdn". In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 401–402, 2016.

[30] Xiong Liu, Haiwei Xue, Xiaoping Feng and Yiqi Dai. "Design of the multi-level security network switch system which restricts covert channel". In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pages 233–237, 2011.

[31] Ping Li, Songtao Guo, Jiahui Wu and Quanjun Zhao. "Blockrev: Blockchain-enabled multi-controller rule enforcement verification in sdn". *Security and Communication Networks*, vol. 2022, 7294638, 2022.

[32] Hui Yang, Yongshen Liang, Jiaqi Yuan et al. "Distributed blockchain-based trusted multidomain collaboration for mobile edge computing in 5g and beyond". *IEEE Transactions on Industrial Informatics*, vol. 16, no. 11, 7094–7104, 2020.

[33] Jun-Huy Lam, Sang-Gon Lee, Hoon-Jae Lee and Yustus Eko Oktian. "Securing distributed sdn with ibc". In *2015 Seventh International Conference on Ubiquitous and Future Networks*, pages 921–925, 2015.

[34] Fengjun Shang, Yan Li, Qiang Fu et al. "Distributed controllers multi-granularity security communication mechanism for software-defined networking". *Computers & Electrical Engineering*, vol. 66, 388–406, 2018.

[35] Muhammad Faraz Hyder and Muhammad Ali Ismail. "Securing control and data planes from reconnaissance attacks using distributed shadow controllers, reactive and proactive approaches". *IEEE Access*, vol. 9, 21881–21894, 2021.

[36] Heng Zhang, Zhiping Cai, Qiang Liu et al. "A survey on security-aware measurement in sdn". *Security and Communication Networks*, vol. 2018, 2459154, 2018.

[37] Rochak Swami, Mayank Dave and Virender Ranga. "Software-defined networking-based ddos defense mechanisms". *ACM Comput. Surv.*, vol. 52, no. 2, apr 2019.

[38] Jing Zheng, Qi Li, Guofei Gu et al. "Realtime ddos defense using cots sdn switches via adaptive correlation analysis". *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 7, 1838–1853, 2018.

[39] Shuhua Deng, Xing Gao, Zebin Lu, Zhengfa Li and Xieping Gao. "Dos vulnerabilities and mitigation strategies in software-defined networks". *Journal of Network and Computer Applications*, vol. 125, 209–219, 2019.

[40] Nitheesh Murugan Kaliyamurthy, Swapnesh Taterh, Suresh Shanmugasundaram et al. "Software-defined networking: An evolving network architecture—programmability and security perspective". *Security and Communication Networks*, vol. 2021, 9971705, 2021.

[41] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin and Kuang-Yuan Tung. "Intrusion detection system: A comprehensive review". *Journal of Network and Computer Applications*, vol. 36, no. 1, 16–24, 2013.

[42] R. Sekar, A. Gupta, J. Frullo et al. "Specification-based anomaly detection: A new approach for detecting network intrusions". In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 265–274, New York, NY, USA, 2002. Association for Computing Machinery.

[43] Deepa N. Pandalai and L.E. Holloway. "Template languages for fault monitoring of timed discrete event processes". *IEEE Transactions on Automatic Control*, vol. 45, no. 5, 868–882, 2000.

[44] Seungsoo Lee, Changhoon Yoon, Chanhee Lee et al. "Delta: A security assessment framework for software-defined networks.". In *Network and Distributed System Security (NDSS) Symposium*, 2017.

[45] Christian Röpke and Thorsten Holz. "Sdn rootkits: Subverting network operating systems of software-defined networks". In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, pages 339–356, Berlin, Heidelberg, 2015. Springer-Verlag.

[46] Seungwon Shin and Guofei Gu. "Attacking software-defined networks: A first feasibility study". In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 165–166, New York, NY, USA, 2013. Association for Computing Machinery.

[47] Seungwon Shin, Yongjoo Song, Taekyung Lee et al. "Rosemary: A robust, secure, and high-performance network operating system". In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 78–89, New York, NY, USA, 2014. Association for Computing Machinery.

[48] Sungmin Hong, Lei Xu, Haopei Wang and Guofei Gu. "Poisoning network visibility in software-defined networks: New attacks and countermeasures.". In *Network and Distributed System Security (NDSS) Symposium*, volume 15, pages 8–11, 2015.

[49] Talal Alharbi, Siamak Layeghy and Marius Portmann. "Experimental evaluation of the impact of dos attacks in sdn". In *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–6, 2017.

[50] ONF. *https://opennetworking.org/onos/, last visited the 07/09/2021*.

[51] Christos Bouras, Anastasia Kollia and Andreas Papazois. "Teaching 5g networks using the onos sdn controller". In *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 312–317, 2017.

[52] ONF. *OpenFlow Specification v1.3 https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf, last visited the 14/09/2021*, June, 2012.

[53] Loïc Desgeorges, Jean-Philippe Georges and Thierry Divoux. "Detection of security and safety threats related to the control of a sdn architecture". In *4th IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control, CESCIT 2021*, 2021.

[54] Loïc Desgeorges, Jean-Philippe Georges and Thierry Divoux. "A technique to monitor threats in sdn data plane computation". In *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6, 2021.

[55] Rogério Leão Santos de Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda and Ligia Rodrigues Prete. "Using mininet for emulation and prototyping software-defined networks". In *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6, 2014.