Journal of Ubiquitous Systems & Pervasive Networks
Volume 3, No. 1 (2011) pp. 00-00

Dynamic Segmentation, Configuration, and Governance of SDN

Mohammed Alabbad [a], Ridha Khedri [b]*

[a] *Communications and IT Research Institute, King Abdulaziz City for Science and Technology (KACST), Riyadh, 11442, Saudi Arabia*
[b] *Dept. of Computing and Software, McMaster University, Hamilton, Ontario, Canada*

Abstract

Software Defined Networks (SDN) is a networking paradigm that helps transform networks by breaking away from the restrictive constraints put by networking hardware used in traditional non-SDN networks. They bring improved agility, scalability, and programmability of the control and the switching of the traffic. The challenges of structuring the SDN data plane for security still necessitate further investigation especially to deal with dynamic SDN networks. The use of the Robust Network and Segmentation (RNS) algorithm, which is based on Product Family Algebra, is essential for implementing layered defence and segmentation strategies to compartmentalize the networks and attain an access-control secure network. In this paper, we present an additional plane in charge of the configuration and governance of SDN data planes that we call Dynamic Configuration and Governance (DCG) plane. It is intended to give agility to dynamic networks. It implements the RNS algorithm in the SDN environment. Moreover, we propose and assess three architectures that use the DCG plane. The assessment results identify an architecture that is suitable for dynamic networks and another for networks that are more stable regarding changes to policies and network topology.

Keywords: *Dynamic SDN, SDN Architecture, Secure SDN, SDN Segmentation, Formal Methods for Network Segmentation*

1. Introduction

Computer networks consist of connected communicating resources that are protected by a security system. The goal of this system is to ensure proper access to resources and to enhance their availability and integrity. *Firewalls* play an important role in this security system by enforcing security policies that enable only legitimate users to access resources. Other mechanisms such as Intrusion Detection System (IDS) can also be a part of the security system. Firewalls are network security devices that monitor incoming and outgoing network traffic and decide whether to allow or block specific traffic based on given security rules. The operation of determining which set of resources to be placed under each of the firewalls is commonly referred to as segmentation. Network security policies are deployed on network devices that could be switches or firewalls. While in traditional networks the network devices encompass the control plane and the forwarding plane, in Software Defined Networks (SDN) there is a separation between these planes. This decoupling of the two planes enhances the modifiability due to the separation of concerns; the control is separated from the forwarding.

SDN is a network architecture that allows the control of a network using software applications, which takes the stateful control to higher levels by involving several of the attributes of the network and its traffic into the control. *OpenFlow* protocol [1] is a communication protocol that gives the control plane access to the forwarding/data plane devices. The control plane decides on how the traffic is handled using a dedicated program called the controller. The data plane devices forward traffic based on the decisions of the control plane. This separates the concern of the control from that of the execution of the control decisions. This separation of concerns has the benefit of allowing the controller to dynamically and simultaneously manage multiple network nodes. Moreover, it brings, as stated above, advantages to network management such as the ease of modifiability and the ease of testing. These advantages come with several challenges especially related to the security of the network [2, 3] such as

*Corresponding author. Tel.: +1-905-525-9140 ext. 23163
Fax: +1-905-524-0340; E-mail: khedri@mcmaster.ca

DOI: 10.5383/JUSPN.03.01.000

the vulnerability of the network to man-in-the-middle attack and Denial-of-Service (DoS) attacks.

In a network of computing resources, a security system to be effective has to adhere to the strategies of layered protection (i.e., layered defence) and segmentation (i.e., compartmentalization) [4, 5]. Layered protection means having multiple firewalls consecutively on the traffic path. On the other hand, segmentation means grouping resources into clusters of "similar" security requirements, which would help in providing the resources in a segment with the same defence mechanisms. The two strategies are proved to be effective for securing network resources [6, 7]. By adopting the two strategies we end up having resources segmented into different segments that are protected by layers of firewalls. Moreover, segments are placed into the network based on the strength of their security requirements. Therefore, segments with higher security requirements are placed deeper in the network while segments with lower requirements, such as the Demilitarized Zone (DMZ), are placed close to the network's entry point.

The above two strategies have been formalised and implemented in the Robust Network and Segmentation Algorithm (RNS) [8]. We briefly present the RNS algorithm in Section 3. The RNS algorithm provides a systematic approach to segment network resources into clusters and provides the network topology that indicates the placement of these clusters in the network. Given a set of resource policies, the RNS algorithm invoking algebraic calculations generates the topology of the network in which the resources under consideration ought to be organized. Moreover, it generates the policies that have to be enforced by the firewalls. The algorithm can be used in SDN environment to create the data plane topology and to assign policies to be enforced by the switches. Moreover, it could also be used to segment large networks into multiple sub-networks with multiple controllers. The RNS algorithm and its interfaces to the data plane and the control plane form a module for dynamic configuration and governance of the network. We call this module Dynamic Configuration and Governance (DCG) plane. It forms a layer at the same level as the *Application Plane* as shown in Figure 1.

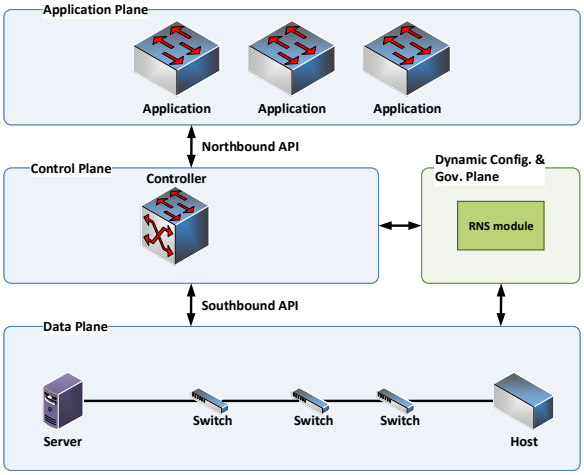

Fig. 1: SDN Architecture and the module running the RNS algorithm

Today networks are very dynamic. It is typical within a modern network to have resources that are added and removed frequently, which leads to a change of the network. The RNS algorithm can be used in real-time for this dynamic configuration of the network as it evolves. For instance, resources in Internet of Things (IoT) [9] are very dynamic: frequently connect to and disconnect from the network. This leads to the necessity of dynamic control on the access to the network. A policy that is defined for the network at a time t might become irrelevant at a later time $t + \delta(t)$. Recalculating the topology of the network and the policies for its firewalls as the involved resources or their policies change leads to changes in the network topology. The latter change entails reconfiguration of the control at the control plane, or data plane if some of the control is delegated to the switches. Hence, it is essential to assess the effect of a dynamic amendment of the control on the network performance. Two of the major questions related to dynamic networks that we aim at addressing are the following:

1. How can we separate the management of the dynamic aspects of an SDN in a real-time setting from the control and the data related issues? For this question, we apply the design principle of separation of concern and we propose an additional plane (DCG plane) that specialises in reconfiguring the network following changes to its resources or their policies.
2. What SDN architecture is the most suitable for dynamic networks? For this question, we present and discuss the results of the assessment of the three SDN architectures under consideration.

To assess the efficiency of the use of DCG plane that is supported by the RNS algorithm in SDN. For this purpose, we build three SDN architectures using *mininet* [10] that conform to the topology calculated by the RNS algorithm that we present in Section 3. Since in SDN network the firewalls can be placed either at the control plane or the data plan (see the discussion in Section 2.), we conceive the following three possible implementations for the firewalls calculated by the RNS algorithm:

- **Architecture 1**: A single and centralised stateful firewall located at the control plane. The firewall governs all the data plane switches. Figure 5 illustrates this architecture.
- **Architecture 2**: Multiple distributed stateful firewalls located at the control plane. Each of the firewalls is assigned to a unique switch at the data plane. Hence, we have as many firewalls as switches. This architecture is presented in Figure 6
- **Architecture 3**: Multiple distributed stateful firewalls located at the data plane. Each switch is transformed into a stateful firewall. This is made possible by the usage of the data plane abstraction, *BEBA software switch* [11], which is an extension of *OpenState* [12]. Figure 7 shows this architecture.

To assess the usage of the RNS algorithm, as an essential component of the DCG plane to handle dynamic changes to a network, within the three above architectures. We aim at determining the most appropriate architecture to use with the RNS algorithm for SDN environment. We look at the performance of the above architectures. We consider the following performance attributes:

1. Setup cost: It is evaluated by the number of packets exchanged between control and data planes during the setup phase. We use *Wireshark* [13] to count the number of exchanged packets.
2. Reachability: To test the effectiveness of the enforced policies, we use *ping* [14] utility such that each host tries to reach every other host in the network.
3. Response time or latency: We use *ping* [14] utility to find out the time needed for the communication between two selected nodes.
4. Bandwidth: We assess this performance parameter using iPerf [15] *tcp test* to obtain the bandwidth for the link under consideration.
5. Latency variation(jitter): We use *iPerf udp* test to get the jitter for the link under consideration.
6. Resilience to topology change: It is measured by the number of packets exchanged to fulfil an intended topology change.

1.1. Main contribution of the paper

The contribution of the paper is twofold. First, it illustrates the use of the RNS algorithm as an essential component of the DCG plane in SDN. It shows how the algorithm is essential for a dynamic SDN. We give the details on the use of the RNS algorithm in SDN for the implementation of the three architectures under consideration. Second, we assess these architectures to capture their drawbacks and strengths. This helps us identify the most appropriate architecture for using RNS algorithm in SDN. We find out that when the network is very dynamic, Architecture 2 is the most appropriate. For a relatively stable network, Architecture 3 is the most appropriate. Although it has the highest cost in the setup and the update phases, in the operation phase it does not need any communication between the control and the data plane. While some of the material of the paper has been presented in [16], this paper brings additional details on the background material, on the techniques used in the implementation of the architecture, and additional assessments of the proposed architectures.

1.1.1. Structure of the paper

The structure of the paper is as follow. In Section 2., we present the background and review the related work. In Section 3., the RNS algorithm is presented along with its implementation. In Section 4., we present three architectures for the implementation of firewalls. Section 5. presents the testbed and selected use cases for the experimentation. In Section 6., we show and discuss the assessment and its results. Finally, in Section 7., we conclude and point for our future work.

2. Background and Related Work

2.1. Background

The background for the work presented in this paper covers SDN and stateful data plane. In the following, we cover these topics.

2.1.1. Software Defined Network (SDN)

In traditional networks, the control plane and the data plane are combined in the same network device. Therefore, network administrators need to configure each device separately. This approach makes traditional networks hard to setup, maintain and manage. Moreover, traditional networks cannot cope with the demands of modern networks such as dynamic changes and scalability. SDN separates or decouples the control plane from the data plane. The control or the management is centralized and has a global view of the network. Data plane devices (e.g., switches) are basically dummy forwarding devices. They forward traffic based on rules specified remotely. The rules could be coming from applications and are triggered by information extracted from traffic or traffic events [3].

SDN architecture consists of three planes: application, control, and data planes [3, 17] as shown in Figure 1. In this work, we propose an additional plane to be added which is the DCG plane. Each plane has its own specific functionality. An SDN network has at least a single controller at the control plane, a northbound application programming interface (API) between the control plane and the application plane, and a southbound interface between the control and data planes. We propose an interface between the control plane and the DCG plane as well as an interface between the data plane and the DCG plane.

Data plane has network devices (e.g., switches) that forward packets without taking decisions on their own, and they communicate with the controller using southbound APIs (e.g., *OpenFlow* protocol). Each *OpenFlow* enabled switch has a flow table. Each entry in the flow table has matching fields, action, and counters. The matching fields can be any of the packet header attributes such as source mac address, source IP, destination mac address, destination IP, destination port, protocol, etc. A flow table entry can execute many actions including forward the packet on a specific port, forward packet to controller, or drop. Flow table entries have priority to specify the order of the evaluation. An entry with high priority gets evaluated first, and an entry with low priority is evaluated later. The development of SDN architecture allowed for the introduction of software switches. The most used of which is *open vSwitch* [18].

The southbound API is the interface between the control plane and the data plane network devices. *OpenFlow* is the most accepted and used southbound API. There are multiple *OpenFlow* messages that can be sent from the data plane devices to the controller including *packet-in*, *switch features reply*, *flow remove*, etc. And the *OpenFlow* messages from the controller to data plane switches include *packet-out* and *add flow*.

SDN control plane runs the network operating system (NOS) which has a global view of the network and configures network devices based on policies and commands defined by applications [2, 3, 19]. It also abstracts the low level/data plane network for the application plane. There are two types of controller architectures: centralized and distributed. The centralized controller architecture has a single controller responsible for managing data plane devices. Examples of centralized controllers are Ryu [20], POX [21], and floodlight [22]. SDN distributed controller architecture has multiple controllers with interfaces between them. ONOS [23] adopts an architecture that distributes the controller. For further information on SDN controllers, we refer the reader to [24].

2.1.2. Stateful Data Plane

In a typical SDN architecture, data plane switches are stateless. For SDN to handle stateful communications, switches need to forward heavy traffic to the controller. This results in a controller overhead and latency in response time. Stateful data plane

approaches allow data plane switches to track stateful connections and therefore take some load off the controller which results in less latency [3, 25].

OpenState [12] is a stateful data plane abstraction that extends *OpenFlow* to handle stateful connections at the data plane. The motivation behind *OpenState* is that some simple operations can be done based on the switch knowledge and can be delegated to the switches, therefore allowing the controller to focus on global network decisions. The controller is still informed of the delegated operations and remains in control of the switches. As shown in Figure 2, each switch holds two tables: state table and eXtensible Finite State Machine (XFSM) table. The state table keeps track of the state of the connection. XFSM is a Mealy Machine and the XFSM table is the tabular representation of its transition function. For a packet and its current state, *OpenState* uses an XFSM table to identify the action to be taken and the new state. In this paper, we use *BEBA software switch* [11] which extends *OpenState* to handle TCP flags in the implementation of Architecture 3.

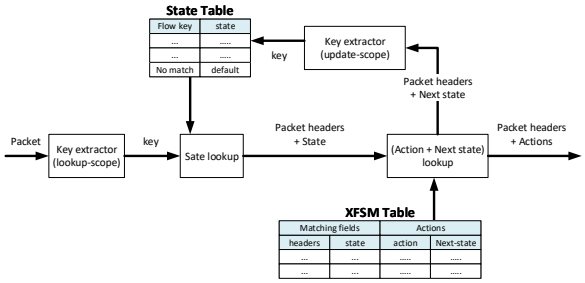

Fig. 2: Packet Flow Diagram in *OpenState*

There are other approaches (e.g., Flow-level State Transitions (FAST) [26], Stateful Data Plane Architecture (SDPA) [27]) that use more than two tables. We refer the reader to [25] for other approaches for building a stateful data plane.

2.2. Related Work

In this section, we provide coverage of the literature related to the topics relevant to this paper. We cover SDN architectures that have been proposed, the challenges within dynamic SDN, and the architectures to implement firewalls in an SDN environment.

2.2.1. SDN architecture

In the literature, there is an emphasis on the whole SDN architecture (e.g., [28, 29]) and on control plane structure (e.g., [23, 30]). While there is a large literature dealing with the controller placement at the control plane, the authors of [19] indicate a shortage of research work on the placement of resources and switches at the data plane. One of the topics discussed in regards to SDN data plane structure is that of data plane flexibility [19]. Data plane flexibility can refer to many issues including adding and removing resources (i.e., changing the topology). The issue of micro-segmentation and slicing is also one of the topics discussed in data plane structure [31]. Network slicing and micro-segmentation are two related concepts concerning isolating parts of the network from each other. Network slicing is the instantiation of a complete end-to-end logical self-contained network with all its functionalities for a specific service or application. Micro-segmentation provides finer-grained isolation of network parts. It allows for dividing networks into finer segments up to an individual machine level and defining security control policies for each segment and therefore limiting lateral traffic movement between segments.

In this paper, we focus on using the topology and the firewall policies provided by the RNS algorithm (presented in Section 3.) to structure the data plane. As far as we know, there are no systematic approaches in the literature that provides guidance on how to, automatically and without human involvement, segment the resources at the data plane.

2.2.2. Dynamic SDN

Resources in modern networks are dynamic. They join and leave the network at any time. This is related to the flexibility of SDN data plane discussed in [19]. We also find that many approach dynamic networks by focusing on routing and load-balancing in the control plane [32, 33] and data plane [34]. Papers [32–34] focus on the dynamic aspect of packet handling. In contrast, we are considering a dynamic topology of the network: Resources get added and removed and the structure of the networks changes by changing the locations of the switches and their policies. The dynamism that we are considering is related to the network topology and the access policies that regulate its access.

Modern networks and related technologies such as IoT, 5G, SDN and Network Function Virtualization (NFV) have experienced exponential growth in the last few years and are expected to grow further in the future. However, security challenges are some of the factors that limit the growth of such technologies. These challenges are the result of the nature of the vulnerabilities of the environment. One of the characteristics of such an environment is its dynamism. To mitigate against the changing security threats and the dynamic nature of the environment, security solutions need to be adaptive [35]. The static security solutions of traditional networks are not sufficient to provide security for such changing and evolving environment. Therefore, the focus of research should be on adaptive security [35]. In our work, the DCG plane provides such a feature. It allows to automatically reconfigure a network as the security situation requires.

For example, we find in [36, 37] that SDN paradigm is mostly deployed to static data centres topology. The application of SDN into dynamic networks such as spontaneous Wireless Mesh Network (WMN) [36] and Mobile Adhoc Network (MANET) [37–39] is recently emerging. In [36], a middleware at every node is proposed that provides better management of WMN networks as well as allowing the network to be flexible, dynamic, and scalable. The approach employs multiple SDN controllers, one for each collection of nodes or what is referred to as WMN islands. The focus of these studies is to use SDN in dynamic networks. However, the focus of our work is to structure networks topologies for security. Our work can be used with these approaches to structure dynamic networks for security.

2.2.3. SDN Firewalls

In the literature, we find many architectures to implement firewalls in an SDN environment. These architectures present designs of firewalls in SDN that are either stateless or stateful, centralised or

distributed, or whether the firewalls reside at the data plane or the control plane.

2.2.3.1. Single stateless firewalls
We find in [40–42] designs of single stateless firewalls. In [42], the rules are entered individually and given a name by the user through the Command-line interface (CLI), which makes this architecture limited in handling a policy with a large number of rules. Moreover, in this architecture, all traffic is handled by the firewall at the control plane except for limited deny rules which are inserted in the switch flow tables. This manual approach for entering the rules would have limited usage in a large dynamic network. It has a scalability limitation as all the traffic is assessed by the firewall at the control plane. There is a high coupling between all the switches and the firewall. We did not consider this architecture as we built on later improved ones and because we are interested in stateful firewalls.

2.2.3.2. Distributed stateless firewalls at the control plane
The architecture presented in [43] uses distributed stateless firewalls. The architectures in [44–46] while they use only a single firewall at the control plane, they adopt the same approach as what we found in [43] of inserting deny rules in the switches at the data plane. In the architecture presented in [43], the controller assigns a firewall module for each switch. Each firewall inserts deny rules in the switch, such that the switch drops unwanted traffic without forwarding the packets to the controller. When a packet arrives at a switch with no entry in its flow table to handle the packet, it forwards the packet to the firewall at the controller. The latter firewall instructs the switch on how to forward the packet and insert an entry to handle such packets in the future. Such an approach is basic and does not take the state of the connection into account and blocks traffic in both directions. However, in real settings, it is desirable to allow traffic in one direction and block traffic in the other direction. While these architectures enable distributed firewalls, they are stateless, which does not meet our needs to implement a stateful firewall. Stateless firewalls are not able to block packets that are not part of an established connection and therefore not able to protect from fake packets. Moreover, stateless firewalls block traffic in both ways while in real scenarios it is required to allow traffic initiated from the internal network and allow outside reply traffic while block traffic initiated from outside to internal resources. Stateful firewalls keep track of the connection state and therefore block packets that are not part of an established connection. Moreover, they allow for one-direction traffic. The architectures in [47, 48] use a stateful firewall. Firewalls at the control plane suffer potentially from controller overhead and scalability issues because of the amount of traffic needed to be forwarded to the controller.

2.2.3.3. Distributed stateful firewalls at the data plane switches
The architectures presented in [49–51] implement distributed stateful firewalls at the data plane switches. These approaches take a load off the controller by putting the stateful firewall logic in the data plane switches. However, these approaches are costly in the setup and maintenance phases. In [50], we find a firewall for SDN called Stateful Distributed FireWall (SDFW). It has one or more modules at every SDN plane: A user interface at the application plane for the user to specify high-level security policies, modules for translating user policies to *OpenFlow* rules, and other management modules at the control plane. Besides the data plane switches having the stateful capability. Open vSwitch conntrack module [18] is used to enable data plane state tracking.

In [49], an architectural design for a stateful firewall for SDN coined FORTRESS is presented. Two different architectural designs are presented: Stand-alone and cooperative. In the stand-alone design, no packets are exchanged between data plane switches and the controller. In the cooperative design, traffic can take different paths and therefore the connection state needs to be synchronized between the different data plane switches. The implementation of this system is done using *OpenState* [12] where finite state machines are used to inspect incoming packets at each data plane switch. They used a Mealy machine for every protocol. Their implementation is limited to only the TCP protocol.

2.3. SDN/NFV-Based Security Systems

NFV is proposed as an approach to decouple network functions (e.g., firewall, load balancer, NAT, or web proxy) from its dedicated hardware. Then, the network functions are implemented as software instances running on powerful servers. It is claimed that NFV together with cloud platforms enable building flexible virtual networks [52]. It deals with changes of Virtual Networks (VNs) by ensuring that the virtual firewalls be properly amended to ensure the protection of the resources. Based on heuristics, some mechanisms were designed to dynamically adapt virtual firewalls to VNs changes [52]. However, these mechanisms cannot be easily fully automated. The RNS algorithm could play an important role in the process of dynamically adapting virtual firewalls to VNs changes. In this sense, the RNS algorithm could be a part of an NFV enabling the virtualization of firewalls. In [53], we find the presentation of three architectures implementing stateful firewalling with NFV and SDN: Controller-Centric Approach, Virtual Network Function (VNF)-Centric Approach, and Hybrid SDN/NFV Approach. The Controller-Centric Approach exploits the capabilities of SDN switches for implementing the data plane firewall functionality. In this case, the *firewall app* as given in the architecture presented in [53] could play the role of DCG plane shown in Figure 1. However, the DCG plane is specialized in network segmentation and automatic generation of bulletproof (access-wise) policies for the firewalls. In the VNF-Centric Approach and the Hybrid SDN/NFV, the VNFs approach relies on virtualized firewalls that are deployed in a cloud environment. In this case too, the DCG plane can be a part of the *vFirewall* component in the architectures (VNF-Centric and Hybrid SDN/NFV) presented in [53].

3. Robust Network and Segmentation (RNS) Algorithm

To ensure a network segmentation that guarantees the best structure and configuration of the network, the RNS algorithm, given in Algorithm 1, has been proposed in [8]. It is a procedure that takes a set of resources and their individual access control policies, then it derives the topology of the network that connects them to one entry point allowing authorised access to each of them. It calculates the policies to be executed at the firewalls on the paths to the resources from the entry point such that we have defence in depth [54, 55] and an access control protection that is robust. The topology obtained by the algorithm and the segmentation it entails ensure the robustness of the network. The algorithm,

that we only briefly present in this paper, uses Product Family Algebra (PFA) [56–58] theory to calculate the topology and the firewall policies. In determining the topology and the policies for the firewalls, the algorithm works out a segmentation of the resources such that resources of similar access control policies are grouped under the same firewall. The placement of firewalls is determined by the notion of refinement on families of policies. For further details, we refer the reader to [54]. The commonalities between policies are calculated using their Greatest Common Divisor (GCD) as presented in PFA. Firewalls have to enforce common policies. For instance, if all the policies associated with the resources under a firewall deny access for any traffic coming from a range of IPs, then the firewall will deny all the traffic coming from this considered range of IPs. If we take a resource from a segment and arbitrary move it to another segment, then the weight associated with the access control policy of the firewall protecting its initial segment will be higher than the weight of the segment it has been moved to. In ranking policies of the firewalls, weights are given to the different security requirements. For instance, if traffic from an internal resource is deemed less dangerous than that from an external one, the weight will take into account this requirement. For further discussion on the role played by the weight function in assigning a resource to a segment, we refer the reader to [8].

The RNS algorithm is a polynomial algorithm and it is efficient in calculating a network topology and the policies of its firewalls. Hence, it could be easily used to reconfigure dynamic networks. For instance, at each change of the availability or the policies of the network resources, the algorithm is executed to recalculate within seconds the topology and the new firewall policies. Then, the network is changed to have the calculated topology and to enforce the new calculated firewall policies. The RNS algorithm is essential for the DCG plane, which is responsible for amending the network configuration and the network access control policies. It dynamically determines changes to the network topology and the policies of its firewalls to ensure proper almost real-time secure governance. The mathematical foundations of these procedures are given in [8].

The originality of the RNS algorithm resides in the fact that it is based on rigorous algebraic calculations. Although many approaches and formalisms exist to build a secure network and harden security in existing networks, they do not use a formal approach to achieve network segmentation. Their approaches lack the ability to automate the derivation of the best solution for network segmentation and prove its correctness. The RNS algorithm deals well with the dynamic nature of networks. At each change of the network topology or policies, the algorithm calculates the amendments to the topology, distribution of the firewalls, and the policies that the latter need to enforce.

The output of the algorithm (i.e., topology and firewall policies) achieves the principle of layered defence. Moreover, the firewall policies on every path from the entry point to resources get stricter by reducing the attack surface as we go deep into the network. The focus of internal firewalls is then on monitoring and controlling internal traffic and lateral movement, which enhance the overall network performance.

3.1. RNS implementation

For this study, the RNS algorithm has been implemented as a module located in the DCG plane. In the context of SDN,

Algorithm 1 Robust Network and Segmentation (RNS) Algorithm [8]

procedure Segmentation(R) ▷ $R = \{r_1, r_2, \ldots, r_n\}$
  $G \leftarrow NULL$ ▷ $G = (V, E, r)$
  $r \leftarrow$ Create-node(R) ▷ Create root r
  Add-node-to-G($G, r, \emptyset, false$) ▷ Add root r to G
  $S_1, S_2, \ldots, S_m \subset R$ such that $\bigcup_{i=1}^{m} S_i = R$, and no two subsets have resources with same polices
  $F = \emptyset$
  for each $s \in \{S_1, S_2, \ldots, S_m\}$ **do**
    $F = F \cup$ Create-node(s)
  end for
  while $F \neq \emptyset$ **do**
    $w_{max} \leftarrow$ maximum weight of any $s \in F$.
    $T \leftarrow \emptyset$
    for each $s \in F$ **do**
      if $s.weight = w_{max}$ **then**
        $T \leftarrow T \cup s; F \leftarrow F - s$
      end if
    end for
    Add-Nodeset-to-G(G, F, T, w_{max})
  end while
end procedure

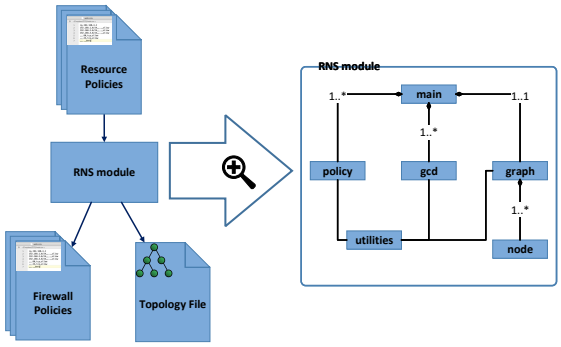

Fig. 3: Structure of the RNS module and its input and output

the RNS module generates the data plane topology and structure. It determines how many switches are needed, where they should be placed, and the links between resources and switches. Moreover, it generates policies to be enforced at each switch. In the case of a policy or topology change, the DCG plane gets notified and re-executes the RNS module dynamically to generate the updated topology and firewall policies. The updates are then carried to the data plane topology and the firewall policies are updated.

The RNS module at the DCG plane consists of six classes as shown in Figure 3. A main class, a class for storing and manipulating policies, a class for storing and calculating GCDs of the families of policies, and a class for storing and generating network graph that uses a node class. Moreover, it has a utility class that encompasses useful methods used by the classes of the RNS module. In the assessment work that we carried for this work, we provide to the RNS module a list of resource names along with their policy files. In the policy file, the first line indicates the IP address of the resource. The rest gives the policy by sequentially stating the rules. The grammar of the rules is given in Figure 4.

⟨rule⟩ ⟶ [⟨source_ip⟩], [⟨source_port⟩], [⟨destination_ip⟩],
[⟨destination_port⟩], [⟨protocol⟩], ⟨action⟩
⟨source_ip⟩ ⟶ ⟨ip_number⟩ | ⟨sub_network⟩
⟨source_port⟩ ⟶ ⟨port_number⟩ | ⟨port_range⟩
⟨destination_ip⟩ ⟶ ⟨ip_number⟩ | ⟨sub_network⟩
⟨destination_port⟩ ⟶ ⟨port_number⟩ | ⟨port_range⟩
⟨protocol⟩ ⟶ TCP | UDP | ICMP | all
⟨action⟩ ⟶ $allow$ | $deny$
⟨sub_network⟩ ⟶ ⟨ip_number⟩/⟨dgits⟩
⟨ip_number⟩ ⟶ ⟨8bit_digit⟩.⟨8bit_digit⟩.⟨8bit_digit⟩.⟨8bit_digit⟩
⟨8bit_digit⟩ ⟶ $0 - 255$
⟨port_range⟩ ⟶ ⟨port_number⟩ : ⟨port_number⟩
⟨port_number⟩ ⟶ ⟨digits⟩
⟨digits⟩ ⟶ ⟨digit⟩ | ⟨digit⟩⟨digits⟩
⟨digit⟩ ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Fig. 4: Firewall Rule Grammar

The non-terminals ⟨source_ip⟩ and ⟨destination_ip⟩ are optional indicating the source IP and destination IP, respectively. They can be a specific resource IP address, a network, or a sub-network. When these non-terminals are not given, it indicates the IP that covers all IP domain (i.e., $0.0.0.0/0$). The non-terminals ⟨source_port⟩ and ⟨destination_port⟩ are optional indicating the source and destination ports, respectively. They can be a specific port or a range of ports. When not provided, it indicates the whole range of possible ports (i.e., 0 to 65535). The non-terminal ⟨protocol⟩ is optional and indicates the communication protocol. The non-terminal can be a specific protocol (i.e., TCP, UDP, or ICMP) or *"all"* which indicates all protocols. When no value is given for this field, it indicates the *"all"* value. The non-terminal ⟨action⟩ is the only mandatory one which indicates the action to be taken by the switch. The possible values in our simulation are *allow* or *deny* which instruct the switch to forward the packet to its destination or drop the packet, respectively.

The RNS module at the DCG plane creates a policy object for each resource. The policy object reads a policy file and stores the policy. It transforms the sequential rules into disjoint rules such that executing them in any order produces a consistent policy. It does the transformation as it reads rules one by one from the policy file. If a new rule has a domain that intersects with that of an already existing rule, the intersected part is removed from the new rule as it already exists in another rule. The policy object uses a weight function to calculate and stores the weight of the policy.

The RNS module proceeds to implements the RNS algorithm. In the case of a policy or topology change, the DCG plane gets notified and re-executes the RNS module dynamically to generate the updated topology and firewall policies. The updates are then carried to the data plane topology and firewall policies are updated.

4. Implementation of the Architectures

In this section, we present the implementation of the three architectures introduced in Section 1. They implement stateful firewalls, which keep track of connections state. Therefore, each firewall needs to follow the progress of a session by recording its state attributes and values. The firewall changes the state of the connection upon receiving a packet. The architectures presented in this paper record sessions' attribute values using state tables. In all architectures, each firewall has its own state table. Therefore, Architecture 1 has a single state table as shown in Figure 5, Architecture 2 has a state table for each firewall as shown in Figures 6. Architecture 3 has a state table at each firewall for each protocol as shown in Figure 7.

In all the architectures that we are considering, the topology that is generated by the DCG plane is used to create the data plane architecture as shown in the Figures 5, 6 and 7. It is the same topology that we will be using in the assessment section. Also, the firewall policies in all architectures under consideration are generated by the DCG plane. In the setup phase, each firewall fetches its policy. In Architecture 1, the single firewall reads its policy, processes it, and stores it in a policy holder as shown in Figure 5. The same applies to each firewall in Architecture 2 in Figure 6. However, in Architecture 3, as each switch registers with the controller, its policy is read and processed at the control plane then pushed down to the flow tables of the switch at the data plane as shown in Figure 7. In the operation phase, switches in Architectures 1 and 2 forward the first packets of every communication to the control plane for checking policy and connection state. Once a connection is established or denied by the policy, flow table entries are inserted in the switch to handle future packets. However, Architecture 3 checks policy and track state connection at the switch without the need for any communication with the control plane.

Before we go on to explain the details of the implementation of each architecture, it is important to explain how the ICMP, UDP, and TCP communications are processed in a network.

ICMP is a protocol that provides error and information messages for IP-based network. *ping* application uses ICMP messages to test connectivity. It sends an echo request to a host and waits for the echo reply, if no reply is received within a certain period of time it times out and the remote host is declared unreachable.

UDP is a connectionless protocol which means there is no need to establish communication before sending data. *iPerf* sends a UDP stream from the sender host. In the end, the receiver host sends an acknowledgement to the sender host.

TCP is a reliable connection-oriented protocol. TCP needs to establish the communication before sending data. The connection is established using the three-way handshake in which three packets are exchanged between the communicating hosts. The first packet in the handshake, sent by the first host, is identified by setting the SYN bit. The second host replies with a second packet in which the SYN and ACK bits are set to indicate the acknowledgement of receiving the first packet and continuing the handshake. To which, the first host sends a third packet that has the ACK bit set to inform the other host of the establishment of the connection.

4.1. Architecture 1 Implementation

The first architecture is a single centralised firewall as shown in Figure 5. In this architecture, the firewall application creates a single firewall object responsible for enforcing the policies of all the switches in the data plane.

To implement this architecture, we have created a firewall application attached to the controller. It has a policy holder, a state table, a package that has protocol handlers, a handler for each protocol, and a module that implements the switch functionality or to communicate *OpenFlow* commands to the switch. In this architecture, the state is handled by the firewall, it has a single state table to keep track of all the connections for all the switches in the network. The same applies to the policy, in this architecture we

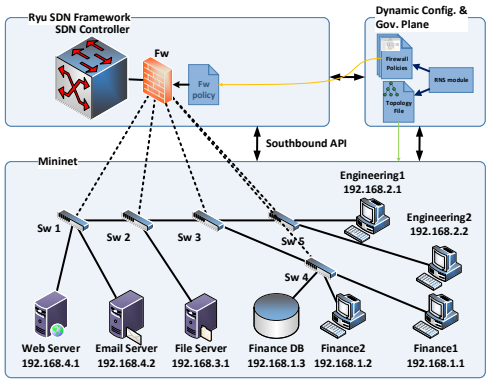

Fig. 5: Architecture 1

have a single policy that gets enforced. Hence, the question is then how to obtain the policy of this unique firewall from the several policies to be deployed at the internal control points calculated by the RNS algorithm? The idea is to add to the state-space of the policies as an attribute that indicates on what switch a rule is enforced. Hence, generating this global policy is straightforward from the policies of the internal access control points calculated by RNS algorithm.

Once this application is initiated and the firewall object is created at the control plane, it reads and stores its policy that governs the decisions of all the switches. The policy generated by the DCG plane for this architecture is one single policy. When the topology is created on *mininet*, each switch registers with the controller and the firewall instructs it to add a single rule with low priority to forward packets to the controller. When a switch receives a packet, it matches the added rule and forwards the packet to the firewall. The firewall inspects the packet header and assigns it to the appropriate handler. For example, a first ICMP packet in communication is checked against the state table. If the state table has no entry for it, the policy is checked. If the policy denies the packet, the firewall instructs the switch to drop the packet and add an entry to its flow table to drop similar packets for a certain period of time. However, if the packet is allowed by the policy, the firewall adds an entry to the state table to handle the reply packet. It also instructs the switch to forward the packet and adds an entry to its flow table to forward similar packets for a certain period of time. When the reply packet arrives at the switch, the switch forwards the packet to the firewall. The firewall finds an entry in the state table, then updates the state table setting the connection state to established and instructs the switch to forward the packet and add an entry to its flow table. A UDP connection is handled in a similar way. The firewall handles the TCP protocol handshake in a similar way except the first packet has a SYN flag, the second packet has a SYN-ACK flag, and the third packet has an ACK flag. The firewall prevents DDoS attack by keeping track of request packets that have no reply, if it passes a certain threshold, the firewall instructs the switch to add an entry to drop such packets and avoid overhead.

In the data plane switches, each time the firewall add an entry to the flow table, it set an expiry time for its usage. Once an entry reaches its expiry time, it is removed from the flow table and the firewall gets a notification by an *OpenFlow* message. Once it receives this notification message, the firewall application removes the state table entries.

4.2. Architecture 2 Implementation

Architecture 2 consists of multiple firewalls each one is responsible for managing a single switch as shown in Figure 6.

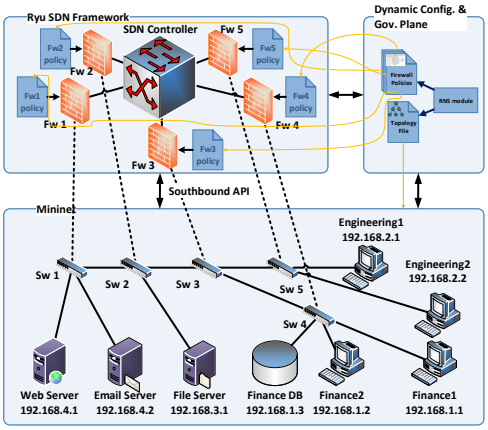

Fig. 6: Architecture 2

To implement this architecture, we have created a firewall application at the controller. It creates a firewall object for each switch. Each firewall object has a policy holder, a state table, a package that has a protocol handler for each protocol, and a module that implements the switch functionality or communicates *OpenFlow* commands to a switch. In this architecture, the state is handled by each firewall separately as it has a state table to keep track of connections for the assigned switch. Compared to Architecture 1, this architecture presents a better design as it applies the principle of separation of concerns: what concerns a switch is delegated to a firewall object.

In the setup phase, Once a switch is created at the data plane and sends its feature to the controller, the firewall application creates a firewall object designated for this switch. A firewall object reads the policy that governs the decisions of its switch and stores it. When the switch receives a packet that is not matched by any entry table in its flow tables, it forwards it to its corresponding firewall. The firewall checks its state table and, if needed, the policy, then it instructs the switch on how to handle the packet. When a communication is established, the firewall instructs the switch to add an entry to its entry table to handle similar future packets.

4.3. Architecture 3 Implementation

Architecture 3 is a distributed firewall architecture at the data plane. In this architecture, we transform data plane switches into stateful firewalls using *BEBA software switch* as shown in Figure 7. Such that switches handle firewall rules and keep track of the connection state without forwarding traffic to the controller.

In this architecture, we use a state machine to handle each protocol. This is why each protocol is handled by a separate flow table as we explain below. A state machine for a protocol forwards legitimate packets or drops packets according to their state transitions, and changes state if needed. If a packet is part of an established connection or a connection to be established, then it is forwarded by the firewall. Otherwise, it is dropped.

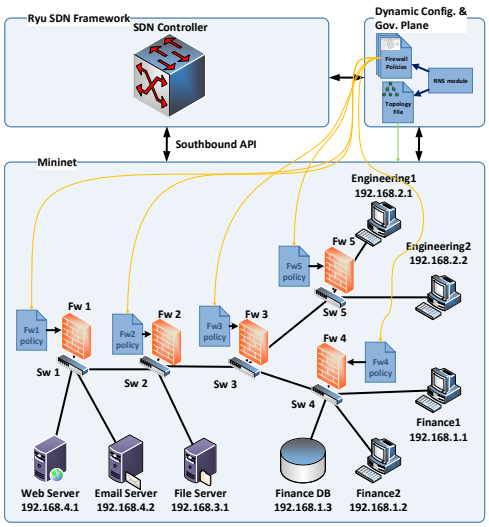

Fig. 7: Architecture 3

In Architecture 3, each protocol is handled by a separate flow table which has a state table. Therefore, we have a state table for each of TCP, UDP, and ICMP protocols. Policies might involve rules that are specific to a protocol. Therefore, the policy at each switch can be split into (sub-)policies associated with the flow tables of the protocols. Each flow table enforces the rules that are relevant to its protocol.

To implement Architecture 3, we have created an application attached to the controller. The application consists of a main class, and five more classes each is assigned to handle a flow table. The first class inserts rules in the first flow table, `T0`, to route packets to the appropriate flow table. The second class inserts flow entries in the flow table, `T1`, to handle ICMP packets. The third class inserts TCP packets rules in the flow table `T2`. The fourth class inserts rules in the flow table `T3` to handle UDP packets. The last class inserts entries in the flow table `T4`, which is intended to keep track of what port is assigned to communication on each mac address.

In Architecture 3, the firewall application starts with the controller. When a switch sends a feature reply message to the controller (i.e., registers at the controller), the firewall object reads the policy associated with the switch and stores it. It then starts a first table object which inserts six rules in the first flow table, the first two rules are to drop every LLDP and IPV6 packets. The third, fourth, and fifth rules forward ICMP, TCP, and UDP to tables `T1`, `T2`, and `T3`, respectively. The sixth rule forwards packets not matched by the above rules to `T4`.

Afterwards, the firewall starts the second object, which is to prepare the flow table `T1` to handle ICMP packets. The first *OpenFlow* command sets `T1` into a stateful table. Then it sets the lookup-scope attributes, which are the attributes that the switch extracts from the packet and match with the state table entries as discussed in Section 2.1. The lookup-scope in the ICMP case is the tuple (`source_ip`, `destination_ip`). The next attributes to be set at the switch are the update-scope attributes, which are the attributes used by the switch to update the state table. In ICMP, the update-scope is the tuple (`destination_ip`, `source_ip`). Finally, it is the step of inserting entries in the XFSM flow table. The first entry is to handle an established connection. It checks if the state found in the state table is 1 (i.e., established) then the switch forwards the packet to `T4` to be forwarded to the right port, and update the state for the other direction to 1 (i.e., established). This rule is given the highest priority. Then the object checks the policy rules and inserts flow entries for the rules that are related to ICMP or “all" packets. These rules are used to match the first packets in a communication (i.e., state 0). An allow rule passes the packet to `T4` to be forwarded, and updates the state for the other direction to 1, otherwise, it drops the packet. An *allow* rule is given a medium priority and a *deny* rule is given a low priority. A first packet is processed by the state table which does not match any entry and is given the default state 0. The XFSM table processes the packet to either forward it and update the state table, or to drop the packet.

The third and fourth objects have similar functionality to handle TCP and UDP packets. The fifth object is intended to update the `T4` entries, which are for keeping track of the ports associated with every mac address.

5. Assessment of The Architectures

5.1. Testbed environment

To assess the three architectures presented in Section 1., we use the following components:

- *mininet* 2.2.2 [10]: It is a tool used to emulate and prototype SDNs, running on Ubuntu 14.04.4 (64 bit) virtual machine on VirtualBox 6.0.
- BEBA controller: It is based on Ryu OpenFlow Controller 3.29.
- *BEBA software switch* supporting *OpenFlow* 1.3.

For generating flows and collecting measurement data we used *ping*, *Wireshark* 1.10.6, and *iPerf* 2.0.5 utilities. The testbed environment is setup on a MacBook Pro with a CPU 2.7 GHz Intel Core i5 and a memory of 8 GB 1867 MHz DDR3.

5.2. Selection of the topology to be used in the testbed

The topology used in the testbed is the topology generated by the RNS module in the DCG plane for a network composed of two engineering workstations (*Engineering1* and *Engineering2*), two finance workstations and a database (*Finance1*, *Finance2*, and *Finance DB* respectively), a *File server*, a *Web server*, and an *Email server*. The topology can be seen in the Figures 5, 6, and 7.

The high-level policies of these resources are the following: *Engineering1* and *Engineering2* belong to the engineering department and limit access to resources belonging to the respective department. *Finance1*, *Finance2*, and *Finance DB* have a similar policy applied to the finance department. The *File server* allows access to resources belonging to internal departments (i.e., engineering and finance) only. The *Web server* and *Email server* allow access to internal resources and allow limited access to outer resources (i.e., HTTP and SMTP protocols).

The above requirements are translated into low level policies and then fed to the RNS module in the DCG plane. The DCG plane generates the topology as shown in Figures 5, 6, and 7. It also generates firewall policies.

To test the dynamic nature of the network, we added two admin resources to the set of resources. The policy of these resources is to allow access to traffic generated from the admin resources

only. These resources are to gain access to every resource in the network and this is achieved by updating the policies of all network resources. The DCG plane recalculates the topology and generates updated firewall policies.

6. Results and discussion

In this section, we present and discuss the results of the assessment of the three architectures. The tests are done on the SDN topology generate by the DCG plane for the network resources mentioned above. We have tested the three architectures for the topology shown in Figures 5, 6, and 7.

One of the criteria to compare the architectures is the cost for the setup of the network. The cost is measured by the number of packets exchanged between the control plane and the data plane in the setup phase. For Architectures 1 and 2, 46 packets are exchanged to complete the setup. Architecture 3 took on average 1508 packets. We performed 10 tests on Architecture 3 and the minimum number of packets exchanged is 1459 and the maximum number is 1742. This is due to the fact that in Architectures 1 and 2 the only messages exchanged are the setup messages, while in Architecture 3 besides setup messages, the messages for setting stateful tables, inserting flow table entries are also exchanged at this stage.

6.1. Reachability

After the setup of the environment, the first test we perform on all architectures is whether the policies are enforced as expected for the ICMP protocol, and for that, we did a reachability test between all the resources in the network. Architectures 1, 2, and 3 all have the same result as shown in Figure 8. The reachability test is done by the command *pingall*[1], where every host tries to ping every other host in the data plane. In the first line in the result, we see a label at each line (i.e., `eng1`) which is the name of the resource initiating the ping request. After the arrow `->` a resource name (i.e., `eng2`) indicates a successful communication with that resource (i.e., reply), and an `x` indicates a failed communication (i.e., blocked). For example, `eng1` is able to access `eng2`, `file`, `web`, and `email` while failing to access `fin1`, `fin2`, and `finDB`.

```
mininet> pingall
*** Ping: testing ping reachability
eng1 -> eng2 X X X file web email
eng2 -> eng1 X X X file web email
fin1 -> X X fin2 finDB file web email
fin2 -> X X fin1 finDB file web email
finDB -> X X fin1 fin2 file web email
file -> X X X X X web email
web -> X X X X X X email
email -> X X X X X X web
*** Results: 51% dropped (27/56 received)
mininet>
```

Fig. 8: Reachability test

[1] The *pingall* command tests the reachability for the ICMP protocol only. For the other protocols we use the *iPerf* tool as shown in bandwidth and latency variation tests.

6.2. Response Time

The response time or latency test measures the time it takes to get a reply for a request. We carried this test by sending several ICMP packets using *ping*. We measure the response time for the traffic originating from *Engineering1* to *Web server*.

In Figure 9, we present the results for only 10 ICMP packets as the results are the same for a number above 10 packets. Figure 9 shows that the first packet took about the same time for Architectures 1 and 2 and less time for Architecture 3. The reason is that switches in Architectures 1 and 2 do not have entries in their flow tables to handle the packet, and the packet is forwarded to the controller. On the other hand, switches in Architecture 3 have entries in their flow tables to handle the packet, which reduces the time to handle the first packet. Moreover, Architecture 1 and Architecture 2 needed to exchange 32 packets between data plane switches and the controller while Architecture 3 does not need to exchange any packets. Architectures 1, 2, and 3 show similar response times for the subsequent packets. The reason is that the subsequent packets switches already have entries in all architectures to handle the packets that follow the first packet.

An important factor impacting the response time is the number of hosts trying to communicate in a short time interval and several sessions are being established. For this purpose, we performed the same tests as shown in Figure 9 measuring the response time for the traffic originating from *Engineering1* to *Web server* while there are different communications going in the network. Figure 10 presents the results for 10 ICMP packets. It shows that the first packet takes almost the same time for Architectures 1 and 2 and less time for Architecture 3 for the same reason presented above. Architectures 1, 2, and 3 show similar response times that is due to the fact that subsequent packets switches already have entries in all architectures to handle subsequent packets to the first packet of a new connection.

To see the difference in response time between the three architectures more clearly, we have created three different topologies with the *Engineering1* on one end and *Web server* on the other end. The three topologies consist of $1, 10$, and 20 switches between these resources. These switches enforce the same policy. In all tests, *Engineering1* sends a single ICMP packet (i.e., ping request) and we measured the response time. Figure 11 shows the average results for 10 tests. The average times for the case of a single switch for Architectures 1, 2, and 3 are 30.31 ms, 29.5 ms, and 1.18 ms, respectively. The average times for the case of 10 switches for Architectures 1, 2, and 3 are 1312.2 ms, 1300.3 ms, and 607.5 ms, respectively. Finally, the average times for the case of 20 switches for Architecture 1, 2, and 3 are 4963.3 ms, 5003 ms, and 2116.5 ms, respectively.

We performed the above test while there is ongoing communication in the network. Figure 12 shows the average results for 10 tests. The average times for the case of a single switch for Architectures 1, 2, and 3 are 46.530 ms, 42.140 ms, and 0.176 ms, respectively. The average times for the case of 10 switches for Architectures 1, 2, and 3 are 2691.6 ms, 2676.3 ms, and 1409.0 ms, respectively. Finally, the average times for the case of 20 switches for Architecture 1, 2, and 3 are 8123.1 ms, 8278.6 ms, and 2924.7 ms, respectively. It is clear from Figure 11 and Figure 12 that Architectures 1, and 2 has comparable response time and Architecture 3 has significantly better time-performance.

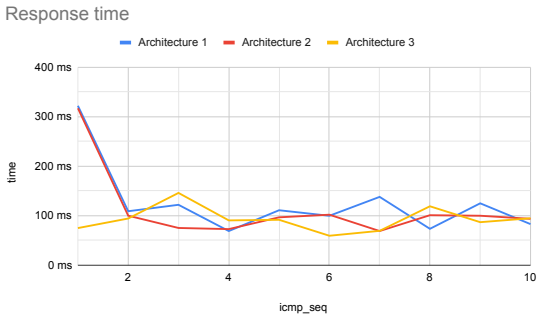

Fig. 9: Response time for 10 ICMP packets

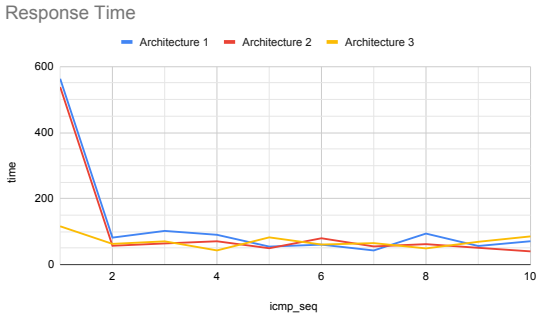

Fig. 10: Response time for 10 ICMP packets with several sessions

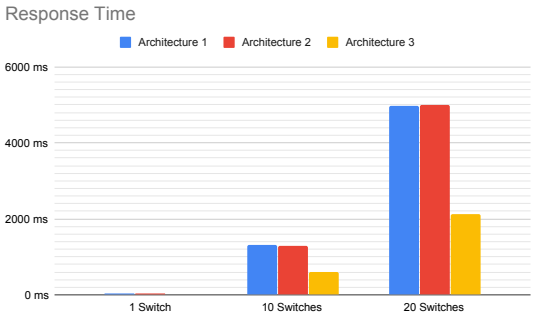

Fig. 11: Response time for topologies of 1, 10, and 20 switches

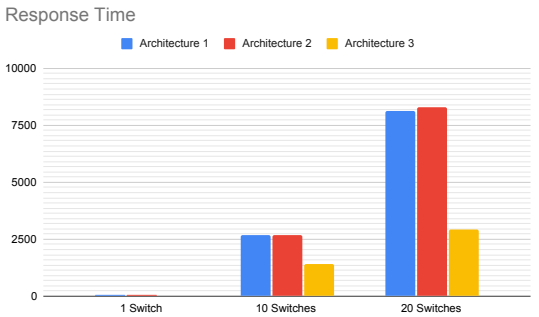

Fig. 12: Response time for topologies of 1, 10, and 20 switches with several sessions

6.3. Bandwidth

One of the tests we performed to compare the three architectures is network bandwidth. Network bandwidth is the maximum rate or volume of data that can be transferred on a link per unit of time. The bandwidth test is done by performing a TCP communication using *iPerf* between the selected hosts (i.e., `eng1` and `web`). The result is shown in Table 1 and broken down into 0.5 second intervals in Figure 13. Note that bandwidth test between virtual hosts depends on the host machine CPU speed. The bandwidth for the virtual host in *mininet* varies because of the host machine running processes and CPU load at a given time [59]. Therefore, the three architectures will not vary in bandwidth as they use the same environment with the same switches and links.

	Interval	Transfer	Bandwidth
Architecture 1	`0.0-20.4 sec`	`512 KBytes`	`206 Kbits/sec`
Architecture 2	`0.0-19.0 sec`	`512 KBytes`	`221 Kbits/sec`
Architecture 3	`0.0-20.4 sec`	`512 KBytes`	`205 Kbits/sec`

Table 1. Network bandwidth for Architectures 1,2, and 3

For a TCP communication, Architecture 1 and Architecture 2 exchange 154 packets between the data plane and control plane while Architecture 3 does not exchange any packet.

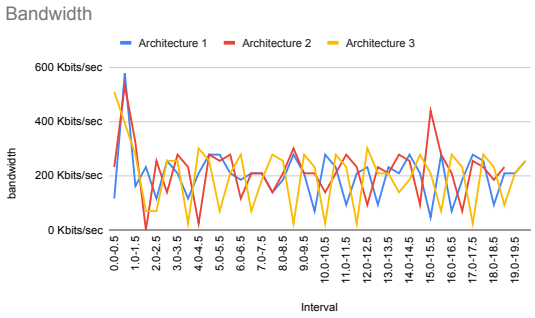

Fig. 13: Bandwidth

6.4. Latency Variation

Latency variation or jitter is the variance in delay time or latency between packets' arrival. The jitter test is done by performing a UDP communication using *iPerf* between the selected hosts (i.e., `eng1` and `web`). The result is shown in Table 2 which is broken down into 0.5 second intervals in Figure 14. We notice that the three architectures have very similar jitter and that it is not affected by the difference in architectures for the same reasons discussed for the bandwidth.

Architecture 1 and Architecture 2 exchange 45 packets in UDP connection while 0 packets are exchanged in Architecture 3.

	Interval	Transfer	Bandwidth	Jitter
Architecture 1	`0.0-8.8 sec`	`1.09 MBytes`	`1.05 Mbits/sec`	`17.349 ms`
Architecture 2	`0.0-9.1 sec`	`1.14 MBytes`	`1.05 Mbits/sec`	`16.792 ms`
Architecture 3	`0.0-9.9 sec`	`1.25 MBytes`	`1.06 Mbits/sec`	`15.962 ms`

Table 2. Network jitter for Architectures 1, 2, and 3

Looking at the tests done on the operation phase which are the latency, bandwidth, and jitter, we can notice the following. First, results are similar for bandwidth and jitter tests for the three architectures as they are based on the environment consisting of topology, switches, and links which are the same for our

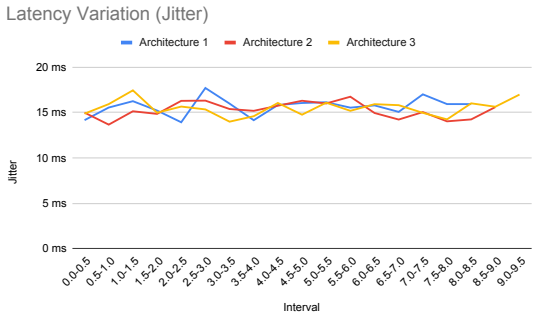

Fig. 14: Latency variation (jitter)

architectures. Second, the result for the response time is high in Architecture 1 and Architecture 2 and low in Architecture 3. The reason for this is the time architectures 1 and 2 take to add entries in the flow table of the switches in response to the first packet. Finally, we also notice that for all these tests, Architecture 1 and Architecture 2 exchange packets between the data plane and the control plane while Architecture 3 does not exchange any packet.

6.5. Resilience to Topology Change

In network topology, many changes to the topology can occur. These changes can be intentional and planned for by network administrators such as adding resources, removing resources, or changing resource policies. Moreover, in a dynamic environment, resources can be in and out of the network at will. Hence for efficient security, the governance of the network access needs to shadow the changes. In any of these cases of change, the RNS module in the DCG plane is re-executed on the fly to generate an updated topology and firewall policies. Then, the data plane topology running on *mininet* needs to be updated at run-time. For this purpose, the updated topology is compared to the old topology and a topology update script is created. The script contains *mininet* commands to delete or add hosts, switches, or links. We run the script on *mininet* CLI. The controller gets notified of such topology changes, and it updates the firewall policies that are provided by the DCG plane.

To assess the effect of a change, we added two administration workstations. These workstations can access all resources and they allow access between themselves and deny everything else. We use this case to re-execute the RNS module and generate new firewall policies and an updated topology. The *mininet* topology is updated on the fly as explained above.

To compare the response for the update by the three architectures, we evaluate the update cost of each architecture by counting the number of exchanged packets between the control and data plane to fulfil the update. In our example, Architectures 1 and 2 exchanged 54 packets between control and data planes. Architecture 3 exchanged 1670 packets on average. The difference is huge between Architectures 1 and 2 on one side and Architecture 3 on the other side.

We also did the reachability test of all the architectures after the topology update. The result for the architecture 1, 2, and 3 are shown in Figure 15.

The use of *mininet* does not allow to test the effect of topology changes perfectly with *BEBA software switch* as it is not supported fully by *mininet*. Instead, *mininet* fully supports *Open vSwitch*.

```
mininet> pingall
*** Ping: testing ping reachability
eng1 -> eng2 X X X file web email X X
eng2 -> eng1 X X X file web email X X
fin1 -> X X fin2 finDB file web email X X
fin2 -> X X fin1 finDB file web email X X
finDB -> X X fin1 fin2 file web email X X
file -> X X X X X web email X X
web -> X X X X X X email X X
email -> X X X X X X web X X
admin1 -> eng1 eng2 fin1 fin2 finDB file web email admin2
admin2 -> eng1 eng2 fin1 fin2 finDB file web email admin1
*** Results: 50% dropped (45/90 received)
mininet>
```

Fig. 15: Reachability test after topology update

One of the future suggestions is to implement the RNS algorithm using the new version of *Open vSwitch*.

7. Conclusion and Future Work

The RNS algorithm is a product family approach for policy-based network design. It uses the policies of a set of resources to segment them to achieve the best access control security. This paper has presented three different architectures for implementing the RNS algorithm into SDN. We assessed these architectures to compare them. We observe a difference in cost between setup and operation phases. Architectures 1 and Architecture 2 have a low setup cost but high operational cost, while it is the opposite for Architecture 3. Hence, there is a trade-off between the time to setup proper routing mechanisms and the time for normal communication operation. It becomes obvious that if an environment is high in traffic but more stable, then Architecture 3 is more suitable. However, if the topology is more dynamic compared to traffic volume, then Architectures 1 and 2 are more suitable.

Architecture 1 and Architecture 2 are similar in terms of where the firewalls reside, however they are different in terms of design. Architecture 2 is more suitable for modifiability than Architecture 1 as it is designed based on the separation of concerns principle and gives more flexibility in updating policies when we need to update the affected policies only rather than updating the single policy that governs the whole network traffic. Moreover, for a given packet arriving at the controller, we need to go through a limited number of rules rather than going through all rules for all switches in the network.

Many security threats or attacks can target an SDN environment affecting the control plane, application plane, southbound API, northbound API, or data plane [3] which applies to all architectures. Architecture 3 by being stateful at the data plane, by limiting the number of packets exchanged between the control and data plane, and by handling traffic locally at the switch, is more resilient to face vulnerabilities and SDN attacks targeting the controller, northbound API, and southbound API.

To improve the performance of the DCG plane, we can use the Defence in Depth (DD) strategy as an application at the control plane, for reactive and quick changes to only update policies without changing the structure of the whole topology. Moreover, topology changes are done periodically or at downtime as intended by the network administrator.

The DCG plane can be combined with other approaches to improve the security of an SDN environment. Intrusion Prevention System (IPS) for SDN such as WedgeTail [60] that inspects traffic going through switches to detect attacks. Such a solution can work

with our proposed architectures to secure SDN. First, we provide the best segmentation for resources that support IPS to have a better view of the traffic going through the network. Moreover, in the case of IPS detecting clandestine entry point or unusual network traffic, the use of DCG plane to recalculate policies and topology to isolated infected resources or segments preventing the spread of infection.

For future work, we plan on implementing the RNS algorithm to structure the control plane using distributed controllers such as ONOS [23], besides structuring the data plane. Another future work is to use *open vSwitch* as it is fully supported by *mininet* which allows for more flexibility to test the effect of topology changes. We also plan to carry further empirical studies on the performance of the RNS algorithm in various real networks. The aim is to assess its performance in relation to network sizes and various underlying network architectures.

Also, our future work aims at extending the role of the DCG to encompass other network management aspects such as preventing covert channels [61–63]. An analytic module that scrutinises the data that flows through the network could be also added to the DCG plane. It would use ontologies and organized data analysis techniques (e.g., [64, 65] and [66]) to develop an awareness about the security situation that might play a role in the configuration of the network.

Acknowledgments

This article is a revised and enlarged version of [16]. This study was funded by the Natural Sciences and Engineering Research Council of Canada—NSERC—(CA) (RGPIN-2020-06859).

References

[1] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[2] Ijaz Ahmad, Suneth Namal, Mika Ylianttila, and Andrei Gurtov. Security in software defined networks: A survey. *IEEE Communications Surveys & Tutorials*, 17(4):2317–2346, 2015.

[3] 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, 2020. URL https://doi.org/10.1007/978-3-030-22277-2_14.

[4] Mariusz Stawowski. The principles of network security design. *Information Systems Security Association (ISSA)*, 2007.

[5] Mariusz Stawowski. Network security architecture. *Information Systems Security Association (ISSA)*, 2009.

[6] Center for Internet Security (CIS). Critical security controls for effective cyber defense version 6.0. Technical report, CIS, 2015.

[7] Google Inc. Google's approach to it security. Technical report, Google, 2012.

[8] Neerja Mhaskar, Mohammed Alabbad, and Ridha Khedri. A formal approach to network segmentation. *Computers & Security*, pages 1–32, 2021. URL https://www.sciencedirect.com/science/article/pii/S0167404820304351.

[9] Vishva Nitin Patel, Devansh Shah, and Nishant Doshi. Emerging technologies and applications for smart cities. *Journal of Ubiquitous Systems and Pervasive Networks*, 15: 19–24, 2021.

[10] 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, June 2014.

[11] Davide Sanvito, Luca Pollini, Nicola Bonelli, Eder Leão Fernandes, and Carmelo Cascone. BEBA software switch. Available: http://www.beba-project.eu/ (Accessed: April 27, 2020), 2020.

[12] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. OpenState: Programming platform-independent stateful openflow applications inside the switch. *SIGCOMM Comput. Commun. Rev.*, 44(2):44–51, April 2014. ISSN 0146-4833.

[13] Gerald Combs. Wireshark. Available: https://www.wireshark.org/ (Accessed: May 29, 2020), 2020.

[14] Mike Muuss. The story of the PING program. Available: https://ftp.arl.army.mil/~mike/ping.html (Accessed: May 29, 2020), 1983.

[15] Jon Dugan, John Estabrook, Jim Ferbuson, Andrew Gallatin, Mark Gates, Kevin Gibbs, Stephen Hemminger, Nathan Jones, Feng Qin, Gerrit Renker, Ajay Tirumala, and Alex Warshavsky. iPerf. Available: https://iperf.fr/ (Accessed: May 29, 2020), 2020.

[16] Mohammed Alabbad and Ridha Khedri. Configuration and governance of dynamic secure SDN. In *The 12th International Conference on Ambient Systems, Networks and Technologies (ANT 2021)*, Procedia Computer Science series, pages 1–8, Warsaw, Poland, March 23 – 26 2021. Elsevier Science.

[17] Sibylle Schaller and Dave Hood. Software defined networking architecture standardization. *Computer Standards & Interfaces*, 54:197 – 202, 2017.

[18] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Keith Amidon Pravin Shelar, and Martín Casado. Open vSwitch. Available: https://www.openvswitch.org/ (Accessed: April 27, 2020).

[19] Enio Kaljic, Almir Maric, Pamela Njemcevic, and Mesud Hadzialic. A survey on data plane flexibility and programmability in software-defined networking. *IEEE Access*, 7:47804–47840, 2019.

[20] Rui Kubo, Tomonori Fujita, Yuji Agawa, and Hikaru Suzuki. Ryu SDN framework—open-source SDN platform

software. Available: https://osrg.github.io/ryu/index.html (Accessed: May 06, 2020), 08 2014.

[21] Ali Al-Shabibi and Murphy McCauley. POX Controller. Available: https://noxrepo.github.io/pox-doc/html (Accessed: Mar 23, 2020), 2020.

[22] Floddlight controller. Available: https://floodlight.atlassian.net (Accessed: May 06, 2020).

[23] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian P. O'Connor, Pavlin Radoslavov, William Snow, and Guru M. Parulkar. ONOS: Towards an open, distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[24] Liehuang Zhu, Md Monjurul Karim, Kashif Sharif, Fan Li, Xiaojiang Du, and Mohsen Guizani. Sdn controllers: Benchmarking & performance evaluation. *arXiv preprint arXiv:1902.04491*, 2019.

[25] Tooska Dargahi, Alberto Caponi, Moreno Ambrosin, Giuseppe Bianchi, and Mauro Conti. A survey on the security of stateful SDN data planes. *IEEE Communications Surveys & Tutorials*, 19(3):1701–1725, 2017.

[26] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. Flow-level state transition as a new switch primitive for SDN. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 61–66, New York, NY, USA, 2014. Association for Computing Machinery.

[27] Chen Sun, Jun Bi, Haoxian Chen, Hongxin Hu, Zhilong Zheng, Shuyong Zhu, and Chenghui Wu. SDPA: Toward a stateful data plane in software-defined networking. *IEEE/ACM Transactions on Networking*, 25(6):3294–3308, 2017.

[28] Pradip Kumar Sharma, Saurabh Singh, Young-Sik Jeong, and Jong Hyuk Park. DistBlockNet: A distributed blockchains-based secure SDN architecture for IoT networks. *IEEE Communications Magazine*, 55(9):78–85, 2017.

[29] Laizhong Cui, F. Richard Yu, and Qiao Yan. When big data meets software-defined networking: Sdn for big data and big data for sdn. *IEEE Network*, 30(1):58–65, 2016.

[30] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 7–12, New York, NY, USA, 2012. Association for Computing Machinery.

[31] Olli Mämmelä, Jouni Hiltunen, Jani Suomalainen, Kimmo Ahola, Petteri Mannersalo, and Janne Vehkaperä. Towards micro-segmentation in 5G network security. In *European Conference on Networks and Communications (EUCNC)*, 2016. Project code: 101719 ; European Conference on Networks and Communications : Workshop on Network Management, Quality of Service and Security for 5G Networks, 2016 EuCNC2016, EUCNC 2016 ; Conference date: 27-06-2016 Through 30-06-2016.

[32] Hadar Sufiev, Yoram Haddad, Leonid Barenboim, and José Soler. Dynamic SDN controller load balancing. *Future Internet*, 11:75, 03 2019.

[33] Tao Wang, Fangming Liu, Jian Guo, and Hong Xu. Dynamic SDN controller assignment in data center networks: Stable matching with transfers. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016.

[34] Lucas Soares da Silva, Carlos Renato Storck, and Fãtima de L. P. Duarte-Figueiredo. A dynamic load balancing algorithm for data plane traffic. In *LANOMS*, 2019.

[35] Rakesh Kumar and Rinkaj Goyal. On cloud security requirements, threats, vulnerabilities and countermeasures: A survey. *Computer Science Review*, 33:1 – 48, 2019. URL http://www.sciencedirect.com/science/article/pii/S1574013718302065.

[36] Paolo Bellavista, Alessandro Dolci, Carlo Giannelli, and Dmitrij David Padalino Montenero. SDN-based traffic management middleware for spontaneous WMNs. *Journal of Network and Systems Management*, 28(4):1575–1609, 2020.

[37] Ian Ku, You Lu, and Mario Gerla. Software-defined mobile cloud: Architecture, services and use cases. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 1–6, 2014.

[38] Venkatraman Balasubramanian and Ahmed Karmouch. Managing the mobile ad-hoc cloud ecosystem using software defined networking principles. In *2017 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6, 2017.

[39] Hans C. Yu, Giorgio Quer, and Ramesh R. Rao. Wireless SDN mobile ad hoc network: From theory to practice. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–7, 2017.

[40] Tariq Javid, Tehseen Riaz, and Asad Rasheed. A layer2 firewall for software defined network. In *2014 Conference on Information Assurance and Cyber Security (CIACS)*, pages 39–42. IEEE, 2014.

[41] Avinash Kumar and NK Srinath. Implementing a firewall functionality for mesh networks using SDN controller. In *2016 International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*, pages 168–173. IEEE, 2016.

[42] Michelle Suh, Sae Hyong Park, Byungjoon Lee, and Sunhee Yang. Building firewall over the software-defined network controller. In *16th International Conference on Advanced Communication Technology*, pages 744–748. IEEE, 2014.

[43] Justin Gregory V Pena and William Emmanuel Yu. Development of a distributed firewall using software defined networking technology. In *2014 4th IEEE International Conference on Information Science and Technology*, pages 449–452. IEEE, 2014.

[44] Karamjeet Kaur, Krishan Kumar, Japinder Singh, and Navtej Singh Ghumman. Programmable firewall using software defined networking. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 2125–2129. IEEE, 2015.

[45] Sergey Morzhov, Igor Alekseev, and Mikhail Nikitinskiy. Firewall application for floodlight SDN controller. In *2016 International Siberian Conference on Control and Communications (SIBCON)*, pages 1–5. IEEE, 2016.

[46] Thuy Vinh Tran and Heejune Ahn. A network topology-aware selectively distributed firewall control in SDN. In *2015 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 89–94. IEEE, 2015.

[47] Vipin Gupta, Sukhveer Kaur, and Karamjeet Kaur. Implementation of stateful firewall using POX controller. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1093–1096. IEEE, 2016.

[48] Thuy Vinh Tran and Heejune Ahn. FlowTracker: A SDN stateful firewall solution with adaptive connection tracking and minimized controller processing. In *2016 International Conference on Software Networking (ICSN)*, pages 1–5. IEEE, 2016.

[49] Maurantonio Caprolu, Simone Raponi, and Roberto Di Pietro. FORTRESS: An efficient and distributed firewall for stateful data plane SDN. *Security and Communication Networks*, 2019, 2019.

[50] Ankur Chowdhary, Dijiang Huang, Adel Alshamrani, Abdulhakim Sabur, Myong Kang, Anya Kim, and Alexander Velazquez. SDFW: SDN-based stateful distributed firewall. *arXiv preprint arXiv:1811.00634*, 2018.

[51] Ali Zeineddine and Wassim El-Hajj. Stateful distributed firewall as a service in SDN. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 212–216, 2018.

[52] Juan Deng, Hongxin Hu, Hongda Li, Zhizhong Pan, Kuang-Ching Wang, Gail-Joon Ahn, Jun Bi, and Younghee Park. VN-guard: An NFV/SDN combination framework for provisioning and managing virtual firewalls. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network*, pages 107–114, 2015.

[53] Claas Lorenz, David Hock, Johann Scherer, Raphael Durner, Wolfgang Kellerer, Steffen Gebert, Nicholas Gray, Thomas Zinner, and Phuoc Tran-Gia. An SDN/NFV-enabled enterprise network architecture offering fine-grained security policy enforcement. *IEEE Communications Magazine*, 55(3):217–223, 2017.

[54] Ridha Khedri, Owain Jones, and Mohammed Alabbad. Defence in depth formulation and usage in dynamic access control. In M. Maffei and M. Ryan, editors, *Proceedings of the 6th International Conference on Principles of Security and Trust (POST), co-located with the Twentieth European Joint Conferences on Theory and Practice of Software (ETAPS)*, volume 10204 of *Lecture Notes in Computer Science*, pages 253–274, Uppsala, Sweden, April 22–29 2017. Springer.

[55] Kai Jander, Lars Braubach, and Alexander Pokahr. Practical defense-in-depth solution for microservice systems. *Journal of Ubiquitous Systems and Pervasive Networks*, 11:17–25, 05 2019.

[56] Peter Höfner, Ridha Khedri, and Bernhard Möller. Feature algebra. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science series*, pages 300 – 315. Springer, August 21 – 27 2006.

[57] Peter Höfner, Ridha Khedri, and Bernhard Möller. Algebraic view reconciliation. In *6th IEEE International Conferences on Software Engineering and Formal Methods*, pages 85 – 94. Cape Town, South Africa, November 10 – 14, 2008.

[58] Peter Höfner, Ridha Khedri, and Bernhard Möller. An algebra of product families. *Software & Systems Modeling*, 10(2):161–182, 2011.

[59] Idris Zoher Bholebawa, Rakesh Kumar Jha, and Upena D. Dalal. Performance analysis of proposed openflow-based network architecture using mininet. *Wireless Personal Communications*, 86(2):943–958, 2016.

[60] Arash Shaghaghi, Mohamed Ali Kaafar, and Sanjay Jha. Wedgetail: An intrusion prevention system for the data plane of software defined networks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 849–861, New York, NY, USA, 2017. Association for Computing Machinery.

[61] Jason Jaskolka and Ridha Khedri. Mitigating covert channels based on analysis of the potential for communication. *Theoretical Computer Science*, 643: 1–37, August 2016. ISSN 0304-3975.

[62] Jason Jaskolka, Ridha Khedri, and Qinglei Zhang. On the necessary conditions for covert channel existence: A state-of-the-art survey. In E. Shakshuki and M. Younas, editors, *Proceedings of the 3rd International Conference on Ambient Systems, Networks and Technologies*, volume 10 of *Procedia Computer Science*, pages 458 – 465, Niagara Falls, Ontario, Canada, August 2012. Elsevier.

[63] Jason Jaskolka and Ridha Khedri. A formulation of the potential for communication condition using c2ka. *Electronic Proceedings in Theoretical Computer Science*, 161:161–174, Aug 2014. ISSN 2075-2180. . URL http://dx.doi.org/10.4204/EPTCS.161.15.

[64] Alicia Marinache, Ridha Khedri, Andrew Leclair, and Wendy MacCaull. DIS: A data-centred knowledge representation formalism. In *IEEE International Conference on Reconciling Data Analytics, Automation, Privacy, and Security (RDAAPS)*, pages 1–8, Hamilton, Ontario, Canada, May 18 - 19 2021. IEEE Computer Society.

[65] Andrew LeClair, Ridha Khedri, and Alicia Marinache. Toward measuring knowledge loss due to ontology modularization. In Jan Dietz, David Aveiro, and Joaquim Filipe, editors, *In Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2019)*, volume 2 of *IC3K*, pages 174–184, Vienna, Austria, September 17–19 2019. SCITEPRESS Science and Technology Publications, Lda. ISBN 978-989-758-382-7.

[66] Ridha Khedri, Fei Chiang, and Khair Eddin Sabri. An algebraic approach for data cleansing. In *the 4th International Conference on Emerging Ubiquitous Systems and Pervasive Networks*, volume 21 of *Procedia Computer Science*, pages 50 – 59. Procedia Computer Science, October 2013.