Journal of Theoretical and Applied Information Technology
15th January 2021. Vol.99. No 1
© 2005 – ongoing JATIT & LLS

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

A FIREWALL-ADVERSARIAL TESTING APPROACH FOR SOFTWARE DEFINED NETWORKS

[1] RAMI MALKAWI, [2] IZZAT ALSMADI, [3]AHMED ALEROUD, [4]PAVEL PETROV

[1] Assistant Professor, Yarmouk University, Information Systems Department, Jordan

[2]Associate Professor, Texas A&M University - San Antonio, USA

[3]Associate Professor, Yarmouk University, Information Systems Department, Jordan

[4]Associate Professor, University of Economics - Varna, Varna, Bulgaria

E-mail: [1]rmalkawi@yu.edu.jo, [2] izzat.Alsmadi@tamusa.edu, [3]ahmed.aleroud@yu.edu.jo, [4]petrov@ue-varna.bg

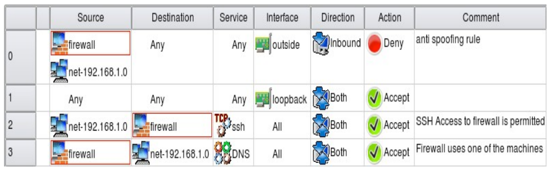
ABSTRACT

Software Defined Networks (SDN) recently evolves to give more roles to software in network control and management. It is feared that such significant roles may risk those networks in terms of reliability and security. As a new architecture, thorough testing and evaluation should take place to ensure that those networks are robust and reliable. In this paper, we focused on testing firewall modules built on top of SDN. We modeled typical interactions between those modules and the network based on flow and firewall rules. We believe that, in future, all security controls including firewalls should be deployed as software services, created in real time, as instances and deployed without any human intervention. This paper describes also an approach that generates synthetic attacks that can target SDNs using an Adversarial approach. It can be used to create models that test SDNs to detect different attack variations. It is based on the most recent OpenFlow models/algorithms and it utilizes similarity with known attack patterns to identify attacks. Such synthesized variations of at-tack signatures are shown to attack SDNs using adversarial approaches.

Keywords: *SDN, OpenFlow, Software evaluation, Model based Testing.*

1. INTRODUCTION

Software Defined Networking (SDN) splits the control plane from the data plane and allocates the control functions in a dedicated software-based controller. The controller communicates with its switches through OpenFlow algorithm. Controller decides the fate of incoming and outgoing traffic and inserts flow rules in switches flow tables. Those rules are added dynamically based on current network traffic. Rules become obsolete after idle time is passed without being used. Rules can be also updated frequently. Traditionally, the firewall security applications were taking the role of deciding the fate of network traffic. They can block or permit traffic based on rules that are added to the firewall table by network administrators. In that sense SDN controller acts as the traditional firewalls in deciding the fate of network flows. However, the major difference is that flow table rules in switches that controller remotely adds are dynamic; they are added, updated or removed dynamically based on network traffic and state. On the other hand, firewall rules are static and they are only updated manually through network administrators. Investigations on how SDN networks work show that controller itself performs some of tasks that firewalls in traditional networks perform. SDN controller make decisions related to what to do with packets. SDN firewalls then need to work as supporting modules to the controller itself. Firewalls, software, hardware, or mixed, are responsible for monitoring network traffic to permit or deny this traffic based on certain criteria specified by network administrators and exist in policies or access control lists (ACLs). Typically, most traditional firewall systems work in information from layers 2-3 of the seven layers in the OSI model (i.e. Port number, IP and MAC addresses). Particularly, you can define firewall rules to prevent or permit data based on: IP addresses, Ports, or MAC addresses. Figure 1 below shows typical examples of firewall rules in traditional networks.



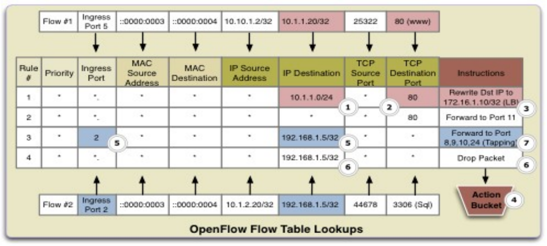
		Source	Destination	Service	Interface	Direction	Action	Comment
0		firewall net-192.168.1.0	Any	Any	outside	Inbound	🔴 Deny	anti spoofing rule
1		Any	Any	Any	loopback	Both	✓ Accept	
2		net-192.168.1.0	firewall	TCP ssh	All	Both	✓ Accept	SSH Access to firewall is permitted
3		firewall	net-192.168.1.0	DNS	All	Both	✓ Accept	Firewall uses one of the machines

Figure 1: Firewall rules, examples (traditional networks)

The figure shows main attributes that should be specified in each firewall rule. Those include IP, MAC and Port addresses for source and destination. If the user selects the option (any, wild cards), then that will be a general flag with no specific source or destination. For example, the first firewall rule in indicates that all traffic going from the firewall (as an IP address) to any destination should be denied or blocked. Service option includes protocol using this traffic. Interface indicates the network card that the rule will be applied on. The line below shows another example of a textual firewall rule. The line shows the same information in addition to specific inbound and outbound ports. However in some cases, without using complex technical means, the content of the network packets which are used by text protocols can easily be read [1].

There can be usually some other options related to whether events should be recorded or not and some other optional features that may vary from one firewall vendor to another.

In SDN, a firewall module can be added typically as a northbound (REST) API to the controller. REST API is a standard add-on interface for interacting with the controller and adding applications to the network.



Flow #1	Ingress Port 5	::0000:0003	::0000:0004	10.10.1.2/32	10.1.1.20/32	25322	80 (www)

Rule #	Priority	Ingress Port	MAC Source Address	MAC Destination	IP Source Address	IP Destination	TCP Source Port	TCP Destination Port	Instructions
1	*	*	*	*	*	10.1.1.0/24	*	80	Rewrite Dst IP to 172.16.1.10/32 (LB)
2	*	*	*	*	*	*	*	80	Forward to Port 11
3	*	2	*	*	*	192.168.1.5/32	*	*	Forward to Port 8,9,10,24 (Tapping)
4	*	*	*	*	*	192.168.1.5/32	*	*	Drop Packet

Flow #2	Ingress Port 2	::0000:0003	::0000:0004	10.1.2.20/32	192.168.1.5/32	44678	3306 (Sql)

Figure 2: SDN firewall rules' example (OF 1.0)

Figure 2 shows examples of OpenFlow version 1.0 Firewall rules. OpenFlow controller has some common functionalities with firewalls. In particular, both make decisions on blocking or permitting network traffic. However, the major difference is that controller can insert rules dynamically in response to network traffic and state. Controller continuously evaluates current topology by using a link discovery module. It generates LLDP and broadcasts packets routinely to neighboring switches. Based on response from those switches, controller can predict current network topology. Additionally, controller includes also a learning switch module that learns about new devices based on their MAC addresses. Rules are inserted dynamically by the controller into the switches' flow tables.

We think that testing controller applications is a large research open area. There are several reasons for that. First, this is due to the large number of currently available controllers, open or commercial. Exact and detail functionalities may not be the same. Security problems related to the programming language or to the program itself can be expected given that this is a new area or a new way of dealing with network traffic or information. Formal testing approaches have limitations related to robustness and dealing with state explosion problems. Traditional testing methodologies and coverage aspects can be applied to the controller program as well as any developed modules on the controller. Second, since security is one of the major challenges in SDNs. Researchers argue that SDNs are vulnerable and easier to target, and several authors proposed IDSs to protect SDNs. However, it is challenging to identify all attacking techniques that may target SDNs. Adversaries have enough knowledge and motivation to attack and bypass those systems, since most IDSs rely on classification algorithms and it is possible to create examples that evade those classifiers. In this paper we also follow an approach that utilizes labeled regular flows to analyze samples of flows generated using SDNs, we then show how it is possible to evade SDN-based IDSs using synthetic samples that are created using Generative Adversarial Networks (GAN). Third, Motivated by the new relation between firewalls and networks, we focused in this paper on testing SDN firewall modules and their interactions or communications with SDN network in general and controller in particular. This is since as we mentioned earlier, there are cross functionalities between what the controller and the firewall are doing. We defined a state based model that can best describe the nature of interaction between SDN controller, firewall modules and switches' flow tables. Flow tables in switches are the common area that both firewall module and SDN controller's decisions impact. Our contribution in this regard is to envision how future firewalls should be developed. This can be summarized in the following:

1. Creating centralized access control: Current firewalls typically work in L2-L3 OSI layers. In addition, there is another type of firewalls that act in the high layer (i.e. L7-firewals). Nonetheless, access control decisions are approximately taken in

all network levels or layers. This causes several problems related to conflicts or inconsistency of access control decisions. In our model, we showed, how based on SDN, firewalls can be developed with a centralized access control decision.
2. Model-based development of firewalls: Despite the fact that major functionalities of classical network firewalls are very simple, yet many vendors develop firewalls with different functionalities. This makes testing firewalls in general for conformance testing for example complex due to the need to customize such tests for each vendor firewalls. We should create a model for firewalls that should reflect the main abstracts or functionalities in firewalls where we believe that all software firewalls can then be different instances from such high-level firewalls.
3. Based on our firewall models, we showed how all testing activities can be automatically created to evaluate any instance of a firewall.
4. One of the major ambitious goals for future firewalls is to automatic the current firewall activities that are largely manually accomplished by network administrators. This includes creating, modifying, deleting, etc. firewall rules, network topologies, etc. We showed how those tasks can be automated in future to develop firewalls that are completely autonomous.

The testing of adversarial attacks is conducted based on the following steps

1. Generation of attack examples on SDNs: it is accomplished through feature perturbation implemented using GAN. The results show that GAN networks are very effective in creating adversarial examples that can fool machine learning detection models for SDN.
2. Synthesis of SDN intrusion detection datasets: to the best of our knowledge, this is the first work to synthesize adversarial examples against GAN, which leaves the door open for new defensive mechanism based on the level of perturbation.
3. Evaluation: The existing GAN cyber security models are tested based on existence or absence of malicious features in attack examples. However, data may also contain suspicious or borderline features, which cannot be classified as benign or attacks. The experimental results proved the success of generating adversarial models for SDN examples using GAN. A significant number of the generated examples evaded different intrusion detection algorithms.

The rest of the paper is organized as the following: In section 2 we provide a research background and several motivations for our work. In section 3 we will introduce several research papers that are relevant to the paper subject. In section 4 we present goals and approaches for our model based SDN testing. In section five, we conduct experiments and their results. Paper is then concluded with a summary in section 6.

2. RESEARCH BACKGROUND

In OpenFlow, controllers store rules or Access Control Lists (ACL) for all network switches. Those rules can be sorted based on the priority attribute where if rules contradict with each other, the one with higher priority will be applied on the subject flow. If no rule is matched with the current flow, the flow is forwarded to the controller to make a decision about. Subsequent packets in the same flow are judged based on the decision of the first evaluated packet. In firewalls where priority does not exist or does not apply, conflicting rules can be handled in different manners. For example, some firewalls use the last matching rule option. In some other firewalls, the first matching rule is applied. This may however indicate a system inconsistency related to the fact that there are many firewall rules in the same firewall that contradict or contain each other completely or partially. While a firewall may not need to get rid of such cases all the time, nonetheless, it should be able to know such occurrences and handle them consistently. Consequently, we believe the existing techniques needs to emphasis on the following aspects:

- While Controllers authorize inserting rules or flows inside switches, this should be implemented in coordination with the firewall module. We performed some experiments and noticed that the controller, without a firewall module performs certain rules and checking on how to add or remove flows. We added and enabled a firewall module (using Floodlight controller) and noticed that it is not clear how controller synchronizes its decisions with the firewall module [2].
- Thorough testing should be conducted in this area to make sure that added or developed firewall modules on top of the controller are completely consistent with the controller itself.
- SDN can be a supportive tool for network testing. Testing and fixing errors by large includes three main steps: The first one is to run the program looking for problems based on a

reference (e.g. requirements, code, performance, etc). If errors are found a debugging process starts to find locations and causes of errors.

- Traceability analysis is used in testing to evaluate impact or connection between a system component with another. For example, in software programs, testers evaluate traceability between requirements and code to make sure that all requirements are developed, no more and no less. It can be also used for maintenance or reconfiguration purposes. Traceability can be also evaluated between test cases and code to evaluate coverage. In OpenFlow testing, we can develop several instances of traceability. Traceability between source code and flow tables is important specially if such traceability can be evaluated at run time automatically. It can show a direct view on the code that is changing flow tables. This can be important for both testing and diagnosis as well as for security or vulnerability assessment. In addition, we may use traceability analysis methods for rules' merging and optimization. There are existing research proposals for rules optimization process particularly in traditional firewalls[3, 4]. However, OpenFlow includes new artifacts and the dynamic nature in those artifacts makes static optimization approaches insufficient.

3. LITERATURE REVIEW

Many authors tried to use traditional testing techniques to evaluate different aspects of SDN. Software programs written in SDN can be classified in several different categories: Controllers, APIs, or middle-boxes and applications written on top of the network to interact with the controller through APIs. However, extensive testing and quality assurance methodologies should be conducted to evaluate how much such programs conform to SDN or OpenFlow guidelines. From security perspective in particular, little work has been done to test those programs for security problems or vulnerabilities. Natarajan et al. [5] demonstrated developing an OpenFlow controller in cloud computing. Authors discussed several code level issues and problems with possible choices. For example, timeout variables are included in the program to decide when to drop a flow after an idle time when there is no proper match or when traffic is in progress for a while.

E. Al-Shaer and Al-Haj [6], presented a tool called FlowChecker to check possible miss-configuration in switch flow table(s). Networks can be modeled as BDD (Binary Decision Diagrams). Conflicts between the different rules are checked statically or dynamically. The tool is intended to be used offline where the input is the content of the flow table and the output includes possible conflicts or miss-configurations. Different rules may contradict or shadow each other. SAT based model checking is used in [7] for data plane verification. Their approach revealed several bugs in the campus network. Real time verification can be used in OpenFlow for: OpenFlow policies, controller or modules' programs, flow table rules, etc. Real time decision making can be risky if for example a legitimate host is falsely blocked or the opposite. Canini et al. [8] combined symbolic execution with model checking for OpenFlow testing. SDN controller is modeled as a state machine. They developed an open source tool for model-based testing of OpenFlow applications. The challenge of applying formal models in SDN is the applicability of such approaches in real time scenarios. Zeng et al. [9] focused on the automatic generation and execution of test packets in OpenFlow networks. Khurshid et al. [10] developed VeriFlow that aims at verifying dynamically the correctness of the network variants in wide and also verify some security properties and fault tolerance. The system is implemented to make decisions in real time and with a very short response time. Such process can be triggered whenever a network change in configuration occurs. From testing perspective, VeriFlow defines equivalent classes of traffic flows where behavior is expected to be the same for all members in the same class. Test cases that represent flows are taken from each class. Classes are extracted based on flow variables (e.g. IP address, MAC address, etc.). Typical to most proposed northbound APIs, VeriFlow is developed to act between the controller and northbound applications. It acts as a supporting module to the controller to support and verify decisions made by the controller before they can be enforced.

Kloti et al. [11] discussed OpenFlow protocol security analysis based on Microsoft STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege) threat analysis. They conducted analysis on evaluating security issues related to the protocol itself. They focused on threats on the data plane only. In reality, there is a wide range of possible threats on the SDN network. Several attack trees are demonstrated representing different network security attacks and how they can be deployed from: Vulnerability exploits intrusion to payload.

Handigol et al. [12] used traceability of packets' history as a tool for testing and faults' localization. Packet history is the journey packets take from source till destination through switches and including different headers' modifications. Ball et al. [13] presented VeriCon; a formal tool to verify SDN programs at compile time based on possible topologies and events. This is a formal approach that models SDN controller as a state machine and is then used to formally prove the correctness of software programs. Topologies are expressed in terms of first order logic. A simple imperative language is used to write SDN programs. Authors defined invariants related to topology, safety, and transition and then tried to proof the correctness of those invariants or properties. Lebrun et al. [14] proposed a requirement based testing for SDN programs. They focused on data path requirements and tried to check if SDN controller complies with those requirements. There have been some proposals to enhance firewall queries in both design and evaluation. Gouda and Liu [15] and Liu [16] proposed a rich language for more expressive firewall rules. Authors indicated that such expressiveness or semantic is required in both designing and evaluating firewall rules. They proposed Structured Firewall Query Language (SFQL) to describe firewall policies and Firewall Query Theorem and processing algorithm for processing firewall rules.

Such approaches can help in developing firewall rules and policies that have much more semantic and expressiveness. This can help reduce the number of firewall rules and also improve performance in processing firewall queries. For processing they used FDD Firewall decision diagrams. However, authors assumed only two attributes in each rule (source and destination address). If authors made their evaluation (i.e. 10,000 rules mentioned in the abstract) in a full decision tree with rules of full features, processing time can be far more than what they have reported. In reality, all rule attributes should be included which may make the tree very large and complex. This is especially true given that adding more semantics to firewall rules mean giving them more attributes or attributes' values. OpenFlow 1.3 includes 40 attributes related to the flow details. We think that using some dynamic tree structures can help model firewalls dynamically. The tree can dynamically grow based on the actual number of firewall rules. Nelson et al. [17] introduced the Margrave tool for firewall rules' analysis to support queries at multiple levels. They defined 9 firewall sub-policies based on the decomposition of firewall configurations. Their implementation is applied on traditional firewalls. There are some significant changes on how firewalls work in SDN which make such concrete traditional firewall implementation inapplicable to SDN firewalls.

Unlike signature or dictionary-based intrusion detection techniques, anomaly and role-based detection methods utilize complex methods and knowledge from large historical data to detect and protect against network intrusions. Software Defined Networking (SDN) can help this area through the ability enable users and their applications to aggregate data from different sources and also enable autonomous and real time traffic manipulation and response. Unlike firewalls, IDS/IPS system employ different methods of learning techniques such as deep learning to be able to distinguish normal from malicious traffic.

SDN offers new opportunities to security controls such as IDS/IPS but also pose some concerns. Opponents of SDN centralized controllers indicate that such centralized controller can be an attractive target to different types of intrusion attacks such as flooding attacks [18, 19, 20]. As SDN controllers are expected to be reactive to traffic and employ real time traffic roles to optimize network activities and utilization, attackers can take advantage of such controller behavior and flood the controller with malicious traffic. It is important for controllers to have intelligent methods (e.g. using learning techniques) to distinguish and deal with such malicious behaviors. Researchers indicated the ability of deep learning methods or approaches to overcome some of the challenges found in NIDS environments [21, 22, 23, 24].

Deep learning can be used in analyzing network security since it uses statistical data to calculate any network problems by evaluating the interrelation of neutrals in the system [25]. Examples of deep learning algorithms proposed in this area include deep learning deep neural networks, auto encoders, convolutional neural networks and recurrent neural networks.

On the opportunity side, several papers in literature discussed utilizing SDN to employ global and/or centralized access controls and IDS/IPS to centralized and unify actions to permit or deny traffic [26, 27, 28, 29]. Being in the central of the network, SDN controllers can receive information from the network and system agents that can be deployed across the network in different premises and layers, and eventually unify and centralized decisions on traffic travelling throughout the network. As one of the major attack goals against SDN, most papers evaluated deep learning techniques in SDN-based networks focused on flooding attacks such as Denial of Service and Distributed Denial of Service [30, 31]. Niaz et al. paper [30] evaluated using Stacked Autoencoders in detecting flooding attacks in flow-based networks. Several autoencoders are ensembled or stacked as intermediate input-output models with input features extracted from the incoming packets at the flow level (e.g. number of bytes and packets per flow, entropy, etc.). The incoming packet and extracted features are also classified based on three main protocols: TCP/UDP and ICMP. The hidden layers in auto-encoders can be the simple concepts and multiple hidden layers are used to provide analysis depth and extract unknown knowledge or insights [32].

Dey and Rahman in [33] proposed using recurrent neural networks due to their ability to learn from historical data. They utilized Long Short-term Memory (LSTM) architecture variants such as Gated Recurrent Unit (GRU). Deep recurrent neural networks can have various architectures and more layers from RNN, LSTM or GRU can be added to their architectures. Researchers in [34] used deep neural networks to detect intrusions in flow-based traffic.

As an alternative to deep learning, Han et al. [35] paper proposed using reinforcement learning in flow-based intrusion detection. Reinforcement learning is an outcome-based learning method that supports models through employing several learning cycles and through evaluating outputs of previous cycles to improve future ones.

Traceability analysis: Traceability analysis is used in testing to evaluate impact or connection between a system component with another. For example, in software programs, testers evaluate traceability between requirements and code to make sure that all requirements are developed, no more and no less. It can be also used for maintenance or reconfiguration purposes. Traceability can be also evaluated between test cases and code to evaluate coverage.

In OpenFlow testing, we can develop several instances of traceability. Traceability between source code and flow tables is important specially if such traceability can be evaluated at run time automatically. It can show a direct view on the code that is changing flow tables. This can be important for both testing and diagnosis as well as for security or vulnerability assessment.

Traceability analysis is also important to conduct between firewall rules and policies along with flow tables in switches. Many papers discussed the need to make sure that there are no conflicts between those two artifacts. In addition we may use traceability analysis methods for rules' merging and optimization. It is expected with the dynamic insertion of rules by the controller in flow tables, that flow tables will grow in size and include a large number of flows that have redundancy and conflict. The need to continuously trace flow table rules to merge rules that can be merged or remove rules that can be redundant is a very important process in the current large and dynamic networks. However, achieving such process automatically can be a challenge especially as OpenFlow switches are not supposed to have control or intelligence (in original OpenFlow design). The controller or one of its supporting modules is better to perform such optimization process especially as optimization is also necessary between the different network switches. We think that this is an important research problem given that proposed solutions should consider scalability and overhead issues. There are existing research proposals for rules optimization process particularly in traditional firewalls (e.g. [3, 4]). However, OpenFlow includes new artifacts and the dynamic nature in those artifacts makes static optimization approaches insufficient.

Firewall-Rules-SQL-Like query language: One of the problems when dealing with firewalls is that they don't include a rich (SQL like) process to add or update firewall rules. For example, administrators must enter firewall rules one by one for a case when a range of ports should be allowed. This is since there is no constructs to indicate a range of ports, IP addresses, MAC addresses. The only available one is the: wild card, or [any] which is very open and generic and may not be helpful in many cases. There have been some proposals to enhance firewall queries in both design and evaluation.

Liu and Gouda paper in [15, 16] proposed a rich language for more expressive firewall rules. Authors indicated that such expressiveness or semantic is required in both designing and evaluating firewall rules. They proposed Structured Firewall Query Language (SFQL) to describe firewall policies and Firewall Query Theorem and processing algorithm for processing firewall rules. Such approaches can help in developing firewall rules and policies that have much more semantic and expressiveness. This can help reduce the number of firewall rules and also improve performance in processing firewall queries. For processing they used FDD Firewall decision diagrams. However, authors assumed only two attributes in each rule (source and destination address). If authors made their evaluation (i.e. 10,000 rules mentioned in the abstract) in a full decision tree with rules of full features, processing time can be far more than what they have reported. All rule attributes should be included which may make the tree very large and complex. This is especially true given that adding more semantics to firewall rules mean giving them more attributes or attributes' values. OpenFlow 1.3 includes 40 attributes related to the flow details. We think that using some dynamic tree structures can help model firewalls dynamically. The tree can dynamically grow based on the actual number of firewall rules. Nelson et al. [17] introduced the Margrave tool for firewall rules' analysis to support queries at multiple levels. They defined 9 firewall sub-policies based on the decomposition of firewall configurations. Their implementation is applied on traditional firewalls. There are some significant changes on how firewalls work in SDN which make such concrete traditional firewall implementation inapplicable to SDN firewalls.

There have been some similar works that investigated the feasibility of fingerprinting the controller-switch interactions by a remote adversary, whose aim is to acquire knowledge about specific flow rules that are installed at the switches. This knowledge empowers the adversary with a better understanding of the network's packet-forwarding logic and exposes the network to several threats [36]. The authors in [37] recreate the escalating competition between scans and deceptive views on a Software Defined Network (SDN). Our threat model presumes the defense is a deceptive network view unique for each node on the network. It can be configured in terms of the number of honeypots and subnets, as well as how real nodes are distributed across the subnets. It assumes attacks are NMAP ping scans that can be configured in terms of how many IP addresses are scanned and how they are visited.

4. GOALS AND APPROACHES

The main goal of this paper is to test SDN based firewall modules and then try to apply adversarial modeling. Those modules are built of SDN or OpenFlow networks. They interact with the network through the controller. Communication with the controller is accomplished through the northbound API. Here are some of the questions that our approach will experimentally target:

- If SDN controller can decide the fate of traffic flows similar to the main task of traditional firewalls, do we still need firewalls in SDN? Either firewall rules will override switches' rules as it is centralized as part of the controller, or either it is expected to do more intelligent decisions more like an IDS.
- Are there any possible conflicts that may arise between controller and firewall decision on traffic fate? When conflicts can occur, and which decision dominates? How power share is going to be distributed between them?
- Policies should be developed and enforced by the controller on the switches and hence mature firewalls should exist in or supporting the controller and centralized. However, orchestration should be made between them and switches so that to handle conflicting cases. For example, if a firewall rule denies entries from IP address 192.168.0.1 and one switch permits this, most likely packets from this IP address will be dropped from the firewall before reaching the specific switch to make its own decision.
- If ACLs are migrated from traditional firewall to SDN, where should rules be migrated? To the firewall or to the switches? Can we have stateful migration? As typically migration will be rule by rule. How should we best distribute rules between the firewall and the different switches?
- If switch rules that contradict firewall rules will not be evaluated or tested, how could this be evaluated dynamically and continuously specially as flow table content changes frequently?
- Should firewall insert or drop flows on the controller behalf? Assuming that it may not be a northbound module? Does that contradict with the fact that control is centralized? How can controller delegate some of its responsibilities to firewall module?
- If controller or firewall makes judgment for packets based on initial packets of a complete traffic, will their not be some problems if traffic has sub-packets to different hosts? We did investigate this to start with PINGALL command that will send small packets to all hosts and noticed that if first part is denied all will be denied as controller will write (deny all traffic) without looking at the rest of the traffic. How much similar problems can happen in real time?!

- Shouldn't we have an option for some firewall rules to exist but not activated?! Maybe we need that sometime. In some other cases, maybe we need timing with firewall rules where some rules need to be activated for a certain period of time or only in working hours, etc. How could we implement that?
- If adversarial modeling can be used to evade security controls in SDNs
- We first proposed a state base model to describe firewall module interaction with SDN. The model is intentionally made simple to serve the following goals:
- Possible states in the model and possible transactions are finite.
- Testing activities (i.e. test case generation, execution and verification) should be simply and automatic.
- Given the large number of possible input values if we want to consider all possible values for flow inputs, model should abstract possible inputs into finite classes.

Table 1: Test cases for attributes firewall module, firewall rules, and switch flow table

State	Firewall enabled?	Firewall empty?	Flow table empty ?
S1	YES	YES	YES
S2	YES	NO	NO
S3	YES	NO	YES
S4	YES	YES	NO
S5	NO	YES	YES
S6	NO	NO	NO
S7	NO	NO	YES
S8	NO	YES	NO

Here are the steps to produce the model.

- The model is based on three binary attributes: Firewall module (enabled or disabled), firewall rules table (empty or not) and switch flow table (empty or not). For simplicity, we assume one switch with one flow table. Table 1 shows the 8 possible states given the three previously described attributes.
- We define also 10 possible events that may cause transitions between those states (Firewall: enable, disable, Firewall rules CRUD (i.e. Create, Read, Update and Delete) and Flow table rules CRUD (i.e. Create, Read, Update, and Delete).
- Events cause states' transitions. For example, we describe impact of all events on network state S1 as following (C on firewall is Cfw, C on flow table is Cft and so on)

(S1->enable->S1;S1->disable->S5;S1->Cfw->S3;

S1->Rfw->*S1*;S1->Ufw->*S1*;S1->Dfw->*S1*;S1->Cft->S4;

S1->Rft->S1;*S1*->Uft->*S1*;S1->Dft->*S1*).

What can we learn from the example of events-transitions-sequence for S1 state:

- Transition from S1 state is only possible to S3, S4, and S5 states. Other states should not be reachable from S1.
- We made some state transitions in italic to indicate that there can be possible errors in those states. Will reading, updating or deleting from an empty table (firewall or flow) cause a null error?

In order to reduce the state space of possible inputs for our model and hence test cases, we made some assumptions. We ignore any details related to the nature of parameters in either the firewall or the flow table rules. By considering CRUD methods (Create, Read, Update and Delete), We assumed a verification process that does not depend on the actual content for added, deleted, updated or read firewall or flow rules. We will evaluate the rules' contents in a second layer. Hierarchical models are used to reduce the number of possible states vertically. For example, the event "Create" should be further divided according to the number of variables in the flow or firewall rule. If (Cft) refers to creating a firewall rule in general then: Cft_IP refers to the family of test cases to create firewall rules where the variable that should only change is the IP address. Figure 3 below shows the hierarchical state diagram with three levels.

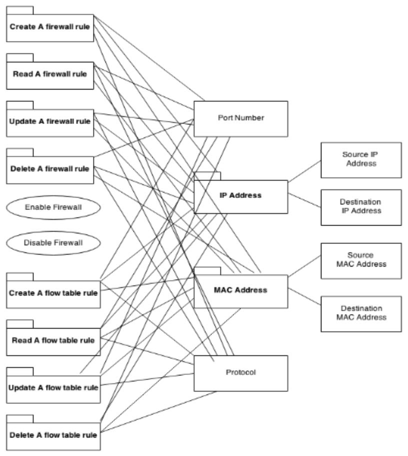


Figure 3: OpenFlow Hierarchical State Model

The first level includes the 10 events we discussed in the first experiment. However, CRUD events are now represented by packages of events. Each one of those events should include: IP address, MAC address, Port number and protocol. IP and MAC address represent also packages as each flow instance requires two of those; source and destination.

Testing SDN Using Data Generated by GAN : Recently, deep-learning adversarial models have attracted quite an attention in Cyber Security. As opposed to traditional techniques, GANs apply a set of non-linear transformations on an original malicious sample to generate an adversarial example that evades classification models. GANs have shown some promising results in intrusion detection [38]. The GAN structure in this paper consists of a Generator network, a Discriminator network and an intrusion detector that handles both OpenFlow and NonOpenFlow traffic. Feature vectors of attacks against SDNs consist of the regular features of a network traffic that are converted into binary values of 0 and 1. A feature vector is represented using a ternary (i.e., three-valued) features, where -1 describes the malicious features, 1 describes legitimate features, and 0 describes suspicious features. We re-encode ternary features into two-bit binary features using the encoding, 0 to 01, 1 to 00, and 1 to 11. Our approach deals with suspicious features that can be classified as borderline features. The original feature vector contains n-ternary features in the original encoding (i.e., n columns). With the proposed encoding, 2n features are created. Each feature in the original data is encoded using two columns, each containing one binary feature. This encoding scheme is applied to both attacks and benign examples. The Generator creates a perturbed version of attack examples to convert them into adversarial examples. The Discriminator learns to fit the intrusion detector, which is implemented using classification algorithms to identify Denial of Service attacks on SDNs. At each round of the training process, the Discriminator sends a feedback to the Generator to modify its weights during the training process to the point where it guarantees that the Generator creates enough examples to evade the intrusion detector. Intrusion examples consist of a feature vector f with n features. Both the input vector f and a noise vector z are fed to the generator. Using our encoding scheme, f consists of m features where m=2n. The features in f take the values of 0 and 1 to identify how malicious the feature is where 11 denotes a very malicious feature. The Hyperparameter z is a vector with random entries in the range [0, 1). The proposed structure of the generator consists of three hidden layers, each with 120 neurons. Hidden layers are activated using LeakyReLU. The output layer consists of 2n neurons, two for each feature, which are all activated using sigmoid function in order to return outputs between 0 and 1. The Generator parameters are updated based on the feedback from the Discriminator. The resulting adversarial examples are binarized using a threshold to create a binary vector with two inputs 0 and 1. However, for backpropagation to work non binarized vectors are used. The perturbation done using GAN preserves the semantics of the original data. In attacks against SDN, it is possible to produce new attacks by removing some features and introducing others.

There is a need to update the weights of the generator using the gradient information from the discriminator. The Discriminator and intrusion detector both take the feature vector f as an input. The Discriminator classifies the given flow as a benign or attack using a single output layer with a certain level of uncertainty. Adam optimizer is used as an optimization function. The training data for the discriminator consists of adversarial sample generated by the generator and the benign sample. The ground truth labels for the discriminator are the predictions made by the intrusion detector, not the actual labels of the samples. Training the generator and discriminator aims at minimizing their loss functions which are measured differently. The predictions of the intrusion detector are used as labels for the discriminators. Therefore, the loss function of the discriminator tries to minimize classification mismatches between the discriminator and the intrusion detector.

5. EXPERIMENTS AND ANALYSIS

From this model presentation, we showed that it can first help us in distributed our test cases in an intelligent rather than random mode. Further, it can help us point to and then focus on some areas that may expose errors or problems. **Experiment1:** We completed the specifications of all states and their transitions and test cases are generated based on those states' transitions.

Table 2: states' transitions, based on the states convention adopted in Table 1

	S1	S2	S3	S4	S5	S6	S7	S8
S1	1,4,5,6,8,9,10		3	7	2			
S2		1,3,4,5,6,7,8,9,10	10	6		2		
S3	6	7	1,3,4,5,6,8,9,10				2	
S4	10	3		1,4,5,6,7,8,9,10				2
S5	1				2,4,5,6,8,9,10		3	7
S6		1				2,3,4,5,6,7,8,9,10	10	6
S7			1		6	7	2,3,4,5,6,8,9,10	
S8				1	10	3		2,4,5,6,7,8,9,10

To show the complete state model, we will give numbers to the events: 1. Enable firewall, 2. Disable Firewall, 3. Create a firewall rule, 4. Read a firewall rule, 5. Update a firewall rule, 6. Delete a firewall rule, 7. Create a flow table rule, 8. Read a flow table rule, 9. Update a flow table rule 10. Delete a flow table rule. Table 2 shows all possible states' transitions, based on the states convention we adopted in Table 1.

As such, we can read the following from Table 2:

- Each state has 3 possible state transitions based on defined events. In other word, for each state, 3 events only should case a state transition. The rest of events should not cause a state change.
- Each state can be reached from three other states.
- Events 1 and 2 (i.e. enable/disable firewall) case 4 different states' transitions each.
- Events 3 and 7 (Create firewall or flow table rule) caused 2 transitions each.
- Events 4, 5, 8 and 9 (read and update for firewall and flow table rules) should cause no state transition.
- Events 6 and 10 (Delete a firewall or flow table rule) may cause a state transition or may not. This is why we include each one of them twice where the event may or may not cause a state transition. The "delete" event will only cause a state transition, if the deleted rule is that last one in the firewall or flow table rule. If the rule that the event is deleting is the last rule, this will
- Based on this model, we can check automatically the network state before and after the event. Each event is developed in our experiment to be a separate test case. This model facilitates the ability to automate the results' verification specially as we are not checking firewall or flow table values. We are only checking the count to see if those tables are empty or not. Test case generation should basically make sure to put the network in all 8 states. Further, test cases should verify correct states' transitions as predicted by the model. 100 % coverage for this model can be achieved using 80 test cases, 10 test case per each state.
-

Experiment 2: From the previous experiment, we had the following observations:

- OpenFlow networks have no direct methods to delete or update flow rules in switches' flow tables. Rules are removed if they pass the idle timeout without usage or the hard time out. Flow rules can be also updated indirectly. For example, if we tried to add a new flow with similar attributes as in a flow rule we may be able to change or replace an existing rule.
- In the first experiment, we verified automatically the correct number of flow or firewall rules. The binary possible states that we assumed do not need to check the actual rules' contents. In the second experiment, we still have 8 possible states. The state model represented by Tables 1 and 2 should not be changed. Events are extended where each one of the 10 events will have six possible alternatives. Total minimum number of test cases to achieve 100 % coverage is then: 8 * 10 * 6 = 480 test case. If we want to valid and invalid inputs for each scenario, this number will be doubled.

Results and analysis for the two experiments are presented. In the first experiment, 80 test cases are generated to achieve 100 % coverage based on the model proposed (i.e.8 state by 10 events). Based on Table 1, we described 8 possible states. We described also the impact of events and the possible transitions from those states.

Based on the model described in Table 1, we developed a test automation framework to automate all testing activities (i.e. test cases generation, execution and verification). Python scripts are used to orchestrate the different testing activities. Pre-conditions and post-conditions for each state transition are programmed based on the state model. For each state, pre-and post-conditions depend on three binary decisions (If the firewall is enabled or not, if the firewall has rules or not and if the flow table has rules or not). For each test case, those three conditions are tested before executing the test case (i.e. pre-condition) and after executing the test case (i.e. post conditions). A successful test case is then the test case that actual post conditions matched its expected ones (i.e. next possible state).

6. SECURITY CONTROLS MODELING

With the evolution of programmable networks (e.g. SDN and NDN) the rule of software in the control and management of networks will continue to expand. Network security controls will also expand in that direction. Future security controls (e.g. firewalls, IDSs, etc.) are expected to be software programs which can be developed just like any other user-level or layer 7 application. Security controls can be also offered as customized or on-demand services where different users can get security controls defined based on their own contexts and domains. In this scope, it is important to divide security control services into two abstraction levels: A high level abstraction level that describes the general functionalities of those security controls that should be similar in all of its instances. The second level can include concrete functionalities that may not be applicable to all its generated instances. In the software paradigm, this is the difference between software templates or classes and objects or instances. The model developed for Floodlight firewall model aims to evaluate main model functionalities. Figure 3 shows abstract firewall class including main methods: CRUD (i.e. create, read, update or delete a rule) and match (i.e. to match a rule with incoming/outgoing traffic). Those are the generic functionalities that any firewall should have.

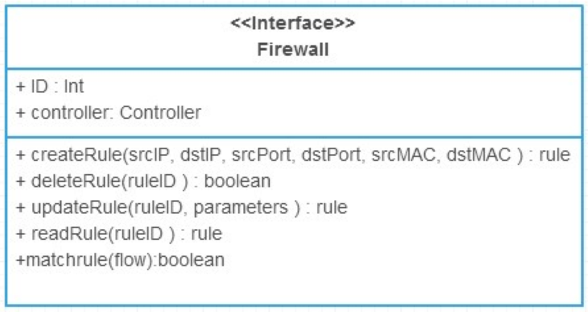

<<Interface>>
Firewall
+ ID : Int
+ controller: Controller
+ createRule(srcIP, dstIP, srcPort, dstPort, srcMAC, dstMAC) : rule
+ deleteRule(ruleID) : boolean
+ updateRule(ruleID, parameters) : rule
+ readRule(ruleID) : rule
+matchrule(flow):boolean

Figure 4: Interface: Firewall

Each instance firewall should inherit from firewall interface and define its own copy of abstract fields and methods. Instance or concrete firewall can also extend original implementation and includes new features. The general logic of how firewalls make decisions include the following constraints:

- A software firewall can act as a service. This means that it can be installed/uninstalled or enabled/disabled. For all features including interface features to be enabled, firewall should be installed. If firewall is installed but disabled, interface services can still be accessed but will not take effect until the firewall is enabled. For firewall to work in its normal operations, it should be installed and also enabled.
- Many firewall programs, as instances can exist in the same system. A one to one relation should exist between the instance firewall and its controller. Such controller decides the jurisdictions in which this firewall can work.
- Firewalls contain rules. Rules are objects that are serialized/de-serialized to a database or a file. All firewall rules are stored in memory when firewall is enabled. One of the significant problems with classical firewalls is that complete or partial redundant rules may exist in the firewall. This may complicate the matching process especially when a certain traffic can be matched by many firewall rules. The current available solutions for such problem is either precedence (i.e. the first matched firewall will take effect) or priority (i.e. the rule with the highest priority will take effect). This however may not be the user or network administrator intention and many cases of inconsistence decisions may arise. When all firewall rules exist in memory, any new attempt to add a new firewall rule should first make sure that such rule does not contradict with an existing rule. Similarly situations of rules' containments

(e.g. where the new rule partially matches an existing rule) should also be identified.

Rules_Vs_Traffic_Matching: The matching (i.e. between firewall rules and incoming/outgoing traffic or flow) is the core task in firewall functionality. One a match is observed, decision in the matching firewall rule is applied on the traffic. There are three match scenarios; when the firewall is enabled and there are rules in the firewall table:

- NO-Match, where the flow does not match any of the firewall rules in the firewall table. In this case, the flow will be permitted. In comparison with 3.a, an enabled firewall with no rules will drop all flows. To reverse this and permit all flows, either disable the firewall or add one random rule (that may not match with any flow).
- Exact-Match where the flow attributes have matched exactly one and only one firewall rule. The flow match can't be with two or more rules as those firewall rules then will be identical (or similar based on current Floodlight firewall implementation). We will use the current approach used in Floodlight firewall where all attributes should exist and match between subject flow and matched firewall rule.
- Inclusive-Match: where the match between a flow and firewall rules (i.e. one or more) can fall within the no- and exact match cases. In other words, the flow can match one or more attributes from different firewall rules. We believe that real network environments and firewalls include a significant number of such cases. Typically, those cases are not solved in a solid solution where firewall has its best judgment of what match should be taken. Rather, match decision is either based on priority or based on first match choice. As our paper focuses on testing existing Floodlight firewall module (which completely ignores inclusive matches), we will only identify inclusive match occurrences without further actions. As mentioned earlier, we accomplished this through template flows and firewallRules that are identified in the model by their names only. This will be handled in future work that focuses on how SDN based firewalls should evolve to accommodate such open issues.

Each test case should include values for the 3 inputs: firewall status, flow and firewall rule. For assertion or test verification, tester is expected also to classify the expected output into one of the 5 possible insertions that are described earlier:

1. InsertFlowFirewallOff(1)
2. InsertFlowFirewallOnFirewallTableEmpty (2)
3. InsertFlowFirewallOnNoMatch(3)
4. InsertFlowFirewallOnExactMatch(4)
5. InsertFlowFirewallOnInclusiveMatch(5)

The model-based automatic assessment of programmable firewalls can be used for the following goals:

- Firewall testing activities: Test cases can be automatically generated, executed and verified based on the model and the actual firewall instance.
- Firewall conformance: As future firewalls are going to be programmable, generated or customized by programs or systems, it is important to ensure that such firewall instances conform to certain standards. As such, existing models can help in creating such standards as well as making sure that created firewalls conform to such standards.
  - **Experiment 3:** Our GAN approach aggregates flow entries exchanged between controller and the OpenFlow (OF) switches. The analysis of the collected OFs emphasizes on discovering similarity of such flows with non-OFs using appropriate classification techniques. We hypothesized that sampling OFs and testing them using an appropriate intrusion detection mechanism can be used as a mechanism to discover threats on SDNs. The first sample in our experiment is taken from a dataset of one hour of anonymized traffic traces from a Distributed Denial-of-Service (DDoS) attack (CAIDA dataset [39]). This type of attack attempts to block access to the targeted server by consuming computing resources on the server and by consuming all of the network bandwidth connecting the server to the Internet. The second sample contains only IP packets. Each record of the dataset represents a packet of several fields such as, packetSize, sourceIP, destination IP, sourcePort, destination- Port, TCPFlags, transport Protocol and packetType [24]. We used Open vSwitch (OVS) as an OpenFlow switch connected to three Linux-based hosts (the attacker, hostl, host2, and host3). The three hosts can only communicate through the OVS switch. We

used GENIExperimenter to create this topology [40]. We utilized an Xen VM with a public IP to run an OpenFlow controller, 1 Xen VM to be the OpenFlow switch, and 3 Xen VMs as hosts. In general, the controller just needs to have a public IP address, so that it can exchange messages with the OpenFlow switch.

Table 3. Samples Used for Training and Testing.

Activity Type	OF	Non-OF
TCP/SYN flood	6214	3216
UDP Flood	-	2512
ICMP flood	6230	3590
Total	12444	9318
TCP/SYN Benign traffic	2712	1100
UDP benign traffic	-	-
ICMP benign traffic	5112	4100
Total	7824	5200
Suspicious and benign	20268	14518
% of suspicious flows	0.61	0.64
% of benign flows	0.38	0.35

- Results: The created instances are used to deceive machine learning-based detectors that are created using KNN and Random Forest Classification Models. We used the original OF and non-OF data with 0.3 OF from both datasets to generate attacks. The reported values in Table 2 represent the success rate of identifying attacks before and after the data is modified using GAN by varying epoch hyperparameter and changing the machine learning classifier. The noise vector contains 10 dimensions. The intrusion detection rates for adversarial examples is between 0.07-0.53 which clearly shows how GAN can still evade the intrusion detection techniques that work on SDNs. In addition to evasion IDSs, our approach can be used to generate private datasets as proved by the values of conditional privacy reported in table IV.

Table 4. Results Using GAN

Z	Test Set Size	Epoch	Detector Type	Activation	Conditional Privacy	Original Attack Detection Rate	Detection Rate After GAN Noise
10	2210	95	NN	LeakyReLU	0.44	0.81	0.51
10	2210	80	NN	LeakyReLU	0.65	0.81	0.24
10	2210	29	Random Forest	LeakyReLU	0.39	0.85	0.53
10	2210	150	SVM	LeakyReLU	0.95	0.83	0.15
10	2210	500	NN	LeakyReLU	1.35	0.81	0.08
10	2210	140	NN	LeakyReLU	2.05	0.81	0.04
10	2210	512	NN	LeakyReLU	1.40	0.81	0.07

7. CONCLUSIONS

In this paper, we proposed two approaches to test the immunity of software defined networks against specific types of attacks. Software-defined networks are introduced to expand the rule of software in network control and management. First, we focused on testing firewalls modules built on top of SDN, we then modeled interactions between those firewall modules and the network based on flow and firewall rules. Our approach is based on a state base model to describe firewall module interactions with SDN controller. We utilized a hierarchical model to reduce the number of possible states. This represents a significant step for developing software-based security controls including firewalls where those security controls are completely autonomous; they can modify their own rules, topology, etc. in response to the network they are deployed in. In addition, we suggested a deep learning-based testing technique to identify attacks on SDNs, we show how Generative Adversarial networks can evade those techniques. The proposed approach synthesizes datasets, taking into consideration the utility- information loss tradeoff. This work can help understand how to use existing attack patterns to discover different attacks that target SDNs. As a future work, we plan to create graph-based detectors for different attack types in SDNs and target those models using adversarial examples.

REFRENCES:

[1] Petrov, P., Malkawi, R., Shichkin, A., Dimitrov, G. & Nacheva, R. "Security Certificates Used in Public Web Sites of Banks in Czech Republic, Slovakia and Hungary". TEM Journal, 8(4), 1224 (2019)

[2] Alsmadi, I., Munakami, M., & Dianxiang, X. Model-Based Testing of SDN Firewalls: A Case Study. In Trustworthy Systems and Their Applications (TSA), 2015 Second International Conference on, 8-9 July 2015 2015 (pp. 81-88). doi:10.1109/TSA.2015.22.

[3] Al-Shaer, E. S., & Hamed, H. H. (2004). Modeling and management of firewall policies. IEEE Transactions on Network and Service Management, 1(1), 2-10.

[4] Al-Saher, E., Hamed, H., Boutaba, R., & Hasan, M. (2005). Conflict classification and analysis of distributed firewall policies. IEEE Journal on Selected Areas in Communications, 23(10), 2069-2084.

[5] Natarajan, S., Huang, X., & Wolf, T. Efficient conflict detection in flow-based virtualized networks. In Computing, Networking and Communications (ICNC), 2012 International Conference on, 2012 (pp. 690-696): IEEE

[6] Al-Shaer, E., & Al-Haj, S. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In Proceedings of the 3rd ACM workshop on Assurable and usable security configuration, 2010 (pp. 37-44): ACM

[7] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P., & King, S. T. (2011). Debugging the data plane with anteater. ACM SIGCOMM Computer Communication Review, 41(4), 290-301.

[8] Canini, M., Venzano, D., Peresini, P., Kostic, D., & Rexford, J. A NICE Way to Test OpenFlow Applications. In NSDI, 2012 (Vol. 12, pp. 127-140)

[9] Zeng, H., Kazemian, P., Varghese, G., & McKeown, N. Automatic test packet generation. In Proceedings of the 8th international conference on Emerging networking experiments and technologies, 2012 (pp. 241-252): ACM

[10] Khurshid, A., Zhou, W., Caesar, M., & Godfrey, P. (2012). Veriflow: verifying network-wide invariants in real time. ACM SIGCOMM Computer Communication Review, 42(4), 467-472.

[11] Kloti, R., Kotronis, V., & Smith, P. Openflow: A security analysis. In Network Protocols (ICNP), 2013 21st IEEE International Conference on, 2013 (pp. 1-6): IEEE

[12] Handigol, N., Heller, B., Jeyakumar, V., Mazières, D., & McKeown, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. 11th USENIX Symposium on Networked Systems Design and Implementation, 2014.

[13] Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., et al. Vericon: Towards verifying controller programs in software-defined networks. In ACM SIGPLAN Notices, 2014 (Vol. 49, pp. 282-293, Vol. 6): ACM

[14] Lebrun, D., Vissicchio, S., & Bonaventure, O. Towards test-driven software defined networking. In Network Operations and Management Symposium (NOMS), 2014 IEEE, 2014 (pp. 1-9): IEEE

[15] Gouda, M. G., & Liu, X.-Y. A. Firewall design: Consistency, completeness, and compactness. In Distributed Computing Systems, 2004. Proceedings. 24th International Conference on, 2004 (pp. 320-327): IEEE

[16] Liu, A. X. (2009). Firewall policy verification and troubleshooting. Computer networks, 53(16), 2800-2809.

[17] Nelson, T., Barratt, C., Dougherty, D. J., Fisler, K., & Krishnamurthi, S. The Margrave Tool for Firewall Analysis. In LISA, 2010

[18] Xu, T., Gao, D., Dong, P., Zhang, H., Foh, C. H., & Chao, H. C. (2017). Defending against new-flow attack in sdn-based internet of things. IEEE Access, 5, 3431-3443.

[19] Tang, T. A., McLernon, D., Mhamdi, L., Zaidi, S. A. R., & Ghogho, M. (2019). Intrusion Detection in SDN-Based Networks: Deep Recurrent Neural Network Approach. In Deep Learning Applications for Cyber Security (pp. 175-195). Springer, Cham.

[20] Manso, P., Moura, J., & Serrão, C. (2019). SDN-based intrusion detection system for early detection and mitigation of DDoS attacks. Information, 10(3), 106.

[21] Ugo Fiore, Francesco Palmieri, Aniello Castiglione, and Alfredo De Santis. Network Anomaly Detection with the Restricted Boltzmann Machine. Neurocomput., 122:13–23, 2013.

[22] Wang, Z. (2018). Deep learning-based intrusion detection with adversaries. IEEE Access, 6, 38367-38384.

[23] Zhang, B., Yu, Y., & Li, J. (2018, May). Network intrusion detection based on stacked sparse autoencoder and binary tree ensemble method. In 2018 IEEE International Conference on Communications Workshops (ICC Workshops) (pp. 1-6). IEEE.

[24] Lakshminarayana, D. H., Philips, J., & Tabrizi, N. (2019, December). A Survey of Intrusion Detection Techniques. In 2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA) (pp. 1122-1129). IEEE.

[25] Dong, B., & Wang, X. (2016, June). Comparison deep learning method to traditional methods using for network intrusion detection. In 2016 8th IEEE International Conference on Communication Software and Networks (ICCSN) (pp. 581-585). IEEE.

[26] Tang, T. A., Mhamdi, L., McLernon, D., Zaidi, S. A. R., & Ghogho, M. (2016, October). Deep learning approach for network intrusion detection in software defined networking. In 2016 International Conference on Wireless Networks and Mobile Communications (WINCOM) (pp. 258-263). IEEE.

[27] Alsmadi, I. (2016). The integration of access control levels based on SDN. International Journal of High Performance Computing and Networking, 9(4), 281-290.

[28] Alsmadi, I. M., & AlEroud, A. (2017). SDN-based real-time IDS/IPS alerting system. In Information Fusion for Cyber-Security Analytics (pp. 297-306). Springer, Cham.

[29] Ibrahim, J., & Gajin, S. (2017). SDN-based intrusion detection system. Infoteh Jahorina, 16, 621-624.

[30] Quamar Niyaz, Weiqing Sun, and Ahmad Y. Javaid. 2017. A Deep Learning Based DDoS Detection System in Software-Defined Networking (SDN). ICST Transactions on Security and Safety (2017).

[31] Qin, Y., Wei, J., & Yang, W. (2019, September). Deep Learning Based Anomaly Detection Scheme in Software-Defined Networking. In 2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS) (pp. 1-4). IEEE.

[32] Shone, N., Ngoc, T. N., Phai, V. D., & Shi, Q. (2018). A deep learning approach to network intrusion detection. IEEE Transactions on Emerging Topics in Computational Intelligence, 2(1), 41-50.

[33] Dey, S. K., & Rahman, M. M. (2018, September). Flow Based Anomaly Detection in Software Defined Networking: A Deep Learning Approach With Feature Selection Method. In 2018 4th International Conference on Electrical Engineering and Information & Communication Technology (iCEEiCT) (pp. 630-635). IEEE.

[34] Javaid, Q. Niyaz, W. Sun, and M. Alam, ‘‘A deep learning approach for network intrusion detection system,’’ presented at the 9th EAI Int. Conf. Bio-inspired Inf. Commun. Technol. (BIONETICS), New York, NY, USA, May 2016, pp. 21–26.

[35] Han, Y., Rubinstein, B. I., Abraham, T., Alpcan, T., De Vel, O., Erfani, S., ... & Montague, P. (2018, October). Reinforcement learning for autonomous defence in software-defined networking. In International Conference on Decision and Game Theory for Security (pp. 145-165). Springer, Cham.

[36] Cui, H., Karame, G. O., Klaedtke, F. and Bifulco, R. "On the fingerprinting of software-defined networks," IEEE Transactions on Information Forensics and Security, vol. 11, no. 10, pp. 2160-2173, 2016.

[37] Kelly, J, DeLaus, M., Hemberg, E., and O'Reilly, U.-M. "Adversarially adapting deceptive views and reconnaissance scans on a software defined network," in 2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), 2019, pp. 49-54.

[38] Zilong, Y. Shi, and Z. Xue. " Idsgan: Generative adversarial networks for attack gene ration against intrusion detection." arXiv preprint arXiv:1809.02077 (2018).

[39] The CAIDA “DDoS Attack 2007” dataset. Available from http://www.caida.org

[40] Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, et al., "GENI: A federated testbed for innovative network experiments," Computer Networks, vol. 61, pp. 5-23, 2014.