SDN Model Based Testing

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 firewalls modules built on top of SDN. We modeled interaction between this firewall modules and the network based on flow and firewall rules.

Index Terms— SDN, OpenFlow, model based testing.

# 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 rule 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 allow or prevent their passage or intrusion based on certain criteria specified by users or network administrators and exist in policies, access control lists (ACLs), etc. This can be applied for both incoming and outgoing traffic. Typically, most traditional firewall systems work in layers 2-3 of the seven layers in the OSI model (i.e. Physical-Data-Link-Network-Transmission, Session, Presentation, and Application). Those two layers are: Data-link and network layers. Particularly, you can define firewall rules to prevent or permit data based on: IP addresses, Ports, or MAC addresses. Figure 4 below shows typical examples of firewall rules in traditional networks.

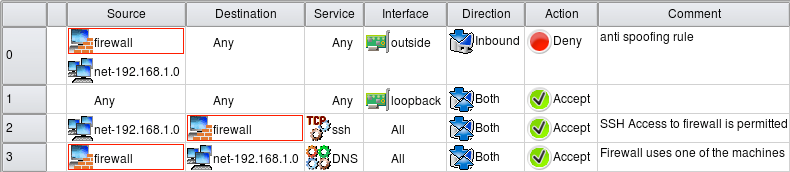


Figure 4: Firewall rules, examples (traditional networks)

Figure 4 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 Figure 4 indicates that all traffic going from the firewall (as an IP address) to any destination should be denied or blocked. Service option can specify the type of applications using this traffic. Interface indicates the network card or interface 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.

**adapter A ip src addr xxx.xxx.x dst addr any tcp src port 20 dst port 80**

There can be usually some other options related to whether such packets should be logged 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 environment for interacting with the controller and adding applications to SDN. Figure 5 below shows SDN examples of OF version 1.0 Firewall rules.

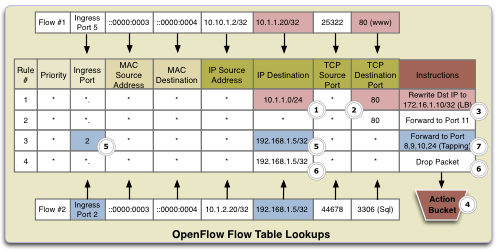


Figure 5: SDN firewall rules’ example (OF 1.0)

As we mentioned earlier, in SDN, controller acts as a firewall (coarse grained firewall). Controllers can continuously evaluate or know current topology by using a link discovery module. Controller generates LLDP and broadcasts packets routinely to neighboring switches. Based on response from those switches, controller can frequently predict current network topology. Controller includes also a learning switch module that learns about new devises based on their MAC addresses. Rules can be added dynamically by the controller to the switches’ flow tables. For example, if a new flow is added to the network, the learning switch checks input and output switches of the flow and also the best route to this flow. This is then added as a new rule to the proper switch.

In SDN, firewall modules have more attributes to define packets through in comparison with firewalls in traditional networks. Both firewall rules and flows or flow table records have included more attributes in SDN tables in comparison with traditional firewalls or routers tables. For example, OpenFlow standard version 1.2 included 13 attributes. However, OpenFlow 1.3 extended the list to cover 40 attributes. Examples of the extra features or attributes include those that cover IP version 6 addresses. As it is an extension many of the extended attributes are left optional.

OF 1.3 considers some of the limitations in earlier versions. One important extension is the ability to have many look-up tables in the flow switch. In fact this was available from 1.1 and above. Each operation has one flow table (e.g. one for ACL and tenant (group of VMs) classification, another one for tenants’ forwarding, matching destination MAC addresses, etc.). This result in minimizing the number of flow entries in each table and hence accelerate the matching process.

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 the effective or the used one. Packets are matched against flow table or firewall rules to decide the proper action. In case of more than one matching rule, the one with the highest priority is applied. If no rule is matched with the current flow, the flow is forwarded to the controller to make decision about. Subsequent packets in the same flow are judged based on the decision of the first evaluated packet. As an example from Floodlight firewall module, below example shows a rule to deny flows based on given attributes.

TCP | 192.168.0.1/24 | 80 | ALLOW | 1 TCP | 192.168.0.1/24 | \* | DENY | 2

As firewall rules are decided based on their priority if they both or all apply to the current packet or flow, rules should be sorted based on their priority. This may improve system performance where wide high priority rules (e.g. those that include wild cards) may behave like a short circuit once matched and subsequent contained rules can be then ignored. However, this should be continuously and dynamically evaluated as this may change on a flow by flow basis.

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 where the last matching rule of several ones is applied. In some other firewalls, the first matching is applied. This however indicates 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.

**FlowTable and Firewall rules**

Whenever a packet passes to OF switches, it matches the packet header with flow table entries to see any possible match. If the switch finds a match, it directs packets based on matched rule decision (e.g. forward, rewrite, drop, etc.) or else the switch will send the packet to the controller if there is no match.

The firewall has the same functionalities and will act on the same way (if firewall module exists and is enabled). Packets will be matched against firewall rules for possible match. It seems that SDN will change the roles of firewalls significantly. Here are some thoughts related to how SDN firewalls may evolve.

**A centrally located virtually distributed firewall**

In SDN architecture, a firewall module is designed to be a northbound application that can be deployed as a module to support the controller. It is then logically located in the controller while it is virtually distributed across network switches. How is that going to be different from traditional firewalls’ deployment?!

Controllers authorize inserting rules or flows inside switches. This should be implemented through 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. This is since in principle, both perform the same tasks; matching current packets with internal rules to make decisions on what to do with those packets.

Thorough testing should be conducted in this area in particular to make sure that added or developed firewall modules on top of the controller are completely consistent with the controller itself.

**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 have to 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 2004, 2009 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. OF 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 2010 paper introduced the Margrave tool for firewall rules’ analysis to support queries at multiple levels. They define 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. In SDN, firewall module acts in the controller or the network operating system and dynamically rewrites rules into switches. Those rules can change frequently based on firewall rules and based on incoming packets.

Motivated by the new relation between firewalls and networks, we focused in this paper in 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 what 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.

The rest of the paper is organized as the following: In section two we will introduce several research papers that are relevant to the paper subject. In section three we will present goals and approaches for our model based SDN testing. In section four, we will conduct experiments and their results. Paper is then concluded with a summary section.

# literature review

In this section, we will select some papers that focus on testing of SDN.

Looking at the papers in this category, we can see that many authors tried to use traditional testing techniques to evaluate different aspects of SDN.

Source code testing is an open research area in SDN testing. Many programs are written to demonstrate software controller functionalities. Other programs are written to demonstrate modules on top of the controller. However, extensive testing and quality assurance methodologies should be conducted to evaluate how much such programs conform to SDN or OpenFlow guidelines.

Traceability analysis is used in testing to evaluate impact or connection between a system components with another. For example, in software programs, evaluators 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 give 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 includes 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 OF switches are not supposed to have control or intelligence (in original OF 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. Al-Shaer et al 2004, 2005). However, OpenFlow includes new artifacts and the dynamic nature in those artifacts make static optimization approaches insufficient.

Test cases are generated to test OpenFlow architecture, source code, flow tables, policies, etc. Traceability analysis methods can be also used to evaluate coverage between those test cases and the different artifacts.

Research in this particular traceability area in SDN testing should focus on methods and tools to perform those traceability evaluations online or offline. For online cases, it is very important to propose solutions with considering scalability and overhead for those solutions.

From security perspective in particular, little work has been done to test those programs for security problems or vulnerabilities.

Natarajan et al 2013 demonstrated developing an OpenFlow controller in cloud computing. Authors discusses 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. Authors claimed that there are no clear good or standard answer for such low level attribute values. Kloeti et al 2012 discussed also the issue of timeout and the tradeoff in packet flows where large timeouts may reduce controller load but then may cause flow tables’ overflows.

Exact same scenario can be applied to or mentioned about TCP acknowledgement message, or what to do in cases of flows’ overlapping, handling malformed packets, etc. From a security perspective, many of those decisions can have a direct impact on making the code more secure or vulnerable.

In their paper, Al-Shaer and Al-Haj (2010) presented a tool called FlowChecker to check possible miss-configuration in switch flow tables. Networks can be modeled as BDD (Binary Decision Diagrams). Conflicts between the different rules can be checked statically or dynamically. The tool is intended to be used statically 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. In reality such tools can be deployed as part of the firewall. We will elaborate on some of the SDN issues related to firewalls in the relevant section.

SAT based model checking is used in Mai et al 2011 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. The second challenge related to real time verification is the ability to obtain the real time view of the topology or the network. This is since such topology can be an important input to the process of real time verification.

Canini et al 2012 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. Although the paper focuses on SDN testing in general, hence it can be extended to security testing in particular. The challenge of applying formal models in SDN is the applicability of such approaches in real time scenarios. This is particularly important for SDN where one of its most important features is that SDN is very dynamic and information changes frequently in real time. Developing a formal model that can automatically cope and adapt with frequent changes can be significantly challenging.

Zeng et al 2012 focused on the automatic generation and execution of test packets in OpenFlow networks. The flows automatic generation process depends on reading router configuration and realizing network topology. Both functional (e.g. rules’ conflicts) and non-functional (e.g. performance) flow based network requirements are evaluated in their testing framework. Authors focused on a small subset of flow variables or those that are very popular (i.e. IP, MAC address, and port). However, current OF flow attributes are 40 that include many different variables. We believe that thorough testing including all those variables is necessary to improve testing quality from coverage perspective.

Khurshid et al in 2012 represents a significant contribution in this section. Authors’ developed application; VeriFlow 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 with real time and 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 middleboxes or 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.

Kloeti thesis in 2012 discussed OpenFlow protocol security analysis based on Microsoft STRIDE (Spooﬁng, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege) threat analysis and attack tree. Authors 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 that we will elaborate more about later on in this paper. Authors demonstrated several attack trees representing different network security attacks and how they can be deployed from: Vulnerability exploit, intrusion to payload.

SDN can be a supportive tool for network testing. Heller et al 2013 argued that network troubleshooting for several types of problems can be automated based on SDN. Testing by large include three main activities: The first one is the testing activities looking for problems based on a reference (e.g. requirements, code, performance, etc). If errors are found a debugging process starts to find location and causes of errors. The last stage after the debugging or the diagnoses process is the treatment process to fix bugs and origin or causes of those bugs. Of course, the process can be evolutionary and those tasks can be repeated very often. For the networks, run time testing is our focus. We want to test at run time network flows, topology, traffic, security, etc. and make sure that everything work as expected. However, reality is not that simple and making a reference of what is good or expected is also not trivial. For networks, policies can be considered the reference that we can match expected results or behavior against. SDN helps defining explicit high level policies and can also help the automatic enforcement and query of those policies. Fault localization or finding the location of the fault is another important task to achieve in SDN based test automation tools. Localization can be in the code (controller or middleboxes), flow tables, firewall rules or policies. Handigol et al 2014 used traceablity of packets’ history as a tool for testing and faults localization. Packet history is the journey packets take from source til destination through switches and including different headers’ modifications.

Ball et al in 2014 presented VeriCon; a formal tool to verify SDN program at compile time based on possible topologies and events. This is a formal approach that models SDN controller as a state machine and is 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 2014 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.

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. This may make applying such methods in real time cases impractical. Traditional testing methodologies and coverage aspects can be applied to the controller program as well as any developed modules on the controller.

# Goals and Approaches

The main goal of this paper is to test SDN based firewall modules. 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 experiments’ objectives will target:

1. 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.
2. 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 ? SDN firewalls can utilize controller-exposed information related to traffic flows. They can focus on how to utilize and optimize the usage of such data in real time. They can be supporting modules for the controller and flow tables’ rules. Quick fast decisions (i.e. coarse grained) are accomplished by the controller and its switches. Thorough and further investigations, if required, (i.e. fine grained) are delegated to the firewall module.
3. 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. Does this mean an integrity problem where a switch is expecting packets or allowing them from a certain host, however, firewall rules reject them?! This however may sound more like or similar to the relation between a host firewall and an enterprise firewall where host rules are contained within the enterprise firewall rules and only rules that do not contradict the enterprise rules will be visible.
4. If you are migrating from traditional firewall to SDN, where should you migrate your rules? 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?
5. 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?
6. 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.
7. 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?!
8. 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?

**First Experiment**

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 fairly 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.

Here are the steps to produce the model.

1. 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 one shows the 8 possible states given the three previously described attributes.

|  |  |  |  |
| --- | --- | --- | --- |
| State | Firewall enabled? | Firewall empty? | Flow table empty ? |
| S1 | TRUE | TRUE | TRUE |
| S2 | TRUE | FALSE | FALSE |
| S3 | TRUE | FALSE | TRUE |
| S4 | TRUE | TRUE | FALSE |
| S5 | FALSE | TRUE | TRUE |
| S6 | FALSE | FALSE | FALSE |
| S7 | FALSE | FALSE | TRUE |
| S8 | FALSE | TRUE | FALSE |

1. 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).
2. Events causes states’ transitions. For example, we can 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 their can be possible errors in those states. In particular, will reading, updating or deleting from an empty table (firewall or flow) will cause a null error?

From this model presentation, we showed that such model 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.

We completed the specifications of all states and their transitions and test cases are generated based on those states’ transitions. 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.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 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\* |

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.

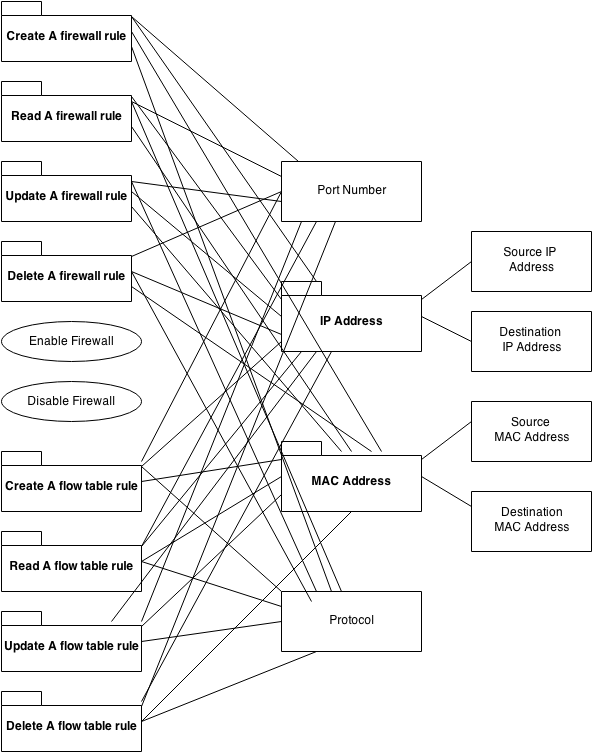
**Second Experiment: Hierarchical state machines**

In order to reduce the state space of possible inputs for our model and hence test cases, we made some assumptions in the first experiment. 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.

From the previous experiment, we had the following observations:

1. 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 maybe able to change or replace an existing rule.
2. 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 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 1 below shows the hierarchical state diagram with three levels.



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.

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.

# Experiments and analysis

We will

# Conclusion

SDN

# References

1. Ehab Al-Shaer, Saeed Al-Haj: FlowChecker: configuration analysis and verification of federated openflow infrastructures. SafeConfig 2010: 37-44
2. E. Al-Shaer and H. Hamed, “Modeling and Management of Firewall Policies,” in IEEE Transactions on Network and Service Management, vol. 1-1, April 2004.
3. E. Al-Shaer, H. Hamed, R. Boutaba and M. Hasan, “Conflict Classification and Analysis of Distributed Firewall Policies,“ in IEEE Journal on Selected Areas in Communications, vol. 23, No. 10, October 2005.
4. A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, “VeriFlow: Verifying network-wide invariants in real time,” ACM SIGCOMM Computer Communication Review, vol. 42, no. 4, pp. 467–472, 2012, 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), April 2013.
5. KUZNIAR, M., PERESINI, P., CANINI, M., VENZANO, D., AND KOSTIC, D. A SOFT Way for OpenFlow Switch Interoperability Testing. In CoNEXT (2012), pp. 265–276.
6. Peter Peresíni, Maciej Kuzniar, Nedeljko Vasic, Marco Canini, Dejan Kostic: OF.CPP: consistent packet processing for openflow. HotSDN 2013: 97-102
7. Canini, Marco; Kostic, Dejan; Rexford, Jennifer; Venzano, Daniele, Automating the Testing of OpenFlow Applications, The 1st International Workshop on Rigorous Protocol Engineering (WRiPE), October, 2011
8. Marco Canini, Daniele Venzano, Peter Peresíni, Dejan Kostic, Jennifer Rexford: A NICE Way to Test OpenFlow Applications. NSDI 2012: 127-140
9. NATARAJAN, S., HUANG, X., AND WOLF, T. Efﬁcient conﬂict detection in ﬂow-based virtualized networks. In ICNC (2012).
10. R. Kloeti, “OpenFlow: A Security Analysis,” April 2013. [Online]. Available: [ftp://yosemite.ee.ethz.ch/pub/students/2012-HS/MA-2012-20 signed.pdf](ftp://yosemite.ee.ethz.ch/pub/students/2012-HS/MA-2012-20%20signed.pdf)
11. Rowan Klöti, Vasileios Kotronis, Paul Smith, OpenFlow: A Security Analysis, Proceedings of the 8th Workshop on Secure Network Protocols (NPSec), part of IEEE ICNP, Göttingen, Germany |October 2013