FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures

Ehab Al-Shaer and Saeed Al-Haj
Department of Software and Information Systems
University of North Carolina at Charlotte
{ealshaer, salhaj}@uncc.edu

ABSTRACT

It is difficult to build a real network to test novel experiments. OpenFlow makes it easier for researchers to run their own experiments by providing a virtual slice and configuration on real networks. Multiple users can share the same network by assigning a different slice for each one. Users are given the responsibility to maintain and use their own slice by writing rules in a FlowTable. Misconfiguration problems can arise when a user writes conflicting rules for single FlowTable or even within a path of multiple OpenFlow switches that need multiple FlowTables to be maintained at the same time.

In this work, we describe a tool, FlowChecker, to identify any intra-switch misconfiguration within a single FlowTable. We also describe the inter-switch or inter-federated inconsistencies in a path of OpenFlow switches across the same or different OpenFlow infrastructures. FlowChecker encodes FlowTables configuration using Binary Decision Diagrams and then uses the model checker technique to model the inter-connected network of OpenFlow switches.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management*

General Terms

Security, Verification

Keywords

OpenFlow, configuration verification, access control, automated analysis, binary decision diagrams

1. INTRODUCTION

OpenFlow is an innovative architecture that provides an open programmable platform for network access control [17]. By separating the data and control plans, users can use the OpenFlow centralized controllers to install filters (match, count and action) in the OpenFlow switches and control the global data processing in the network. Currently, OpenFlow control supports the following actions: forward, drop, encapsulate, encrypt, limit, and classify/enqueue for QoS. The platform is also extensible to support more actions. The controller running new protocols or algorithms might insert, modify, or remove filters in the switches in order to enforce network-wide policies or properties (e.g., guests should access the internet only through a proxy) [17]. Thus, it is assumed that the integrated behavior of the installed filters will globally implement these policies. However, the following conflicts become apparent: (1) the semantic gap between the controller platform (e.g., NOX [13]) and the filter tables in the data processing units, (2) the distribution of access control that supports aggregate flows (wildcards) and many different actions, (3) the ability of sharing one controller by different users, and (4) the ability of using multiple controllers in the same domain. These conflicts together increase the potential of intra-federated (single domain) OpenFlow configuration conflicts. In addition, as two or more OpenFlow infrastructures communicate with each other, potential inter-federated conflicts may appear due to inconsistency in the controller or switch configuration. This may result in invalidation of end-to-end policy enforcement.

Due to these reasons, a correct enforcement of the controller policies might be questionable without the support of formal automated configuration verification tools. This work attempts to address these problems by (1) encoding OpenFlow configuration using Binary Decision Diagrams (BDDs) considering the priority-based matching semantic, various actions, the existence of multiple controllers and multiple users, (2) modeling the global behavior of the OpenFlow network based on FlowTables over single or multiple federated infrastructures in a single state machine, and (3) providing a generic property-based verification interface using BDD-based symbolic model checking and temporal logic. The presented system, called FlowChecker, can be used by administrators/users for (1) verifying the consistency of different switches and controllers across different OpenFlow federated infrastructures, (2) validating the correctness of the configuration synthesis generated by a new implemented protocols, and (3) debugging reachability and security problems. FlowChecker can also be used to conduct "what-if" analysis to study the impact of the new protocols or algorithms on the network by simply changing the state in the FlowTables and then analyzing the effect.

The development of FlowChecker leverages our previous

SafeConfig'10, October 4, 2010, Chicago, Illinois, USA.

research experience and tools such as ConfigChecker [3]. ConfigChecker is a BDD-based symbolic model checker that analyzes the end-to-end access control configuration of all network devices (routers, firewalls, IPSec gateways, NAT and multicasting). We extended ConfigChecker to consider QoS configurations verification and new actions such as limiting. For federated OpenFlow management, we propose a Master Controller (MC) that runs FlowChecker and communicates with federated controllers to identify and resolve inconsistencies or any misconfigurations.

In conclusion, the outcome of this work is a configuration verification tool, or an application run by the administrator or Master Controller to validate, analyze and enforce at run-time OpenFlow end-to-end configuration across multiple federations. It is an important addition to the GENI infrastructure that will give the administrators and users the accountability and analytic power required for large-scale deployment.

2. RELATED WORK

In recent years, a significant amount of research has addressed conflict analysis and configuration modeling. Some research has focused on modeling and configuration for specific devices. For example, firewall modeling and conflict analysis were targeted by ([4], [11], [14], and [20]). Also, there was a considerable amount of work on detecting misconfiguration in routing ([10], [12], [5], and [16]). Other research has been done on creating general model for network devices ([3], [8], and [19]). The concept of OpenFlow switch was introduced in [17] and used in different applications such as [18] and [13]. The work done on OpenFlow switches did not address the problems of conflict analysis and model verification; instead, it showed the basic architecture of OpenFlow model and how it can be used to provide logical separated networks on the physical network. The work done in [18] introduced network virtualization, where by a production network is sliced to multiple virtual networks that run multiple experiments simultaneously, each with their own forwarding decision.

In this work we addressed OpenFlow switches misconfiguration and model verification using BDD-based model checker and (Computational Tree Logic) CTL to write queries to describe properties that we are trying to verify. FlowChecker uses BDDs to model a state machine that encodes the entire behavior of OpenFlow switches in a network.

3. BACKGROUND

In this section we will give a brief background for both OpenFlow switches and ConfigChecker. The background includes the architecture and the formalization used to describe the systems.

3.1 OpenFlow switches

OpenFlow switches are used to partition an entire network to multiple logical networks that can be seen as individual networks [17]. It allows multiple users to run their experiments side by side without affecting each other. Deploying OpenFlow switches in a network will make it programmable network and will provide flexibility to control the traffic in a network. Researchers can take advantage of OpenFlow switches, which provide a rich environment for testing and running new experiments. To prepare a network for such

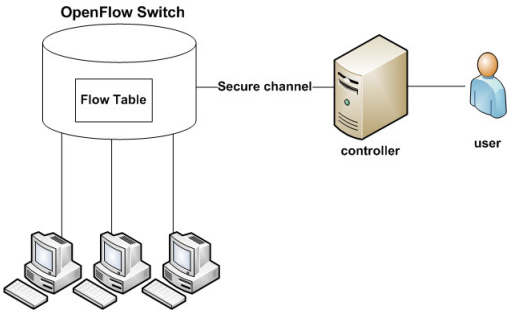

Figure 1: The basic components of an OpenFlow switch

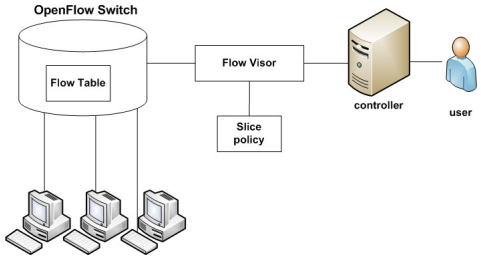

Figure 2: FlowVisor works as an OpenFlow proxy

environments, network administrators need to partition the traffic and mark it as production and research flows. Researchers (we will refer them as users) will be given the right to control their own flows. To create such a network, each user will be given a slice [18] on a network based on his or her needs. Working on a specific slice should indicate working on a separate network, and different slices should not overlap with each other. One of challenges will be to make sure that different slices do not overlap. This task will be among the tasks that FlowChecker will deal with.

An OpenFlow switch has different components: a controller, a secure channel, a FlowTable, and the OpenFlow protocol [17]. Figure 1 shows the main components of an OpenFlow switch. The OpenFlow protocol works as an interface between the controller and the switch.

An OpenFlow switch is controlled by a controller that is connected to the switch through a secure channel [17]. As shown in Figure 1, the controller updates the FlowTable by adding and removing flow entries. Flow entries are the commands used by a user to control traffic. The FlowTable contains flow entries associated with actions that work as commands for the switch to process certain traffic. A list of supported actions includes forward, drop, encapsulate, encrypt, limit, and classify/enqueue for QoS [17]. The ability to add more actions is possible and it depends on the researchers' needs.

More specifically, a flow entry has three parts: Header, Action and Statistics. The header is used to define a specific flow, action is used for processing purpose, and statistics is used for QoS. When a packet comes to an OpenFlow switch, it is matched against flow-entries in the FlowTable. The action will be triggered if a header is matched, then the statistics will be updated [17]. If the packet does not match any flow-entry in the FlowTable, it will be forwarded to the

controller for more processing. In case of multiple matching the action with higher priority will be triggered.

One of the examples illustrating how OpenFlow switch can be used is processing packets instead of flows [17]. The need for processing packets is required for intrusion detection systems which work by testing and inspecting every packet. Using OpenFlow switches, this can be done by forcing all packets to be forwarded to the controller by not adding flow entries in the FlowTable. In this case there would be no matched entry and it will be passed to the controller. Another way to accomplish this would be by routing packets to a programmable switch that can process packets such as a programmable NetFPGA router [17].

The work in [18] introduces the FlowVisor as a network virtualization layer. FlowVisor divides the network to different slices, with each slice having its own set of flows running on switches. The goal of FlowVisor is to support multiple guest controllers on the same OpenFlow. Each controller consists of a user application over NOX running by a single user based on a defined slice. FlowVisor works as an OpenFlow proxy between OpenFlow switches and OpenFlow controllers [18], as shown in Figure 2. All OpenFlow messages between the switch and the controller are sent through the FlowVisor. Multiple controllers can be hosted by the FlowVisor, with each controller requiring its own slice. To make sure that a controller works only on its own slice, FlowVisor keeps a slice policy for each controller. Also, FlowVisor partitions the FlowTable in each switch by marking which flow-entries belong to a specific controller. When a controller tries to insert flow-entry to an OpenFlow switch, the FlowVisor intercepts this message and examines the slice policy for that controller to ensure that the flow entry matches only the traffic related to the controller. In a similar way, OpenFlow switches messages are sent to the controller whose flow space matches the message.

Several challenges may arise when multiple users use multiple OpenFlow switches. One challenge is that multiple controllers are controlling the same switch. Another challenge is that one controller may manage multiple OpenFlow switches. As a result, a misconfiguration may appear in the FlowTables as described in the next sections.

3.2 ConfigChecker: A Tool for Global Verification of Network Configuration

ConfigChecker is a tool which provides comprehensive end-to-end network configuration analysis engine to model all network access control devices (routers, firewalls, NAT, IPSec gateways) and offers a rich, logic-based interface for network configuration analysis.

A network in ConfigChecker is represented and modeled as a state machine. Location and header information for packets are used to define a state in the state machine. Access control devices are modeled using binary decision diagrams (BDDs) [6]. Using state machine representation and BDDs allow the use of symbolic model checking techniques [7] for reachability verification and security properties written in CTL [9]. In ConfigChecker, the use of the extended CTL provides additional operators useful for verifying security properties. Also, an efficient variable ordering is used in the BDD encoding to optimize BDD operations and space requirement. By writing the proper CTL query, ConfigChecker is able to verify reachability and identify security policy violations such as back doors and broken tunnels. These violations may appear because of inconsistencies and misconfigurations between two or more devices in the network. Next, we will present a basic model for ConfigChecker. Case studies and complete model implementation details can be found in [3].

In the basic model, to represent a packet ,we need only information about source and destination information contained in the IP header and the current location of the packet in the network. Therefore, the state of the network can be encoded with the following characteristic function:

$$\sigma : IPs \times Ps \times IPd \times Pd \times loc \rightarrow \{\textbf{true, false}\}$$

$\mathbf{IP}_s$ the 32-bit source IP address

$\mathbf{port}_s$ the 16-bit source port number

$\mathbf{IP}_d$ the 32-bit destination IP address

$\mathbf{port}_d$ the 16-bit destination port number

loc the 32-bit IP address of the device currently processing the packet

Where IPs and IPd are source and destination 32-bit IP address, Ps and Pd are source and destination 16-bit port numbers, and loc the 32-bit IP address of the device currently processing the packet. The function σ encodes the state of the network by evaluating to true whenever the parameters used as input to the function correspond to a packet that is in the network and false otherwise. If the network contains n different packets, then exactly n assignments to the parameters of the function σ will result in true.

The key of modeling a device in a network is by describing the changes that happen to the packet currently located at the device. For example, a firewall may remove a packet from the network by denying action in the access control list or by allowing the packet to move to other device in the network. A router may change the location of the packet without changing any information in the header. A NAT device may change some information in the header as well as the location of the packet. The behavior of a device can be modeled by a list of rules, in which each rule has a condition and an action. The rule condition can be formulated using a Boolean formula over the bits of the state (the parameters of the characteristic function σ). If the packet at the device matches a rule condition, then the appropriate action is triggered. Actions will change a packet by changing packet location and/or some IP header information that reflect the behavior of the specified device. These changes are described by a Boolean formula over the bits of the state. The new values can be constant or variable depending on the values of some bits in the current state. In either case, a transition relation can be formulated as a relation or characteristic function over two copies of the state bits. An assignment to the bits/variables in the transition relation is evaluated to true if the packet described by the first copy of the bits will be transformed into a packet described by the second copy of the bits when it is at the device in question.

We will provide some examples of how real devices can be encoded to illustrate how ConfigChecker works. However, to keep the formulas simple, the examples will contain only 2 bits for the source IP, destination IP, and location IP, and 1 bit for the source port and destination port. Real examples are nearly identical, but with larger fields. The formulas will use the following variables:

s_1, d_1, l_1 The high order bit in the source IP address, the destination IP address, and the location IP address respectively.

s_0, d_0, l_0 The low order bit in the source IP address, the destination IP address, and the location IP address respectively.

s_p, d_p The source port bit and the destination port bit respectively.

$s'_0, s'_1, d'_0, \ldots$ The meaning of these variables is identical to the unprimed versions above, except that these variables represent the values of the bits in the next state.

Modeling Firewalls.. Assume a firewall with IP address 1 has its outbound interface connected to a device with IP address 3. Furthermore, suppose the firewall allows all traffic from IP address 2 to reach any destination as well as all traffic destined to port 1 on the device with IP address 3 coming from any source. All other traffic is blocked. Then all packets that satisfy the following formula will be forwarded to the outgoing interface:

$$(s_1 \wedge \overline{s_0}) \vee (d_1 \wedge d_0 \wedge d_p)$$

Furthermore, in the new state, the packet will look identical, except that its location will be IP address 3. Therefore, the restriction on the new state is:

$$\bigwedge_{i \in \{0,1,p\}} (s'_i \Leftrightarrow s_i) \wedge \bigwedge_{i \in \{0,1,p\}} (d'_i \Leftrightarrow d_i) \wedge l'_1 \wedge l'_0$$

Finally, this transformation only occurs if the packet is currently at the firewall. In other words, the current location of the packet must be IP address 1:

$$\overline{l_1} \wedge l_0$$

Therefore, transition relation for this particular firewall is the conjunction of the three conditions above. In general, the filtering policy of any firewall can be converted into a formula similar to the one in the example. It would have the form, if the current location is equal to this device, and the packet header information allows the firewall to accept the packet, then the new location of the packet is the device connected to the outgoing interface, and no other packet information changes in the new state. This formula encodes the fact that all packets accepted by the firewall are forwarded to the device connected to the outgoing interface, while there is no next state for the packet if it does not accepted by the firewall.

This construction assumes that we can encode a firewall's filtering policy as a Boolean formula over the bits in the packet header information. A simple (but quite large) disjunction of all the minterms representing packets that are accepted demonstrates that this is theoretically possible. In practice, a list of filtering rules is used to configure the behavior of the firewall. This list of rules can be used to directly construct the formula as demonstrated in our previous work [14].

Modeling Routers. Assume a router with IP address 2 sends all packets destined for IP addresses 1 and 0 to IP address 0 (next hop), while all other packets are sent to IP address 3 as a default gateway. The logic for the routing decisions is given by the following formula:

$$(\overline{d}_1 \wedge \overline{l'_1} \wedge \overline{l'_0}) \vee (d_1 \wedge l'_1 \wedge l'_0)$$

Since no packet header information changes we also have the restriction

$$\bigwedge_{i \in \{0,1,p\}} (s'_i \Leftrightarrow s_i) \wedge \bigwedge_{i \in \{0,1,p\}} (d'_i \Leftrightarrow d_i)$$

Finally, this transformation only takes place when the packet is currently at the router; therefore, the following restriction must be satisfied as well:

$$l_1 \wedge \overline{l_0}$$

The transition relation for this particular router is the conjunction of the three conditions above. More models for real devices can be found in [3]

4. FORMAL MODELING OF OPENFLOW CONFIGURATIONS

OpenFlow configurations (or slices in case of FlowVisor) are sequence or rules defined and deployed by controllers in the OpenFlow Switches.

DEFINITION 1. *An* OpenFlow policy, $P = R_1, R_2, \ldots, R_n$, *is a sequence of n filters that determine the appropriate action performed on any incoming packet.*

DEFINITION 2. *A* filter, R_i, *consists of a set of constraints on a set of k filtering fields,* $F = \{f_1, f_2, \ldots, f_k\}$, *together with an action,* a_i, *from the set of all actions, A.*

Each rule can be written in the form:

$$R_i := C_i \rightsquigarrow a_i$$

where C_i is the constraint on the filtering fields that must be satisfied for the action $a_i \in A$ to be triggered. The condition C_i can be represented as a Boolean expression over the filtering field values $fv_1, fv_2, \ldots, fv_k$ as follows:

$$C_i = fv_1 \wedge fv_2 \wedge \cdots \wedge fv_k$$

An example of a field value is IP address, port number, user id, or controller id. The priority-based matching semantic for FlowTable is equivalent to first-matching semantic of firewalls since the first can be reduced to the second by simply ordering the rules based on their priorities. We assume that if two rules have the same priority the first match will take precedence. Therefore, the FlowTable matching semantic fits very well with the if-else Normal Form (INF) and can be encoded using BDDs as follows:

$$P_a = \bigvee_{i \in index(a)} (\neg C_1 \wedge \neg C_2 \ldots \neg C_{i-1} \wedge C_i)$$

$$= \bigvee_{i \in index(a)} \bigwedge_{j=1}^{i-1} \neg C_j \wedge C_i$$

$$\textit{Such that } priority(C_{i+1}) < priority(C_i), \textit{ and}$$
$$index(a) = \{i \mid R_i = C_i \rightsquigarrow a\}$$

Thus the full FlowTable representation (for all actions, users

and controllers) for switch j:

$$P(j) = \bigvee_{\forall n\ =\ a_i\ \in\ Action} P_n.$$

Let us assume that users/administrators and controllers' ids are also encoded as Boolean variables as u and c respectively, then we can represent the FlowTable for filters of action a created by user u on controller c as follows: $P_a^{u,c} = P_a|_{u,c}$ where $|$ is a restrict operation in BDD. Therefore, the representation of all FlowTables in the federated network N that belongs to user u and controller c can be defined as:

$$\phi^{u,c}(N) = \bigvee_{\forall j=switch\in N} \bigvee_{\forall n\ =\ a_i\in\ Actions} P_n^{u,c}(j).$$

Assuming that the default action when a traffic flow does not match any of the filters is to encapsulate and forward this traffic to the controller, then this traffic space that represents these flows will be $\neg P(j)$. Similarly, the slice policy in FlowVisor will have the same encoding assuming that rules with different action can overlap.

5. GENI EXPERIMENTS: OPENFLOW CONFIGURATION VERIFICATION

FlowChecker will be an independent centralized server application that receives queries from OpenFlow applications to verify, analyze or debug OpenFlow configuration. Figure 3 shows an architecture in which FlowChecker can be deployed. These queries could be limited within one domain or across different federated domains. FlowChecker can run on a master controller (spans number of federated OpenFlow infrastructures) and it uses a special protocol to communicate with other controllers and switches.

5.1 FlowChecker Deployment in GENI

There are two ways to deploy FlowChecker in the GENI OpenFlow environment. One way is as a stand-alone service that OpenFlow Controllers will communicate with through a secure channel over SSL (Figure 3). This will involve passing credentials (tokens) to OpenFlow to access the switches. The query will specify topology, users, controllers ids, and name bindings. Controllers send queries and receive responses to/from FlowChecker over this secure channel. If more than one controller exists in the same domain, *i.e.* controllers A and B in domain 3 in Figure 3, then one controller is enough to pass credentials and activate verification. In case of a federated OpenFlow Infrastructure, there are two ways to enable cross-domain verification over multiple OpenFlow networks: (1) all OpenFlow controllers interested in participation in this experiment will subscribe to FlowChecker (called Master Controller) and pass the credential needed to access FlowTables in the target domains, (2) when FlowChecker receives a query from a Controller, it will send a request (query) to the other Controllers to extract the FlowTable entries required for this verification. In the second case, FlowTables can also (optionally) be stored in a DB with role-based access control. There is an obvious trade-off of simplicity vs. security/privacy between the two solutions. However, as a first step we plan the first option in our project. It is worth mentioning that the credentials initiated with each query can be time-based to allow access only for the duration of an experiment.

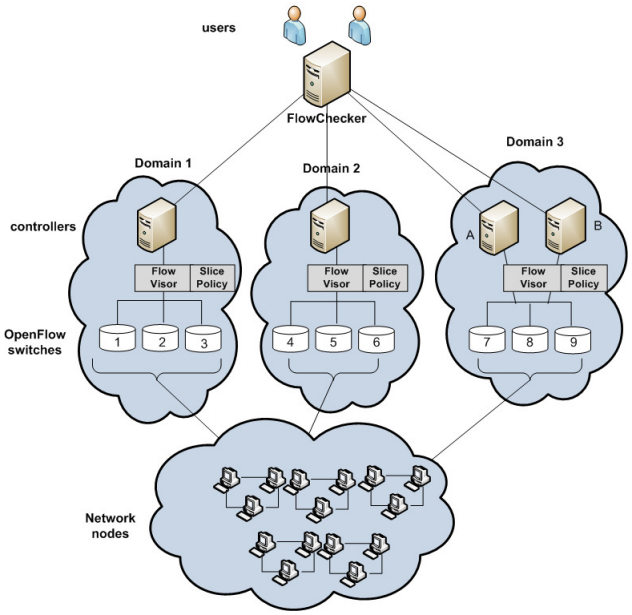

Figure 3: FlowChecker connects multiple domains with multiple controllers and OpenFlow switches

Alternatively, FlowChecker can also be integrated within OpenFlow applications (as a library) running on top of NOX. Figure 4 shows the integration diagram for the FlowChecker, FlowChecker is used by OpenFlow applications to process CTL queries that represent the properties to be investigated. Then, FlowChecker communicates with OpenFlow switches through NOX and FlowVisor. The results of the specified CTL query are sent to OpenFlow application to take the proper action.

Figure 5 describes a possible scenario for validating federated domains. After a user writes his query to check the validity of certain properties using CTL logic, the FlowChecker sends a special message to the controllers that are participating in the validity check asking for extracting FlowTables and sending them back to FlowChecker. When controllers receive the message, they send a request to the OpenFlow switches which are controlled by these controllers to provide their FlowTables. The message sent from a controller to an OpenFlow switch is intercepted by the FlowVisor, then the FlowVisor responds to this message directly to the FlowChecker by sending slices policies for that domain. Unlike FlowTables, controllers have nothing to do with slices policies to keep transparency. FlowTables are sent to the FlowChecker once they are received by a controller. Now, the FlowChecker has the suggested policies "slices policies" and the actual implementation for these policies "FlowTables". A validation is done by comparing suggested slices polices with FlowTables and a report is generated to indicate the conflicts and misconfigurations in the domain and also to reply the CTL query.

5.2 FlowChecker Examples

FlowChecker is a general property-based verifier and analyzer, *i.e.*, any temporal logic formula can be used. In this section, we classify types of verification experiments that can be performed on multiple OpenFlow infrastructures. We

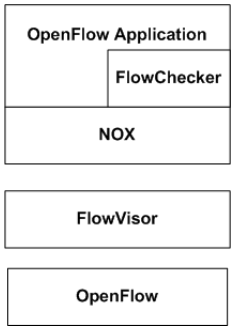

Figure 4: FlowChecker integration

also present examples of important properties to be verified in OpenFlow network.

Intra-Federated OpenFlow Consistency Verification (Intra-Federated Flow Isolation): As OpenFlow allows for wildcard (flow aggregation) of access control configuration, the inter-dependency between different access control installed by the same user over different time period, or multiple users working on the same or different controller might conflict with each other causing errors in the OpenFlow operation such as reachability problems or security violation. General examples of this type of conflicts (e.g., shadowing, correlation) are described in [1, 2, 4]. Examples in OpenFlow environment are as follows (you may refer to Section 4 for notation):

Example (1): "Users flow space should be completely disjoint/isolated", formally:

$$\bigvee_{j=1}^{n-1} \bigwedge_{i=j}^{n-1} P^{i,C} \wedge P^{i+1,C} = FALSE$$

s.t. $Users\ i, j \in Controller(C)$.

Example (2): "All production (non-experiment) flows should be forwarded normally", formally,

$$\Psi \rightarrow \bigvee_{j=1}^{n} P_{a_i}(j) = TRUE.$$

s.t. Ψ is a BDD that represents the production traffic flows and $a_i = normal\_forward$ action. Notice that this does not impose any restriction on non-production traffic, though.

Inter-Federated OpenFlow Consistency Verification (Inter-Federated Flow Isolation): In this part of experiment, we explore inconsistencies between any two entries in FlowTables in the network. For example, "FlowTables of switches of different controllers in the same domain exhibit consistent action for the same overlapping flows". This can be formally defined as follows:

$$\bigvee_{j=2}^{n-1} \bigwedge_{a_i \in Actions} \bigwedge_{j=1}^{n-1} P_{a_i}(j) \wedge \neg P_{a_i}(j+1) = FALSE$$
$$such\ \ that\ \ the\ \ switch\ j \in Controller(C)$$

Intuitively this means that no two FlowTables execute filters of different actions on the same traffic flow. This eliminates both shadowing and spuriousness between switches in the path [14]. It is worth mentioning that this property might only be used with actions that require path consistency/stability such as forward, limit and QoS-classes but not necessary for any general action such as encrypt that can be performed by certain switches.

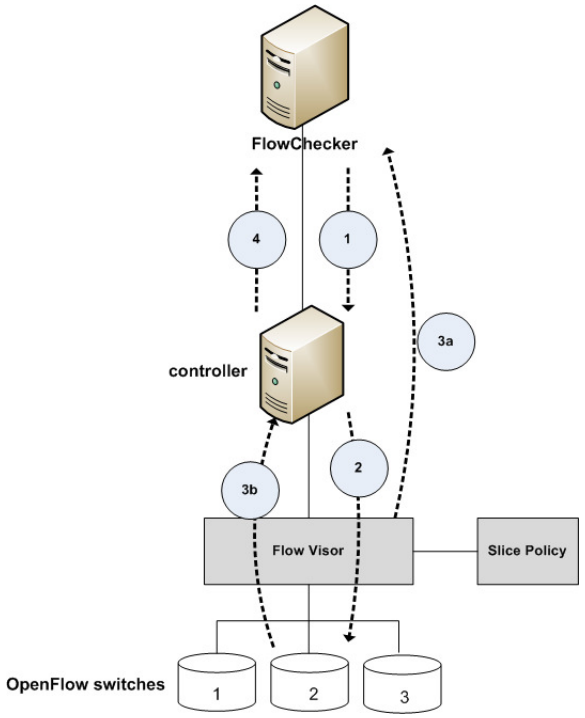

Figure 5: FlowChecker interaction. (1) FlowChecker asks controllers to extract FlowTable entries, (2) a message is sent to the switch through FlowVisor, (3a) FlowVisor intercepts the message and responds to it by sending slice policy to FlowChecker, (3b) OpenFlow switch sends the FlowTable to the controller, and (4) then the controller sends this FlowTable to FlowChecker. Now FlowChecker can verify if the slice policy is enforced correctly in the FlowTable or not.

Property-based Verification for Inter-federated OpenFlow: Using model checker technique like in ConfigChecker (described in Sections 3.2, someone can verify general network properties using temporal logic. Examples in OpenFlow environment are presented below:

Example(1): "if a packet ever encrypted, it will eventually reach the destination as plain text", formally:
$Q = (src = a_1 \wedge dest = a_2 \wedge loc(a_1) \wedge encrypt \rightarrow AF\ decrypt \wedge loc(a_2))$.

where encrypt/decrypt are flags to mark whether a packet is encrypted or decrypted and $loc(a_1)$ indicates the current location of a packet is at address a_1.

Example(2): "guests in the network can only access the Internet through a proxy server" [17], formally:
$(loc(a_1) \wedge src = a_1 \wedge guest(a_1) \wedge dest = a_2 \wedge Internet(a_2))$

$\rightarrow \neg EF(src = a_1 \ \wedge \ dest = a_2 \ \wedge loc(a_2) \ \wedge \ \neg proxy)$

In the previous example, $guest(a_1)$ means that the source address a_1 is used by the guest, $Internet(a_2)$ means that the destination address a_2 is outside the domain, and $proxy$ is a flag to indicate if a packet is sent through a proxy server or not. Literally, if a packet located at an IP address used by a guest and it is destined to an address outside the domain, then the packet should pass through a proxy server.
Example(3): "VoIP phones are not allowed to communicate with laptops" [17], formally:
$(loc(a_1) \wedge src = a_1 \wedge VoIP(a_1) \wedge dest = a_2 \wedge laptop(a_2))$
$\rightarrow \neg EF(src = a_1 \ \wedge \ dest = a_2 \ \wedge loc(a_2))$.

here $VoIP(a_1)$ means that a_1 is assigned to a VoIP phone, similarly, $laptop(a_2)$ indicates that a_2 is a laptop machine.

Flowspace Isolation on FlowVisor: The slice given to a user describes the flowspace which should be isolated from the other users' flowspaces. This means that the flowspace cannot be shared by other users. Assuming S_i and S_j are BDDS for different slices, we can formally verify this as follows: $S_i \cap S_j = \phi, \quad such\ that\ i,\ j \in Users$

FlowVisor Actions: FlowVisor responds to a message m from a controller by allowing it if m matches the user slice, rewriting it if m does not match the user slice but it can be restricted to match it, or dropping it if m does not match and cannot be restricted to match it. This can be formally checked as follows:

$((m \ \rightarrow \ S_i \ = \ TRUE) \ \rightarrow \ allow)$
$((m \ \rightarrow \ S_i \ = \ FALSE) \ \rightarrow \ drop)$
$((m \ \rightarrow \ S_i \ = \ Exp) \ \rightarrow \ rewrite)$,

where m is a message, S_i is the slice policy for user i, and Exp is a Boolean expression.

Interactive Debugging via Counter Examples: For each one of these examples, if the property was not satisfied, a FlowChecker would present a counter example (i.e., misconfiguration) that would invalidate this property in the network. This can be used for interactive debugging by fixing and changing the configuration iteratively until the property is satisfied.

5.3 Soundness and Completeness for OpenFlow Configurations

Using CTL queries we can verify the soundness and completeness of OpenFlow configurations. We do not assume all flows can be targeted from any source IP to and destination IP, even if there is a physical connectivity between IP addresses. The valid connectivity is determined based on the Connectivity Requirement Policy (CRP), which considers the authorization access of all FlowTables users. Let us define $FlowConnect(src, dst)$ as a characterization function for all sets of allowed flows from src to dst that represent CRP.

Definition 1. *A configuration, C, is sound if, for all nodes u and w, all possible paths from u to w are subset of (or implies) authorized paths in FlowConnect(u, w).*

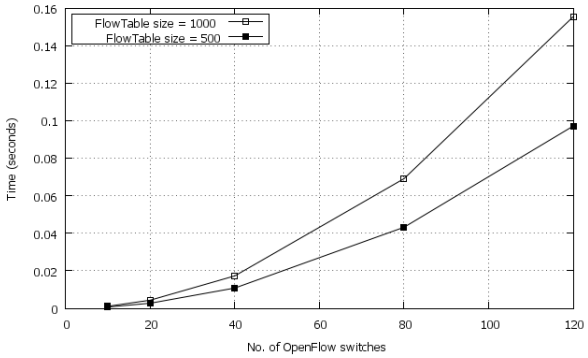

Figure 6: FlowChecker time analysis for intra-federated verification

Intuitively, no node u can communicate with node w using C if this is not allowed in the CRP. Formally, soundness query can be written as follows:
$(loc(a_1) \ \wedge \ src = a_1 \ \wedge \ dest = a_2 \ \wedge \ EF(loc(a_2))$
$\rightarrow FlowConnect(a_1, a_2))$.

Definition 2. *A configuration, C, is complete if, for all nodes u and w, FlowConnect(u, w) contains all possible paths from u to w.*

In other words, if u is allowed to communicate with w in CRP, then there must be a path from u to w to allow this communication. Formally,
$FlowConnect(a_1, a_2) \ \rightarrow$
$[loc(a_1) \ \wedge \ src = a_1 \ \wedge \ dest = a_2 \ \rightarrow \ EF(loc(a_2))]$.

6. EVALUATION

In this section, we will show the performance analysis for FlowChecker with respect to time. In order to test many OpenFlow switches policies, a random FlowTable generator was implemented. The FlowTable generator is controlled by many parameters like FlowTable size and the overlapping between FlowTable entries such as redundancy and subset-superset relationship that can appear by using aggregated entries. FlowChecker is implemented using C/C++ language and BuDDy library [15]. BuDDy library provides a rich environment to use BDDs. As we described is section 4, BDDs are needed to encode FlowTables.

We ran many experiments to measure the impact of FlowTable size on the time performance. Figure 6 shows the time analysis for intra-federated verification for different FlowTable sizes. The quadratic trend appears clearly in Figure 6 as the number of OpenFlow switches increase. In the case of intra-federated verification, we need to make pair-wise comparisons for each pair in the domain, as this will give a complexity of $O(n^2)$ to find if there is an inter-switch conflict between n OpenFlow switches. Our experiments were performed on a 1.8 GHz dual core machine with 2 GB of RAM.

For inter-federated performance analysis, the quadratic trend will appear as in Figure 6 because we still need to do pair-wise comparison for all OpenFlow switches across all domains that we are looking to verify.

7. CONCLUSION

The presented system, called FlowChecker, can be used by applications, administrators, or users to (1) verify the consistency of different switches and controllers across different OpenFlow federated infrastructures, (2) validate the correctness of the FlowTable configuration deployed by new implemented protocols or services, (3) debug reachability and security problems, and (4) analyze the impact of new configurations ("what-if" analysis) without changing the state in the FlowTables.

The time performance analysis shown in section 6 shows that finding misconfigurations in FlowTables can be done at run-time, which gives an important feature to FlowChecker to be used by administrators to resolve any conflict that may appear across multiple OpenFlow switches. The use of CTL language makes it easier to write queries to validate certain properties or to extract statistics to be used for QoS analysis.

8. REFERENCES

[1] E. Al-Shaer and H. Hamed. Firewall policy advisor for anomaly detection and rule editing. In *IEEE/IFIP Integrated Management (IM'2003)*, March 2003. Best Paper Award.

[2] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan. Conflict classification and analysis of distributed firewall policies. *IEEE Journal on Selected Areas in Communications (JSAC)*, 23(10), October 2005. Nominated for Best JSAC Paper Award for year 2005.

[3] E. Al-Shaer, W. Marrero, and A. El-Atawy. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *IEEE International Conference of Network Protocols (ICNP'2009)*, Oct. 2009.

[4] E. S. Al-shaer and H. H. Hamed. Discovery of policy anomalies in distributed firewalls. In *In IEEE INFOCOM '04*, pages 2605–2616, 2004.

[5] R. Alimi, Y. Wang, and Y. R. Yang. Shadow configuration as a network management primitive. *SIGCOMM Comput. Commun. Rev.*, 38(4):111–122, 2008.

[6] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[7] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Journal of Information and Computation*, 98(2):1–33, June 1992.

[8] R. Bush and T. G. Griffin. Integrity for virtual private routed networks. In *IEEE INFOCOM 2003*, volume 2, pages 1467– 1476, 2003.

[9] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. MIT Press, 1990.

[10] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, 2005.

[11] M. Gouda and X. Liu. Firewall design: Consistency, completeness, and compactness. In *The 24^{th} IEEE Int. Conference on Distributed Computing Systems (ICDCS'04)*, March 2004.

[12] T. G. Griffin and G. Wilfong. On the correctness of IBGP configuration. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 17–29, New York, NY, USA, 2002. ACM.

[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, 2008.

[14] H. Hamed, E. Al-Shaer, and W. Marrero. Modeling and verification of ipsec and vpn security policies. In *ICNP '05: Proceedings of the 13TH IEEE International Conference on Network Protocols*, pages 259–278, 2005.

[15] J. Lind-Nielsen. The BuDDy OBDD package. http://www.bdd-portal.org/buddy.html.

[16] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–16, New York, NY, USA, 2002. ACM.

[17] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[18] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Flowvisor: A network virtualization layer. Technical Report OpenFlow Technical Report 2009-1, Deutsche Telekom Inc. R&D Lab, Stanford University, Nicira Networks, October 2009.

[19] G. Xie, J. Z. D. Maltz, H. Zhang, A. G. G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *IEEE INFOCOM 2005*, volume 3, pages 2170– 2183, 2005.

[20] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *IEEE Symposium on Security and Privacy (SSP'06)*, May 2006.