Model Checking Invariant Security Properties in OpenFlow

Sooel Son, *University of Texas at Austin*
Seungwon Shin, *Texas A&M University*
Vinod Yegneswaran, *SRI International*
Phillip Porras, *SRI International*
Guofei Gu, *Texas A&M University*

***Abstract*—The OpenFlow (OF) switching specification represents an innovative and open standard for enabling the dynamic programming of flow control policies in production networks. Unfortunately, thus far researchers have paid little attention to the development of methods for verifying that dynamic flow policies inserted within an OpenFlow network do not violate the network's underlying security policy. We introduce FLOVER, a model checking system which verifies that the aggregate of flow policies instantiated within an OpenFlow network does not violate the network's security policy. We have implemented FLOVER using the Yices SMT solver, which we then integrated into NOX, a popular OpenFlow network controller. FLOVER provides NOX a formal validation of the OpenFlow network's security posture.**

I. INTRODUCTION

OpenFlow (OF), an offshoot of the clean-slate network research initiative [5], [6], [8], is a new and open standard for enabling programmability and dynamic orchestration of enterprise networks. By design, OpenFlow allows network administrators to run custom and third-party applications on a network controller device, which can insert dynamic control flow policies by modifying flow tables at each switch and enable compelling applications such as virtual machine mobility, load balancing, and threat mitigation. However, OpenFlow provides no built-in mechanisms for administrators to vet applications, either statically or at runtime, for compliance with network security policies. We argue that systematic support for automatic security policy enforcement, i.e., the ability to validate that dynamically produced flow rule updates preserve all security properties specified in the network security policy, is vital to the wide adoption of OpenFlow.

We propose a provably correct and automatic method for verifying that a given *non-bypass* property holds with respect to a set of flow rules committed by an OpenFlow controller. Non-bypassability is a basic security property, which is enforced by most firewalls and switches. This property stipulates that packets or flows satisfying specified conditions must adhere to a predefined action, such as forward or drop. Since flow tables of switches in OpenFlow environments can include a large number of prioritized flow entries, manual verification of the non-bypass property on large flow tables across switches is challenging. Furthermore, given the dynamic nature of flow tables, the heterogeneity of vendor implementation in flow table ordering and management, and complex flow rule constructs such as *set* operations that can alter packet content, even automated security evaluation systems are challenged by OpenFlow. Here, we address this challenge of verifying the compliance of a *flow rule set* against an invariant security policy.

Our contributions. The contributions of our work include the following:

- We present a formal approach to prove the conformance of dynamically produced OpenFlow flow rules against non-bypass security properties, including those with *set* and *goto table* actions.
- Using FLOVER we demonstrate how to translate OpenFlow rules and a network security policy into an assertion set, which can then be processed and verified by an SMT solver.
- Our experimental evaluation of FLOVER performance shows that our prototype implementation on Yices can detect coverage and modify violations of up to 200 rules in under 131 ms and 120 ms respectively.

II. RELATED WORK

Our work is informed by prior work on modeling firewall security policies [11], [12]. These studies, however, do not address the dynamic nature of flow rules in software-defined networks. Our work is most similar to FlowChecker, which encodes OF flow tables into Binary Decision Diagrams(BDD) and uses model checking [2] to verify security properties and Veriflow which proposes to slice the OF network into equivalence classes to efficiently check for invariant property violations [10]. However, these systems do not explicitly address intermediate actions such as *set* and *goto* commands. Our work extends beyond these by including formal modeling and verification of *set* and *goto table* action commands. Instead of resolving all intermediate actions when modeling flow rules, our work leverages the capability of a fast and sound SMT solver to resolve all intermediate actions during flow rule verification. NICE uses symbolic execution to verify conformance of OF applications [4]. However, such path exploration approaches do not scale well for large applications.

III. BACKGROUND

OpenFlow facilitates dynamic network orchestration through the separation of control-path and data-path and by enabling dynamic programmability of the control-path. In an OpenFlow network, an OF-controller provides interfaces for creating applications that handle network flow rules dynamically. This greatly simplifies the management of

978-1-4673-3122-7/13/$31.00 ©2013 IEEE

OF-switches as the OF-switch simply enforces flow rules (providing the data path), received from an OF-controller. An OF-switch operates as a generic network switch, except for the OF-protocol extension that enables it to dynamically incorporate flow rules provided by the OF-controller. If an OF-switch receives a network packet for which there is no corresponding flow rule, it reports the packet to an OF-controller, which returns a flow rule that enables the switch to handle subsequent packets from that flow.

Controller applications are in charge of creating flow rules for OF-switches. When an OF-controller receives a flow request from an OF-switch, the OF-controller delivers such requests to a controller application. The application then creates a set of flow rules and sends it to the switch through an encrypted network link. As a controller application may communicate with multiple OF-switches simultaneously, an application can distribute a set of coordinated flow rules across the switches to direct routing or optimize tunneling in a way that may dramatically improves the efficiency of traffic flows, while enabling much greater dynamic control of flows that is hardly achievable using traditional network infrastructure.

A. Non-Bypass Security Property Violations

FLOVER addresses the problem of verifying that the current state of flow rules inserted in a switch's flow table(s) remain consistent with the current network security policy. We decompose the network security policy into a set of assertions, which we refer to as *Non-bypass properties*. Intuitively, a *Non-bypass property* is commonly observable in modern networks as the flow deny and allow rule, which are statically defined to restrict or enable flows throughout and across the network. A *Non-bypass property* specifies whether a certain packet/flow matching a set of conditions should be dropped or forwarded to its destination (we formalize this notion in Section IV-A).

For the purpose of verifying a non-bypass property across an OF-network, it is necessary to verify all flow tables within the OF-network. Table I is a simple instance of our proposed flow rule set model with no overlaps. For simplicity, we denote IP addresses as non-negative integers and provide a formal definition of our OF flow rule set in Section IV. Each entry of the flow rule set consists of conditions over defined fields and a set of actions. We assume that if a given packet matches all conditions of multiple entries, any set of actions corresponding to the matching entry may be performed.

This paper addresses two types of violations of the non-bypass security property that may be present in an OF flow rule set instance. For the first type of violation, we assume that Table I is evaluated against the following non-bypass property: *every packet that goes from source IP [5,6] to destination IP 6 must be dropped*. However, an OF switch using Table I will forward any packet that has 6 for both the source and destination IP address because of the third entry in the first flow table. That is, the final action for every packet satisfying the conditions of a given non-bypass property is inconsistent with the action of the property (and thus some packets can bypass the constraints). We call this kind of misconfiguration a **coverage violation**.

Flow Table	Condition: Field 1 Src IP	Field 2 Src Port	Field 3 Dst IP	Field 4 Dst port	**Action Set**
1	5	[0,19]	6	[0,19]	{ (drop) }
1	5	[0,19]	[7,8]	[0,19]	{ (set $field_1$ 10), (goto 2) }
1	6	[0,19]	[6,8]	[0,19]	{ (forward) }
2	[10,12]	[0,19]	[0,12]	[0,19]	{ (set $field_3$ 6), (forward) }

Table I: Example OpenFlow rule set used to illustrate coverage and modify violations

The second type of violation arises due to the *set* command in an OF flow table. In this example, we define another non-bypass property such that *every packet which goes from source IP address 5 to destination IP address 6 must be dropped*. However, an adversary may tunnel the packet through a series of one or more intermediate receivers such that the transmission chain originates from IP address 5 and ends at destination IP address 6, which is a violation of the non-bypass property. For example, when an adversary sends a malicious packet p whose source and destination IP addresses are 5 and 7 respectively, the packet p is changed into p' that goes from source IP 10 to destination IP 6. Then, the packet p' is forwarded to another switch or the host whose IP address is 6 by the first rule of flow table 2. Thus, packet p' which originates from source IP 5 finally arrives at destination IP 6, which is a clear violation of the specified property. We call this type of violation a **modify violation**.

B. SMT Solving in Yices

Yices is a Satisfiability Modulo Theories (SMT) solver, developed at SRI. The core of Yices implements an efficient SAT solver based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [7]. Yices is provided with an input file modeling given first order logic. If a given model is satisfiable, i.e., there exists at least one instance satisfying all model constraints, Yices outputs such a satisfying example. Otherwise, Yices reports the model to be unsatisfiable. We leverage the soundness of Yices and its ability to efficiently find satisfying examples, to verify flow rule sets.

IV. FLOW VERIFICATION

FLOVER is implemented as an OF application that runs on an OF-controller. Whenever an OF-controller delivers newly created flow rules to an OF-switch, FLOVER verifies a set of specified non-bypass properties against the updated flow rule set. Specifically, if FLOVER receives a flow rule request from an OF-switch, it first creates an update or add command consisting of at least one flow rule [1], and then verifies pre-defined non-bypass properties against the set of created rules

[1]For creating a flow rule, we use a built-in function of NOX controller.

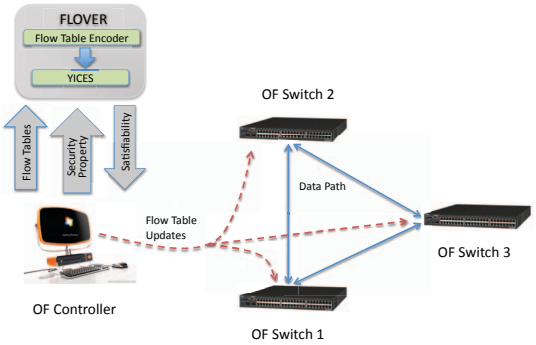

Figure 1: Overview of FLOVER

and previously committed flow rules. Figure 1 illustrates this interaction.

FLOVER internally consists of a flow table encoder and the Yices SMT solver. First, FLOVER encodes the flow rule set, a collection of flow rules and specified non-bypass properties into Yices code. Yices then verifies whether the non-bypass property holds on the encoded model. As long as an OF-controller verifies every committed flow update under its control, FLOVER can prevent an OF network from violating a specified set of non-bypass properties. FLOVER supports two execution modes: *in-line* and *batch* modes. In its *in-line* mode FLOVER performs flow rule validation with each flow rule update while in the *batch* mode, verification is done periodically to improve the controller response time.

We now formalize two key components of our framework: *non-bypass security property* and *flow rule set*. We then present a method of modeling both components in Yices to verify that a flow rule set is free from *coverage* and *modify* violations.

A. Non-bypass Property Representation

A *non-bypass security* property asserts a feature within a given flow rule set. Formally, a *non-bypass security* property is a form of first order logic consisting of universal quantifier, conditions and an action. An action can be *forward* or *drop*. The conditions part of a non-bypass security property is a conjunction of boolean expressions over flow rule set fields. The maximum number of fields is up to 15 [1]. The condition for each field is encoded with a boolean expression specifying a range of non-negative integers because every field consists of a number of bits whose length varies from 3 to 64.

To assert non-bypass properties within a flow rule set against *coverage* and *modify* violations, FLOVER uses two forms of non-bypass security properties, respectively. The formal representation of a non-bypass property denoting that *a flow rule set is free from coverage violations* is as follows:

$$\textit{Non-bypass property}_c = \forall p(\bigwedge_{j=1}^{n} C_j(p) \rightarrow a), a \in \{\textit{forward, drop}\}$$

$$C_j(p) = F_j(p) \in [iLow_j, iHigh_j],\ F_j(p) = j \text{ th field of p}$$

This property denotes that if an initial packet, before modification by an OF-switch, matches the conditions then its final result must be consistent with the action of the property.

The formal representation of a non-bypass property proving that *a flow rule set has no modify violation* is as follows:

$$\textit{Non-bypass property}_m = \forall(p, p')(\bigwedge_{j=1}^{2} C_j(p) \bigwedge_{k=3}^{n} C_k(p') \rightarrow a),$$

$$a \in \{\textit{forward, drop}\}$$

$$(p, p') = \text{ a pair of initial packet } p \text{ and its final packet } p'$$

$$F_1(p) = \text{source IP field of packet p, } F_2(p) = \text{source port field of packet p}$$

This property dictates that if the initial packet p before modification by an OF-switch matches the source IP and port conditions and its final packet p' matches the remaining conditions of the property then, the final result for packet p must be consistent with the action specified by the property.

We assume that the administrator of an OF network has *a priori* knowledge of what non-bypass properties must be enforced within the network. FLOVER checks whether the specified non-bypass properties hold for the rule evaluation sequence imposed by the switch.

B. Flow Rule Set Representation

Table I is an instance of our flow rule set. Each flow entry (rule) has a flow table number indicating where it belongs. Each flow rule has conditions over fields and a set of actions. The condition for each field is a range of non-negative integers, which supports wildcard expression. The action set for each flow rule must contain one of *forward*, *drop* or *goto* as a final action. The action set for a flow rule can have multiple *set* commands that alter the packet matching the condition of the flow entry. *nomatch* represents that a flow table has no matching flow rule. When there is no matching rule for an input packet, an OF-switch submits the packet to the OF controller and requests a flow rule in the default setting[2]. Thus, for packet p, the possible final action set of $FlowRule_i$ in $FlowTable_k$ can be modeled using the following logic forms.

$$FlowRule_i \in FlowTable_k, Cond_i(p) = \bigwedge_{j=0}^{n} F_j(p) \in [iLow_{i,j}, iHigh_{i,j}]$$

$$FlowRule_i(\text{p}) = \{a | a = \ action_{final} \text{ of } FlowRule_i \ s.t \ Cond_i(p) = true \ \}$$

Since the action set of flow entries has a forward, drop or goto action (an intermediate decision delegated to another flow table), the final action of each flow entry is modeled as follows. p' models the *after* state of the initial packet p that could have been changed due to *set* commands.

$$action_{final} \in \{\textit{forward, drop, } FlowTable_l(p')\ \} \ \textit{s.t. } l > k$$

Considering an OF-switch always starts matching packets from the first flow table, all possible final actions matching

[2]An administrator can change the default behavior into *forward* or *drop*

```
1  (define-type states (scalar forward drop nomatch goto) )
2  (define-type packet (record Field1::int Field2::int Field3::int Field4::int ) )
3  (define-type mixstate (record d0::packet d1::states) )
4  (define p::packet)
5  (define a_final::mixstate)
6  (define Cond3::(-> packet bool) (lambda (x::packet) (and
7        (>= (select x Field0) 10) (<= (select x Field0) 12)
8        (>= (select x Field1) 0) (<= (select x Field1) 19)
9        (>= (select x Field2) 0) (<= (select x Field2) 12)
10       (>= (select x Field3) 0) (<= (select x Field3) 19) ) ) )
11 (define Rule3::(-> packet mixstate) (lambda (x::packet)
12       (if (Cond3 x) ;; Condition of the 1st entry in flow table 2
13           ;; forward for true branch
14           (mk-record d0::(update x Field3 6) d1::forward)
15           (mk-record d0::x d1::nomatch) ;; Nomatch for false branch
16 ) ) )
17 (define Cond2::(-> packet bool) (lambda (x::packet) (and
18       (>= (select x Field1) 6) (<= (select x Field1) 6)
19       (>= (select x Field2) 0) (<= (select x Field2) 19)
20       (>= (select x Field3) 6) (<= (select x Field3) 8)
21       (>= (select x Field4) 0) (<= (select x Field4) 19) ) ) )
22 (define Rule2::(-> packet mixstate) (lambda (x::packet)
23       (if (Cond2 x) ;; Condition of the 3rd entry in flow table 1
24           (mk-record d0::x d1::forward) ;; Forward for true branch
25           (mk-record d0::x d1::nomatch) ;; Nomatch for false branch
26 ) ) )
27 (define Cond1::(-> packet bool) (lambda (x::packet) (and
28       (>= (select x Field1) 5) (<= (select x Field1) 5)
29       (>= (select x Field2) 0) (<= (select x Field2) 19)
30       (>= (select x Field3) 7) (<= (select x Field3) 8)
31       (>= (select x Field4) 0) (<= (select x Field4) 19) ) ) )
32 (define Rule1::(-> packet mixstate) (lambda (x::packet)
33       (if (Cond1 x) ;; Condition of the 2nd entry in flow table 1
34           ;; Delegate the decision to another flow table
35           (mk-record d0::(update x Field1 10) d1::goto)
36           (mk-record d0::x d1::nomatch) ;; Nomatch for false branch
37 ) ) )
38 (define Cond0::(-> packet bool) (lambda (x::packet) (and
39       (>= (select x Field1) 5) (<= (select x Field1) 5)
40       (>= (select x Field2) 0) (<= (select x Field2) 19)
41       (>= (select x Field3) 6) (<= (select x Field3) 6)
42       (>= (select x Field4) 0) (<= (select x Field4) 19) ) ) )
43 (define Rule0::(-> packet mixstate) (lambda (x::packet)
44       (if (Cond0 x) ;; Condition of the 1st entry in flow table 1
45           (mk-record d0::x d1::drop) ;; Drop for true branch
46           (mk-record d0::x d1::nomatch) ;; Nomatch for false branch
47 ) ) )
```

Figure 2: Example of transformed Yices code

a given packet p is modeled with the following logic form.

$$FlowTable_1(\text{p}) = \{\ a | a = action_{final} \text{ of } FlowRule_i \text{ s.t. } Cond_i(p) = \text{true}\} \quad (1)$$

$$FlowRule_i \in FlowTable_1, Cond_i(p) = \bigwedge_{j=0}^{n} F_j(p) \in [iLow_{i,j}, iHigh_{i,j}]$$

C. Encoding Flow Rule Sets into Yices

For the purpose of verifying non-bypass properties, FLOVER transforms a given flow rule set into an Yices code specifying all possible pairs of a packet and its final action. Figure 2 shows the transformed Yices input code obtained from the flow rule set in Table I.

Line 1 defines the *state* type, which denotes possible final actions. The *set* action is also covered with an update command that modifies an input *packet* state. Line 2 defines a record type, *packet*, consisting of a number of integer field values. Line 3 defines the *mixstate* record type, which represents an output state of a $Rule_i$ function: *mixstate* consists of *packet* and *states* type values. The *mixstate* record type denotes a pair of a flow rule action and a packet state after applying actions in the flow rule.

FLOVER generates $Rule_i$ and $Cond_i$ functions for each flow entry. The $Cond_i$ function is a conjunction of every condition over fields at the i th flow entry. Therefore, $Cond_i$ gets a packet type input and outputs true if a given packet matches all conditions. Otherwise, it outputs false. The $Cond_i$ function is a representation of the following logic expression: $Cond_i(p) = \bigwedge_{j=0}^{n} F_j(p) \in [iLow_{i,j}, iHigh_{i,j}]$.

```
1  (define cond_bypass_property::(-> packet bool)
2        (lambda (x::packet) (and
3        (>= (select x Field1) 5) (<= (select x Field1) 5)
4        (>= (select x Field2) 0) (<= (select x Field2) 9)
5        (>= (select x Field3) 6) (<= (select x Field3) 6)
6        (>= (select x Field4) 0) (<= (select x Field4) 9)
7        ) ) )
8  (assert (cond_bypass_property) p) )
9  (assert (/= (select a_final d1) nomatch ) )
10 (assert (= (select a_final d1) forward ) ) ;;negation of the action in the
       non-bypass property
```

Figure 3: Non-bypass property for coverage misconfigurations

The $Rule_i$ function models flow entry (rule) matching and modifications to the matched packets. $Rule_i$ receives a *packet* type input and outputs a *mixstate* type, which is a record of the *action* and the *after state* for an input packet. More specifically, $Rule_i$ receives a packet and checks if $Cond_i$ is true. If $Cond_i$ is false, then $Rule_i$ outputs *nomatch*, denoting that $Rule_i$ cannot decide an action for the input packet. When i indicates the last entry of the flow rule set, the action type corresponding to the false branch is also *nomatch* (i.e., the given packet does not match any condition). If the corresponding actions for the *i*th flow entry contain *set* commands, the *set* action is encoded into $(update\ x\ field_c\ val_c)$. Lines 11 and 32 show such an instance of the $Rule_i$ function.

D. Verifying Non-Bypass Property Against Coverage and Modify Violations

To test for *coverage* violations, FLOVER checks if the given non-bypass property is satisfied by a flow rule set. That is, for all packets whose final action can be resolved to *forward* or *drop* by the flow rule set, the action of a given non-bypass property must be consistent with the final action corresponding to every packet which satisfies the conditions of the non-bypass property. Otherwise, there exists at least one packet whose final action is *not* the same as the non-bypass property, but satisfies the conditions of the security property. Figure 3 shows our transformed Yices code denoting the following non-bypass property.

$$\textit{Non-bypass property}_c = \forall p(\bigwedge_{j=1}^{4} C_j(p) \rightarrow drop)$$

$$C_1(p) = F_1(p) \in [5,5], C_2(p) = F_2(p) \in [0,9]$$

$$C_3(p) = F_3(p) \in [6,6], C_4(p) = F_4(p) \in [0,9]$$

FLOVER asserts *packet* p at line 8 and 9, as it considers all packets (i) that satisfy the conditions of a non-bypass security property and (ii) for those that can be resolved to *forward* or *drop*. Since Yices is a counterexample finder, we negate the right hand side of P at line 10, which is *forward*.

Yices takes this code as input data and shows whether the given code is unsatisfiable or satisfiable with a satisfying example. An outcome of **unsat** means that a non-bypass property holds in the given flow rule set because Yices cannot find any packet that (1) satisfies the conditions of the non-bypass property, and (2) its final action is *not* consistent with the action of the non-bypass property. Otherwise, a

```
1  (define cond_wholedomains::(-> packet bool) (lambda (x::packet) (and
2        (>= (select x Field1) 0)   (< (select x Field1) 20)
3        (>= (select x Field2) 0)   (< (select x Field2) 20)
4        (>= (select x Field3) 0)   (< (select x Field3) 20)
5        (>= (select x Field4) 0)   (< (select x Field4) 20)
6  ) ) )
7  (define cond_bypass_property_before::(-> packet bool) (lambda (x::packet) (
         and
8     (>= (select x Field1) 5)  (<= (select x Field1) 5) ;; denote source IP
            field
9     (>= (select x Field2) 0)  (<= (select x Field2) 9) ;; denote source port
            field
10       ) ) )
11 (define cond_bypass_property_after::(-> packet bool) (lambda (x::packet) (and
12    (>= (select x Field3) 6)  (<= (select x Field3) 6) ;; denote dest IP field
13    (>= (select x Field4) 0)  (<= (select x Field4) 9) ;; denote dest port
            field
14       ) ) )
15 (assert (cond_wholedomains p) )
16 (assert (/= (select a_final d1) nomatch ) )
17 (assert (= (select a_final d1) forward ) ) ;;negation of the action in the
       non-bypass property
18 (assert (and
19      (cond_security_property_before p )
20      (cond_security_property_after (select a_final d0) )
21   ) ) )
```

Figure 4: Non-bypass property for modify misconfigurations

non-bypass property does not hold in the flow rule set for a certain packet.

To find a *modify* violation, FLOVER considers not only a final action but also the final *packet* state p' for the entire packet domain. Let us suppose that the non-bypass property for finding modify violations is the following:

$$\textit{Non-bypass property}_m = \forall(p, p')(\bigwedge_{j=1}^{2} C_j(p) \bigwedge_{k=3}^{4} C_k(p') \rightarrow drop)$$

$$C_1(p) = F_1(p) \in [5, 5], C_2(p) = F_2(p) \in [0, 9]$$

$$C_3(p) = F_3(p) \in [6, 6], C_4(p) = F_4(p) \in [0, 9]$$

To find such *modify* violations, we pick source IP and source port as the fields to check for an initial state. The remaining fields are used for checking the final state of a packet. Here, the objective of Yices is to find one counter packet such that (i) its resolved final action is inconsistent with the action of the security property, (ii) its initial *packet* state matches the conditions of the non-bypass property for the source IP and source port fields, and (iii) the final *packet* state matches the condition of the non-bypass property for the remaining fields. This intuition is modeled in Figure 4.

Line 15 asserts a domain range of initial *packet* p to consider all possible packets. Lines 16 and 17 asserts *mixstate* t whose final action is neither drop nor nomatch, which is a negation of an action part of a given non-bypass property. Line 19 asserts that the initial *packet* state of p should match the conditions of the non-bypass property for source IP and source port fields. Line 20 also asserts that *packet* state of a_final should match the condition of the non-bypass property for destination IP and destination port fields. If there exists at least one satisfying example, it denotes that the target flow rule set can create the packet that clearly violates a given non-bypass property at the end. If Yices outputs **unsat**, FLOVER reports that the flow rule set satisfies a given non-bypass property against *modify* violations.

V. EVALUATION

To evaluate FLOVER, we used Mininet [3], a virtual environment to emulate OF-switches, hosts, and OF-controllers.

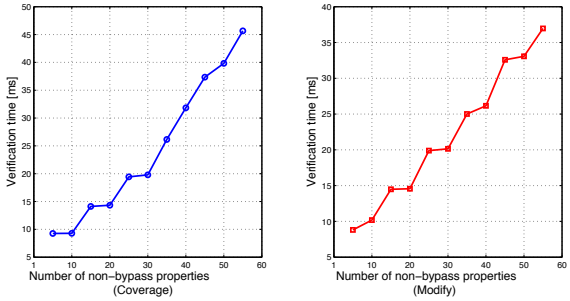

Figure 5: Performance analysis of FLOVER In-Line mode

In Mininet, we built a virtual OpenFlow network consisting of one OF-switch, two hosts connected to the switch, and one OF-controller. We implemented FLOVER as an application running on NOX [13]. The application was implemented in C++ and linked with the Yices library. The experiments were conducted on a Linux workstation with Intel Core i3 CPU and 4 GB memory.

A. In-Line Mode Test

In its in-line mode operation, FLOVER enforces non-bypass properties whenever it receives a flow rule request from an OF-switch. When FLOVER receives such request, FLOVER first creates a set of flow rules and then checks whether such a flow rule set is in conflict with predefined non-bypass properties while recording all created flow rules. If the created rules has no *goto* action then, it is sufficient to verify non-bypass properties against just the newly created rules, rather than all committed rules. Because FLOVER does not commit any rule (1) that can be resolved to *forward* or *drop* and (2) conflicts with non-bypass properties, when an OF-switch selects a flow rule for an input packet, such flow rule is already verified[3]. When the created rule has *goto table* action, we merge a set of created flow rules with recorded flow rules to verify non-bypass properties.

We measure the performance of FLOVER in terms of the flow rule verification time for both coverage and modify violations. Specifically, we investigate the time FLOVER takes to determine whether a created rule set is in conflict with non-bypass properties, as increasing the number of non-bypass properties. Figure 5 shows the evaluation result of applying FLOVER in our test environment. As the number of non-bypass properties increases, the verification time also increases, from around 10 ms (with 5 non-bypass properties) to under 46 ms (with 55 non-bypass properties). Such 10-46 ms penalty is only imposed on the first (SYN or SYN/ACK) packet of every connection.

To put these numbers in perspective, the median inter-arrival time for flows at a datacenter is around 30 ms [9]. Furthermore, our current implementation is unoptimized. We consider two performance optimizations as obvious extensions to enable deployment in large data centers. The

[3]We omit the formal proof due to the page limit

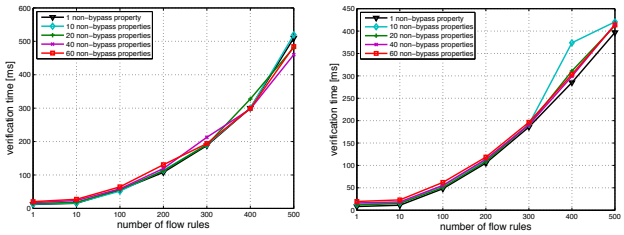

Figure 6: Performance of FLOVER Batch mode on flow rule sets having at least one coverage violation (left) and at least one modify violation (right)

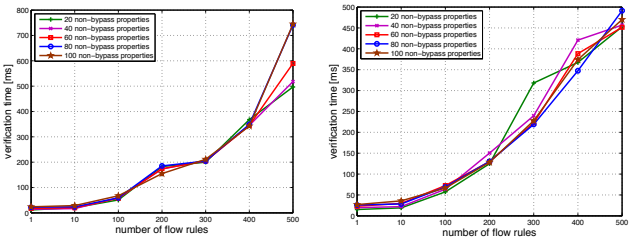

Figure 7: Performance of FLOVER Batch mode on flow rule sets having no coverage violations (left) and no modify violations (right)

first is support for batch-mode operation, that we describe below. The second is enabling a daemon mode operation for Yices to remove the start-up cost associated with each flow verification, which we leave for future work.

B. Batch Mode Test

FLOVER also implements a delayed verification procedure, which aims to reduce connection-setup latency. The batch mode verifies a set of non-bypass properties against the aggregated flow rule set collected since the last verification round through the following three steps: (1) if the number of passed rule sets is larger than a threshold θ, FLOVER performs verification on all updated (but not verified) rule sets, (2) FLOVER reports if (a) collected flow rule set(s) violate(s) non-bypass properties and (3) optionally, FLOVER operates in a passive batch mode where it passes a created rule set to OF-switches without verification but continues recording all flow rule sets.

For the batch mode experiment, we also measure the execution time of FLOVER with increasing θ values (i.e., the number of flow rules which can be enforced without verification). Figure 6 shows the verification times of the batch mode for finding both coverage and modify violations respectively. Figure 7 illustrates the verification times of the batch mode for verifying non-bypass properties when a flow rule set is free from modify and coverage violations.

Detecting coverage and modify violations takes around 15 ms in the best case (i.e., verifying 10 flow rules with 1 security property) and 750 ms in the worst case (i.e., verifying 500 flow rules with 100 security properties). A sweet spot for practical deployment is likely somewhere in between, e.g., 200 rules with 60 security properties and a penalty of 130 ms (which is typically incurred only on the first packet of each connection).

VI. CONCLUSION

We proposed a new way of modeling OpenFlow flow tables with the Yices SMT solver to check for non-bypass property violations. We developed a prototype implementation of our flow verification tool (FLOVER), which translates a given flow table into a series of Yices assertions, and then detects if these assertions are inconsistent with respect to a network security policy. In our evaluation, we find that FLOVER can detect modify and coverage violations of up to 200 rules in less than 120 ms and 131 ms respectively.

REFERENCES

[1] OpenFlow Switch Specification version 1.1.0. 2011. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[2] E. Al-Shaer and S. Al-Haj. Flowchecker: configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and Usable Security Configuration*, 2010.

[3] B. Lantz. Mininet. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.

[4] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *Proceedings of NSDI*, 2012.

[5] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *Proceedings of ACM SIGCOMM*, 2007.

[6] M. Casado, T. Garfinkel, M. Freedman, A. Akella, D. Boneh, N. McKeowon, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Proceedings of Usenix Security Symposium*, 2006.

[7] B. Dutetre and L. Moura. The YICES SMT solver. Technical report, SRI, 2006.

[8] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. In *ACM Computer Communications Review*, 2005.

[9] S. Kandula, S. Sengupta, A. Greenberg, and P. Patel. The nature of datacenter traffic: Measurements and analysis. In *In Proceedings of Usenix/ACM IMC*, 2009.

[10] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proceedings of ACM Sigcomm HotSDN Workshop*, 2012.

[11] A. Liu. Formal verification of firewall policies. In *Proceedings of ICC*, 2008.

[12] A. Liu and M. Gouda. Diverse firewall design. *IEEE Transactions on Parallel and Distributed Systems*, 2008.

[13] NOX. NOX. http://noxrepo.org/wp/.