VeriCon: Towards Verifying Controller Programs in Software-Defined Networks

Thomas Ball and Nikolaj Bjørner
Microsoft Research
{tball,nbjorner}@microsoft.com

Aaron Gember
University of Madison-Wisconsin
agember@cs.wisc.edu

Shachar Itzhaky
Tel Aviv University
shachar@cs.tau.ac.il

Aleksandr Karbyshev
Technische Universität München
aleksandr.karbyshev@in.tum.de

Mooly Sagiv
Tel Aviv University
msagiv@acm.org

Michael Schapira and Asaf Valadarsky
Hebrew University
{schapiram,asaf.valadarsky}@huji.ac.il

Abstract

Software-defined networking (SDN) is a new paradigm for operating and managing computer networks. SDN enables logically-centralized control over network devices through a "controller" software that operates independently from the network hardware, and can be viewed as the network operating system. Network operators can run both inhouse and third-party SDN programs (often called applications) on top of the controller, e.g., to specify routing and access control policies. SDN opens up the possibility of applying formal methods to prove the correctness of computer networks. Indeed, recently much effort has been invested in applying finite state model checking to check that SDN programs behave correctly. However, in general, scaling these methods to large networks is challenging and, moreover, they cannot guarantee the absence of errors.

We present VeriCon, the first system for verifying that an SDN program is correct on *all* admissible topologies and for *all* possible (infinite) sequences of network events. VeriCon either confirms the correctness of the controller program on *all* admissible network topologies or outputs a concrete counterexample. VeriCon uses first-order logic to specify admissible network topologies and desired network-wide invariants, and then implements classical Floyd-Hoare-Dijkstra deductive verification using Z3. Our preliminary experience indicates that VeriCon is able to rapidly verify correctness, or identify bugs, for a large repertoire of simple core SDN programs. VeriCon is compositional, in the sense that it verifies the correctness of execution of any single network event w.r.t. the specified invariant, and can thus scale to handle large programs. To relieve the burden of specifying inductive invariants from the programmer, VeriCon includes a separate procedure for inferring invariants, which is shown to be effective on simple controller programs. We view VeriCon as a first step en route to practical mechanisms for verifying network-wide invariants of SDN programs.

PLDI '14, June 9 – 11 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2594291.2594317

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Invariants; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

General Terms Languages, Verification

Keywords Software-defined networks, Hoare-style verification

1. Introduction

Software defined networking (SDN) is an emerging architecture for operating and managing computer networks, fueled by adoption by major technology companies [11]. SDN allows network administrators to program a (logically) centralized software-based network "controller" that maintains a global view of the network rather than manage tens of thousands of lines of configuration scattered among thousands of network devices (e.g., routers). By centralizing network control and separating it from network hardware, SDN enables network administrators to run (in-house and third-party) programs on top of the controller, e.g., for route computation and to enforce access control policies, without having to wait for features to be embedded in router/switch vendors' proprietary and closed software environments.

The controller configures the network devices, called "switches", through a simple API in the form of a *flow table*: each flow table entry contains a set of packet fields to match, and an action to be taken, such as "send packet out of port X" or "drop packet". When a switch receives a packet it seeks a matching flow entry in its table; if the table has no matching flow entries, the switch sends the packet to the controller, which can then add a flow entry directing the switch how to handle similar packets in the future. The controller maintains a global view of the network by gathering information from the switches (e.g., traffic flow statistics and switch/link failures) and can change the flow table in response to changes in the network and network traffic. The controller effectively plays the role of the network's operating system; programs running on top of it (often called "applications") can use the controller to specify forwarding rules, access control policies, etc., without directly interacting with network hardware or with other SDN programs.

Operating large networks is a complex task that is highly prone to error. This problem is only expected to be exacerbated as network configuration shifts from today's human time scale to event-driven automated configuration with SDN. Guaranteeing network-wide invariants (e.g., enforcing access control) is hence of great importance. SDN opens up the possibilities of applying formal methods to SDN programs to prove the correctness of computer networks. Indeed, much effort has recently been invested in applying finite state model checking to check that SDN programs behave correctly. However, generally speaking, finite-state model checking of SDN programs suffers from two main problems: (1) scaling these methods to large networks is highly nontrivial; and (2) finite-state model checking might identify errors, but cannot guarantee the absence of errors. We present VeriCon, a tool for *provably* verifying network-wide invariants of SDN programs at *compile time*.

VeriCon symbolically reasons about (potentially infinite) network states to verify that network-wide invariants are preserved for *any* sequence of "events" (e.g., the controller receives a packet header from a switch, or a link fails) and on *all* admissible topologies, where invariants and topologies are expressed via first-order logic. VeriCon is *sound* in the sense that if it outputs "no errors" then the preservation of the specified invariants is guaranteed. When verification fails, VeriCon displays a concrete scenario that violates the invariants, in the form of an admissible topology and an event, and it is therefore a useful tool for debugging controller programs. Notice that VeriCon reasons about both the controller and switch correctness.

We designed a simple imperative programming language, called **C**ore SDN (*CSDN*), for writing SDN programs. *CSDN* is but a means to an end; we use it to illustrate that Hoare style verification of SDN programs is feasible in *many* programming languages. VeriCon models the semantics of both *controller events*, such as the receipt of a packet from a switch at the controller, and *switch events*, such as executing a rule entry in a switch's flow table and forwarding an incoming packet to a certain port (or dropping it). Thus, sequences of events in VeriCon capture both controller-to-switch interaction and switch-to-switch interaction, making it possible to reason about SDN programs' behavior for arbitrary sequences of network events (at both the controller and the switches). In fact, we treat the network as just a guarded command program where guards model controller and switch events. This allows us to naturally handle the non-deterministic aspect of networks where switch events can be triggered if the switch includes a matching rule in the flow table, and otherwise controller events occur. Also, these events interact in a non-trivial way. The handler of controller events can change the content of the flow table and then a switch event can cause a violation of an invariant. A premature installation of a flow table rule can break an invariant if the state of the controller is not correctly updated after subsequent switch event. VeriCon can catch these kinds of errors before the program is executed, provided that the programmer can specify the desired invariants.

States in *CSDN* consist of relations representing the network topology and the flow tables of individual switches, and also of auxiliary relations that maintain the controller's information about the global network state. The state — of the controller and switches — is updated by adding or removing tuples from relations (e.g., the switches' flow tables). *CSDN* was designed in the spirit of the OpenFlow standard [1].

VeriCon takes as input a *CSDN* program as well as formulas in first-order logic that specify constraints on the network topology and desired safety properties (network-wide invariants). VeriCon employs the standard weakest preconditions [5] in order to generate verification condition (VC). The VC is a first order formula which holds if and only if: (i) the initial network state satisfies the invariant and (ii) the invariant is inductive, i.e., the execution of arbitrary controller and switch events maintain the invariant. The semantics of the controller events is taken from the *CSDN* program. The semantics of the switch is dictated by the OpenFlow standard [1]. The soundness of completeness of such approaches is completely standard (e.g., see [7]).

An automated theorem prover (Z3 [4]) checks the verification condition, to either verify the correctness of the controller program or to generate a concrete network topology and event that violate the safety properties. It is well known that tools like Z3 are not guaranteed to terminate in general. However, we notice that the generated VCs for many network protocols belong to a class of formulas that are easy to verify by SAT solvers.

We show that VeriConis practical for a large repertoire of controller programs, ranging from simple but well-studied examples, e.g., MAC-learning switches and firewalls, to simplified versions of recently proposed SDN-based systems: Resonance [19] for dynamic access control and Stratos [8] for orchestrating the steering of traffic through middlebox sequences. VeriCon is compositional, in the sense that it checks the correctness of each network event independently against the specified invariants, and so it can scale to handle complex systems well beyond other approaches, e.g., finite state model checking, given appropriate inductive invariants.

Organization Section 2 provides an informal overview of VeriCon. Section 3 shows how admissible topologies and network-wide invariants in SDNs can be formulated in first-order logic. Section 4 discusses how to generate verification conditions for controller programs. To simplify exposition, we limit the discussion in this section to the small core language *CSDN*. Section 5 presents the implementation of VeriCon and our experience with verifying SDN controllers. Section 6 reviews related work.

2. Overview

This section provides an overview of the VeriCon system.

2.1 Programming Controllers

Fig. 1 shows a simple SDN program implementing a stateful firewall, inspired by [12]. The idea here is that end-hosts in the corporate domain can send traffic to the outside world but, for security reasons, traffic from an end-host outside the domain can only enter the domain if that end-host has previously received traffic from some end-host within the domain. This is realized as follows. Two types of hosts are connected to a switch: (i) trusted hosts (within the organization) via port 1; and (ii) untrusted hosts (outside the organization) via port 2. Packets from trusted hosts are always forwarded to untrusted hosts. Packets from untrusted hosts are forwarded to trusted hosts only if the source host has previously received a packet from a trusted host. The auxiliary relation `tr` records the trusted hosts for each switch. We use bold font to denote OpenFlow commands. The program is actually executed in an infinite loop with two type of events: **pktIn** events that are annotated with commands, and **pktFlow** events whose semantics is determined by the current content of the flow table.

Fig. 2 shows a typical topology and Table 1 depicts a particular scenario of network events and their effects.

2.2 Correctness Checking

VeriCon receives three inputs: (i) a SDN program, (ii) a first-order formula describing constraints on the topology, and (iii) a correctness condition expressed as invariants in first-order logic. It verifies that for every event executed in an arbitrary topology the program satisfies the required invariants by means of a theorem prover. In the firewall example, the requirement is that for every packet sent from an untrusted host to a trusted host there exists a packet sent to that untrusted host from some trusted host. This is

rel $\texttt{tr}\,(SW, HO) = \{\}$	// an initially empty relation recording trusted hosts, two arguments: switch and host
@while new events occur do $\{$	
pktIn$(s, src \rightarrow dst, prt(1)) \Rightarrow$	// a packet from a trusted hosts appeared on the switch s without a forwarding rule
s.**forward**$(src \rightarrow dst, prt(1) \rightarrow prt(2))$	// forward the packet to untrusted hosts
$\texttt{tr}.insert(s, dst)$	// insert the target of the packet into trusted controller memory
s.**install**$(src \rightarrow dst, prt(1) \rightarrow prt(2))$	// insert a per-flow rule to forward future packets
@pktFlow$(s, src \rightarrow dst, prt(1)) \Rightarrow$	// a packet from trusted hosts with a forwarding rule
@forward according to the table	
pktIn$(s, src \rightarrow dst, prt(2)) \Rightarrow$	// a packet from an untrusted hosts appeared on the switch s
if $\texttt{tr}(s, src)$ **then**	// check if src is trusted on s
s.**forward**$(src \rightarrow dst, prt(2) \rightarrow prt(1))$	// if yes, forward the packet to trusted hosts
s.**install**$(src \rightarrow dst, prt(2) \rightarrow prt(1))$	// and insert a per-flow rule to forward future packets
@pktFlow$(s, src \rightarrow dst, prt(2)) \Rightarrow$	// a packet from untrusted hosts with a forwarding rule
@forward according to the table	
@$\}$	

Figure 1. A simple stateful firewall monitoring the traffic from untrusted hosts connected to $prt(2)$ to trusted hosts connected to $prt(1)$. Statements beginning with @ are added for expository purposes.

event	**action**	**new-rule**	**new-trusted**
pktIn$(s, c \rightarrow b, prt(2))$	none	none	none
pktIn$(s, a \rightarrow c, prt(1))$	$s.forward(a \rightarrow c, prt(1) \rightarrow prt(2))$	$\langle s, a \rightarrow c, prt(1) \rightarrow prt(2) \rangle$	$\langle s, c \rangle$
pktIn$(s, c \rightarrow b, prt(2))$	$s.forward(c \rightarrow b, prt(2) \rightarrow prt(1))$	$\langle s, c \rightarrow b, prt(2) \rightarrow prt(1) \rangle$	none
pktFlow$(s, c \rightarrow b, prt(2) \rightarrow prt(1))$	$s.forward(c \rightarrow b, prt(2) \rightarrow prt(1))$	none	none

Table 1. A scenario of a packet transmissions for the SDN program shown in Fig. 1 using the topology shown in Fig. 2.

captured by the following *safety invariant*

$$\mathcal{I}_1 \stackrel{\text{def}}{=} \begin{array}{l} S.sent(Src \rightarrow Dst, prt(2) \rightarrow prt(1)) \Rightarrow \exists Src' \in HO : \\ S.sent(Src' \rightarrow Src, prt(1) \rightarrow prt(2)) \end{array}$$

To improve readability, free variables in formulas (such as S and Src, Dst above) are implicitly universally quantified. S denotes an arbitrary switch; Src, Dst, and Src' denote arbitrary hosts. Finally, $S.sent(Src \rightarrow Dst, I \rightarrow O)$ denotes the fact that a packet from source Src to destination Dst was forwarded from input port I to output port O of the switch S.

2.2.1 Unbounded Symbolic Topologies

VeriCon, unlike finite-state model checking (e.g., [3]), verifies that the invariants hold under any admissible network topology of any size. By default, the admissible topologies are all the possible network graphs, but the programmer may choose to force a certain class of admissible topologies; e.g., one can enforce a star shape by requiring that there exists a switch to which all the other switches are connected by a link:

$$\begin{array}{l} \exists S \in SW : \forall S_1, S_2 \in SW : S_1 \neq S_2 \Rightarrow \\ \quad \big(\exists I_1, I_2 \in PR : link(S_1, I_1, I_2, S_2)\big) \Leftrightarrow S_1 = S \vee S_2 = S \end{array}$$

Here $link(S_1, I_1, I_2, S_2)$ denotes the fact that port I_1 of switch S_1 is connected to port I_2 of switch S_2.

VeriCon first checks that the safety invariant and topology constraints are consistent. In the example, $\mathcal{I}_1$ can be trivially satisfied by an empty topology; conjoined with the star constraint, it can be satisfied by a topology of size 1. Note that we can conjoin different requirements and let the theorem prover infer a common topology.

VeriCon then continues to check that the invariants are preserved by executions of arbitrary switch and controller events on an arbitrary network subject to the topology constraints. VeriCon provides a compositional way to check complex code under arbitrary event sequences. However, this requires the specification of inductive invariants[1]. In the firewall example above, $\mathcal{I}_1$, though correct, is not inductive; VeriCon displays a counterexample for each event that violates the invariant. For example, VeriCon generated the counterexample shown in Fig. 3 for the switch event (**pktFlow**). It describes a configuration in which the switch's flow table is unconstrained and permits forwarding from untrusted hosts to trusted ones. In this case, it is not a bug in the code—this situation cannot occur at runtime, but $\mathcal{I}_1$ is not strong enough to exclude it. Another counterexample, which VeriCon produces (Fig. 4), corresponds to situations in which the trusted relation contains superfluous entries when entering a controller event (**pktIn**). We strengthen $\mathcal{I}_1$ to be inductive by adding the following two safety invariants that exclude the above situations:

$$\mathcal{I}_2 \stackrel{\text{def}}{=} \begin{array}{l} S.ft(Src \rightarrow Dst, prt(2) \rightarrow prt(1)) \Rightarrow \\ \exists Src' \in HO : S.sent(Src' \rightarrow Src, prt(1) \rightarrow prt(2)) \end{array}$$

$$\mathcal{I}_3 \stackrel{\text{def}}{=} \begin{array}{l} \texttt{tr}(S, H) \Rightarrow \\ \exists Src \in HO : S.sent(Src \rightarrow H, prt(1) \rightarrow prt(2)) \end{array}$$

Safety invariant $\mathcal{I}_2$ uses the relation symbol ft to refer to a potentially infinite relation describing the flow tables of the individual switches. It states that flow table entries only contain forwarding rules from trusted hosts. $\mathcal{I}_3$ states that the controller data structure $\texttt{tr}$ records the correct hosts. These kind of invariants are common in many SDN programs.

VeriCon reports that $\mathcal{I}_1 \wedge \mathcal{I}_2 \wedge \mathcal{I}_3$ is inductive, and the controller program is verified. Notice that when VeriCon proves that an invariant holds, that invariant is guaranteed to hold for an arbitrary sequence of network events. For example, $\mathcal{I}_1$ also guarantees that packets forwarded by the switch are sent by certified hosts.

Bug Finding VeriCon can identify subtle bugs in an SDN program. When VeriCon is applied to an incorrect SDN program, it produces a concrete counterexample in a readable manner. VeriCon shows the event that violates the safety invariant and the error configuration,

[1] An invariant is *inductive* if it is preserved by executions starting from an arbitrary state satisfying the invariant.

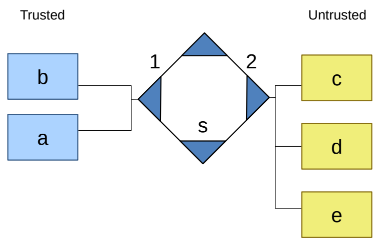
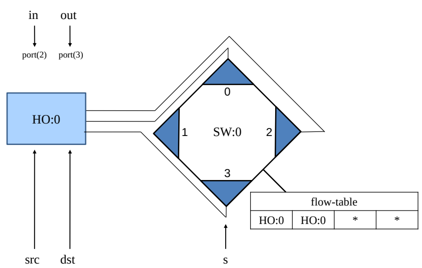
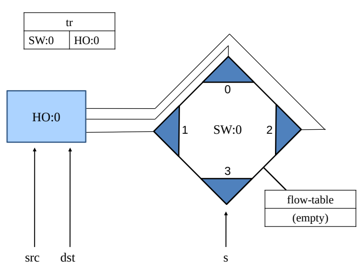


Figure 2. A sample topology for a firewall monitoring the traffic from untrusted to trusted hosts. s is a switch, a through e are hosts: a and b are trusted, c, d, and e are untrusted.



pktFlow$(s, src \to dst, in \to out)$

Figure 3. A flow-table and a packet $src \to dst$ that cause $\mathcal{I}_1$ to be violated when a packet flow event is executed by the switch s, as the packet arrives at input port in.



pktIn$(s, src \to dst, prt(2))$

Figure 4. A trusted-relation (`tr`) and a packet $src \to dst$ that cause $\mathcal{I}_1$ to be violated when a controller event is executed as the packet arrives at port 2 of switch s and is forwarded to the controller.

where the latter includes the concrete topology and the symbolic conditions on packets.

A counter-model generally indicates either an overly-permissive invariant or a real bug in the program. In Section 5, we show an example in which VeriCon detects a real bug in an SDN program.

On-line Topology Changes VeriCon handles potential topology changes by assuming that, between events, the topology can transform into another topology satisfying the specified topology invariants. This means that if the program is proven correct, then this proof is robust w.r.t. links going up or down during execution.

2.2.2 Inferring Inductive Invariants

In general, writing inductive invariants by hand is very tricky since the programmer needs to specify the set of states after an arbitrary sequence of events. Therefore, VeriCon includes a separate simple utility for inferring invariants using iterated weakest preconditions [5]. The main idea is to strengthen the goal invariants by arbitrary executions of controller and switch events. We start with the goal invariant and perform backward analysis on the event handler code (including flow events). For example, $\mathcal{I}_2$ is the weakest precondition which guarantees that $\mathcal{I}_1$ holds after executing the semantics of the switch event **pktFlow**$(s, src \to dst, prt(2))$. Also, $\mathcal{I}_3$ is the weakest precondition which guarantees that $\mathcal{I}_1$ holds after executing the controller command associated with the **pktIn**$(s, src \to dst, prt(2))$ event. Therefore, VeriCon can infer the inductive invariant from the $\mathcal{I}_1$ specification. As shown in Section 5, inductive invariants can often be inferred for simple SDN programs using few strengthening iterations. However, for more complicated programs it may be necessary to apply more advanced invariant inference techniques.

2.3 Limitations

Our current verification methodology is limited in two ways:

- We focus on safety properties. We leave the verification of liveness properties, e.g., that packets must eventually reach their destinations, for future research.
- We assume that events are executed atomically, ignoring out-of-order rule installations. Consistently updating a software-defined network is an important challenge in SDN (see [22]). We plan to address this issue in the future by considering interleavings of rule installations without barriers.

3. Symbolically Modeling SDN Properties with First-Order Logic

We now show that many interesting properties of software-defined networks can be naturally expressed in first-order logic. We use relations to model the network topology, the flow tables of the switches, the SDN program's internal state, as well as histories of transmitted packets. Each of the relations is potentially infinite. *CSDN* commands manipulate relations by inserting and removing tuples from relations. Queries over network states are defined using first-order formulas.

3.1 Predefined Relations

Table 2 describes the predefined relations supported by VeriCon. In addition, the programmer also can define her own relation symbols. The relations $link(S, O, H)$, $link(S_1, I_1, I_2, S_2)$, $path(S, O, H)$, and $path(S_1, I_1, I_2, S_2)$ represent the physical network topology. SDN programs generally do not explicitly manipulate these relations, other than populating them based on link-level discovery protocol (LLDP) information reported by SDN switches.

For clarity, a packet header is represented as a pair $Src \to Dst$. Our implementation supports different packet header fields as functions $PK \to Values$. The most interesting built-in relation involving packet headers is $S.ft(Src \to Dst, I \to O)$, which represents the switches' forwarding tables. This relation denotes the semantics of the switch and not its concrete storage. For example, the SDN program command s.**install**$(h \to *, i \to o)$ updates switch s to include a general forwarding rule for all packets from host h coming from ingress port i to be forwarded to port o. This is implemented in OpenFlow by installing a general matching rule based on the source of the packets. The symbolic effect on the *ft* relation is to add *all* the tuples $S.ft(Src \to Dst, I \to O)$ where $S = s$, $Src = h$, $I = i$, and $O = o$ (Dst is unconstrained).

Relation	Attributes	Intended Meaning
$\mathit{link}(S, O, H)$	$S \in SW, O \in PR, H \in HO$	Host H is directly connected to switch S via port O
$\mathit{link}(S_1, I_1, I_2, S_2)$	$S_1 \in SW, I_1, I_2 \in PR, S_2 \in SW$	Port I_1 of switch S_1 is directly connected to port I_2 of switch S_2
$\mathit{path}(S, O, H)$	$S \in SW, O \in PR, H \in HO$	There is a path from S via port O to a host H
$\mathit{path}(S_1, I_1, I_2, S_2)$	$S_1, S_2 \in SW, I_1, I_2 \in PR$	There is a path from port I_1 of switch S_1 to port I_2 of switch S_2
$S.\mathit{ft}(\mathit{Src} \to \mathit{Dst}, I \to O)$	$S \in SW, \mathit{Src}, \mathit{Dst} \in HO, I, O \in PR$	Switch S has a rule to forward packets $\mathit{Src} \to \mathit{Dst}$ arriving from port I to port O
$S.\mathit{sent}(\mathit{Src} \to \mathit{Dst}, I \to O)$	$S \in SW, \mathit{Src}, \mathit{Dst} \in HO, I, O \in PR$	Packet $\mathit{Src} \to \mathit{Dst}$ arrived at ingress I is forwarded to egress O
$\mathit{rcv}^{\mathit{this}}(S, \mathit{Src} \to \mathit{Dst}, I)$	$S \in SW, \mathit{Src}, \mathit{Dst} \in HO, I \in PR$	Packet $\mathit{Src} \to \mathit{Dst}$ is received at ingress port I

Table 2. Built-in relations describing network states.

The effect is handled in the generated verification condition using a first-order formula expressing the weakest precondition of this command by substituting the *ft* relation symbol. Sending to egress port $O = \texttt{null}$ models dropping packets.

The *history* relation $S.\mathit{sent}(p, I \to O)$ records packets whose header is p forwarded by a switch S from ingress port I to egress port O. This can happen either by executing a forwarding rule at a switch or by sending a packet to the controller which then instructs the switch to forward the packet from I to O. This relation records the history of forwarding and is used for reasoning. For example, it occurs in the invariant $\mathcal{I}_1$ defined in eq (2.2).

Note that "$S.r(\vec{x})$" is just syntactic sugar for the first-order formula "$r(S, \vec{x})$", used to enhance readability.

The predicate $\mathit{rcv}^{\mathit{this}}(S, P, I)$ allows assertions to refer to the packet currently being handled by the switch or the controller code. The examples of such assertions include *transition invariants*, defined below. For a controller event $\mathbf{pktIn}(s, p, i)$ (respectively, for a switch event $\mathbf{pktFlow}(s, p, i \to o)$), $\mathit{rcv}^{\mathit{this}}(S, P, I)$ holds if and only if $S = s$, $P = p$ and $I = i$.

3.2 Invariants

Fig. 5 shows the syntax of standard (typed) first-order formulas which are used to describe invariants of SDN programs and topology constraints. In the atomic formulas, *Rid* is either a predefined or a user-defined relation. For readability, we write atomic formulas $r(S, \langle \mathit{Src}, \mathit{Dst} \rangle, I, O)$ as $S.r(\mathit{Src} \to \mathit{Dst}, I \to O)$, where S denotes an arbitrary switch, *Src* denotes a source host, *Dst* denotes a destination host, I is an input port, and O is an output port.

VeriCon supports three kinds of invariants:

(i) The *topo* invariants define the admissible topologies. These are assumed to hold in the initial state. VeriCon checks that these invariants are consistent with *safety* and *trans* invariants and that together they form an inductive invariant that is preserved under the execution of switch and controller events.

(ii) The *safety* invariants are supposed to hold at the initial state and be preserved for any execution of switch and controller event sequence.

(iii) The *trans* invariants are checked after the execution of every event. They describe the properties of transitions caused by the event in a similar way to postconditions in procedures.

VeriCon simplifies the verification task by assuming that both switch and controller events are executed atomically. In particular, when the controller executes a sequence of commands the invariant is checked before and after the whole sequence and not after individual commands. It is straightforward to check that the invariant holds after every command. However, this will lead to many false alarms as the code usually assumes atomicity.

F	$::=$	**True**	`true`
	$\mid$	$\mathit{Trm} = \mathit{Trm}$	`equality`
	$\mid$	$\mathit{Rid}(\mathit{Trm}^*)$	`atomic formulas`
	$\mid$	$\forall \alpha \colon \mathit{Tid}.F$	`universal quantification`
	$\mid$	$\exists \alpha \colon \mathit{Tid}.F$	`existential quantification`
	$\mid$	$\neg F$	`negation`
	$\mid$	$F \wedge F$	`conjunction`
	$\mid$	$F \vee F$	`disjunction`
Trm	$::=$	α	`logical variable`
	$\mid$	$\mathit{Fid}(\mathit{Trm})$	`uninterpreted functions`

Figure 5. A syntax of first-order formulas.

T	Formula	Intended Meaning
T_1	$\neg \mathit{link}(S, I_1, I_2, S)$	*no self-loops*
T_2	$\mathit{link}(S_1, I_1, I_2, S_2) \Rightarrow \mathit{link}(S_2, I_2, I_1, S_1)$	*symmetry of links*
T_3	$\mathit{rcv}^{\mathit{this}}(S, P, I) \Rightarrow \mathit{path}(S, I, P.\mathit{src})$	*packets arrive from reachable hosts*
T_4	$\mathit{prt}(M) = \mathit{prt}(N) \Rightarrow M = N$	*injective ports*

Table 3. Examples of interesting topology invariants for SDNs.

3.2.1 Topology Invariants

First-order logic can express many topology invariants naturally, as shown in Table 3. These invariants are crucial for VeriCon to precisely reason about networks. For example, the invariant T_3 asserts that packets cannot be received from disconnected hosts. Without this invariant the theorem prover may issue false messages. Notice that some of these invariants may not be relevant or need not hold for some topologies. The current implementation provides a library of invariants which can optionally be included in the controller code.

3.2.2 Safety Invariants

VeriCon permits *safety invariants* that define the required consistency of network-wide states. VeriCon checks that every event preserves all the topology and safety invariants. In the firewall example, we check that $\mathcal{I}_1 \wedge \mathcal{I}_2 \wedge \mathcal{I}_3$ is preserved by the execution of flow and controller events.

3.2.3 Transition Invariants

VeriCon also permits the programmer to define *transition invariants*, which describe the effect of executing event handlers. VeriCon checks that all the transition invariants are satisfied after the execution of every switch and controller event.

```
rel connected (SW, PR, HO) = {}  // new relation with 3 args
pktIn(s, src → dst, i) ⇒          // packet from src into dst
   var o : PR                     // var. for egress ports
   connected.insert(s, i, src)    // learn a new connection
   if connected(s, o, dst) then   // if dest. is already known
      s.forward(src → dst, i → o) // forward the packet
      s.install(src → dst, i → o) // install a new rule
   else s.flood(src → dst, i)     // flood if dest. is unknown
```

Figure 6. A simple learning switch controller code

A simple example of a transition invariant is the absence of "black holes", i.e., packets are never dropped. This is expressed as

$$rcv^{this}(S, Src \to Dst, I) \Rightarrow \exists O \in PR : S.sent(Src \to Dst, I \to O)$$

The latter invariant does not hold in the firewall example since it drops packets from untrusted hosts. However, the invariant does hold for the simple learning switch shown in Fig. 6. This SDN program learns connected hosts as soon as new packets from them appear. When a packet arrives, it is forwarded to the port to which the destination host is connected or, if this port is unknown, the packet is sent via **flood** to all the ports excluding the input port.

Interestingly, with VeriCon one can even state (and prove) that the learning switch SDN program correctly forwards packets. One way to express this using the transition invariant

$$\begin{aligned} trans: \quad & rcv^{this}(S, Src \to Dst, I) \land O \neq I \land path(S, O, Dst) \Rightarrow \\ & S.sent(Src \to Dst, I \to O) \end{aligned}$$

However, VeriCon reports that this invariant is violated in network topologies where multiple ports can be used to reach hosts. We can restrict the topology by adding an extra topology invariant

$$topo: \quad path(S, I_1, H) \land path(S, I_2, H) \Rightarrow I_1 = I_2$$

Alternatively, it is possible to state the correctness of the learning switch in networks with multiple outgoing ports using the formula L_4 shown in Table 4. Here, O_1 quantifies over the existence of path and O_2 denotes the port chosen by the flood command or via learning. This invariant is a bit complicated since it deals with situations in which there are multiple ports leading to the destination host. In fact, it is possible to show that the learning switch also correctly learns the connections using the invariants shown in Table 4.

4. Verifying Controller Programs

This section defines the *CSDN* language, a simple imperative language for writing SDN programs, with an eye towards making verification tractable. The only data structures in *CSDN* are relations, which model both the internal state of the controller and the flow tables. *CSDN* commands can query and update the relations. As shown later, updates to relations are expressible using Boolean operations, a fact which greatly simplifies the verification task. As a result, Z3 is able to precisely reason about many interesting properties, as discussed in Section 4.3.

4.1 The *CSDN* Language

Fig. 7 describes the abstract syntax of *CSDN*. The SDN program first declares external relations, initializes these relations, and specifies constraints on the network topology. Unless the programmer restricts the set of admissible topologies (using the keyword **topo**), all topologies are admissible. The programmer can initialize user-defined relations and specify controller-specific invariants. The built-in relations $sent$, $recv$ and ft are initially empty by default. Events have attributes that include the switch where they occur (denoted by s), the sending host (src), the receiving host (dst), and the ingress port at which the packet arrives at the switch (i).

The controller code reacts to events by performing a sequence of (conditionally executed) commands. The command **assume** F instructs the verifier to assume that F holds in the sequel, whereas the command **assert** F causes the verifier to produce an error if F does not hold. The **insert** and **remove** commands are used to update a given relation with a set of tuples. The **flood** command forwards a packet to all switch ports except the packet's ingress port. For readability, we define an **install** command as shorthand for updating the flow table of the relevant switch and a **forward** command to encode sending the current packet, i.e.,

$$\begin{aligned} S.\mathbf{install}(P, I \to O) &\stackrel{\text{def}}{=} ft.\mathbf{insert}(S, P, I \to O) \\ S.\mathbf{forward}(P, I \to O) &\stackrel{\text{def}}{=} sent.\mathbf{insert}(S, P, I \to O) \end{aligned}$$

4.2 From Programs to Formulas

We now show how to convert *CSDN* programs into first-order formulas, which can be fed into the theorem prover. We use the standard Dijkstra's weakest (liberal) precondition calculus, originally invented for specifying the meaning of guarded commands [5]. The main idea is to compute a weakest formula in first-order logic that ensures that execution of a command c leads to the state satisfying a postcondition Q. Such formula is called the *weakest liberal precondition* $wp[\![c]\!](Q)$. The formula $wp[\![c]\!](Q)$ can be used to check the correctness of c for a given precondition P by asserting that $P \Rightarrow wp[\![c]\!](Q)$. Alternatively, models of $P \land \neg wp[\![c]\!](Q)$ are counterexample to the fact that Q holds when c is executed on states satisfying P. Finally, I is an inductive invariant for a command c if and only if $I \Rightarrow wp[\![c]\!](I)$. Notice that these rules only apply to prove the safety of the networks.

Table 5 contains syntax directed rules for computing the weakest (liberal) preconditions of *CSDN* commands. These rules compute a formula representing the largest set of states from which a command executes without failure, thus defining the axiomatic meaning of *CSDN* commands. The first section defines the meaning of atomic commands. The second section defines the meaning of controller and switch events. For brevity, we omit the rules for the while-loops (e.g., see [7]), which assert that (i) the loop invariant initially holds, (ii) that the loop invariant is preserved by the loop body, and (iii) that the loop invariant and the negation of program condition imply the postcondition.

It is interesting to note that destructive updates to relations (insertions and removals) are simply handled using Boolean operations. An alternative is to use a version of McCarthy [18] store functions specialized to updating relations in the way Table 5 prescribes. However, in our case, this just introduces more overhead than savings because relations are never passed as first-class objects and there are no nested access patterns, where McCarthy stores are known to provide a more succinct representation.

Controller events $\mathbf{pktIn}(s, p, i)$ are assumed to be triggered when a packet arrives at a switch and there is no entry in the flow table for handling this packet. Switch events $\mathbf{pktFlow}(s, p, i \to o)$ represent a new packet arriving at a switch and being handled according to an existing entry in the flow table. Observe that this existing packet-handling rule can use the output port `null` to drop the packet. We will slightly abuse the notation, as flow events are implicit and do not include commands. We call events and their commands *guarded commands*.

Priorities The OpenFlow standard supports priorities that allow a programmer to install some rules without removing existing rules. Only the flow rule with the maximal priority is executed. We implement flow tables with priorities by including an extra column in ft. Accordingly, the semantics of the flow event from Table 5 is

I	Formula	Intended Meaning
L_1	$S.ft(Src \to Dst, I \to O) \Rightarrow path(S, O, Dst)$	*Correctly learned connections*
L_2	$\texttt{connected}(S, I, H) \Rightarrow path(S, I, H)$	*Consistent SDN program data structure*
L_3	$S.ft(Src \to Dst, I \to O) \Rightarrow \texttt{connected}(S, I, Src) \wedge \texttt{connected}(S, O, Dst)$	*Consistent learning*
L_4	$rcv^{this}(S, Src \to Dst, I) \wedge (\exists O_1 \in PR : O_1 \neq I \wedge path(S, O_1, Dst)) \Rightarrow$ $\exists O_2 \in PR : path(S, O_2, Dst) \wedge S.sent(Src \to Dst, I \to O_2)$	*Guranteed forwarding*

Table 4. Invariants for the learning switch. L_1, L_2, L_3 are safety invariants and L_4 is a transition invariant.

Ctrl ::=	*Init**(*Evnt* $\Rightarrow$ *Cmd*)*	controller	*Var* ::=	**var** *Id* : *Tid*	local variable
Init ::=	**rel** *Rid*(*Tid**)	declare relation	*Cmd* ::=	**skip**	do nothing
|	**rel** *Rid*(*Tid**) $=$ (*Pred**)*	..with initialization	|	*Id*.**flood** (*Exp**)	flood to all ports
|	**var** *Id* : *Tid*	new variable	|	**assume** *F*	assume that *F* holds
|	**topo** *Fid* : *F*	topology invariant	|	**assert** *F*	assert that *F* holds
|	**inv** *Fid* : *F*	safety invariant	|	*Rid*.**insert** (*Pred**)	relation insertion
|	**trans** *Fid* : *F*	transition invariant	|	*Rid*.**remove** (*Pred**)	relation removal
Event ::=	**pktIn** (*Exp**)	packet-in event	|	**if** *Cond* **then** *Cmd** **else** *Cmd**	conditional
Exp ::=	*const* | *Id*		|	**while** *Cond* **inv** *F* **do** *Cmd*	while-loop
Cond ::=	**True** | **False**	boolean values	|	*Id* $=$ *Exp*	variable assignment
|	*Rid* (*Exp**)	tuple in a relation	|	*Cmd Cmd*	sequence
|	*Exp* $=$ *Exp*	equality	|	{*Var** *Cmd*}	block of commands
|	$\neg$*Cond*	negation	*Pred* ::=	*Exp*	restrict to values
|	*Cond* $\wedge$ *Cond*	conjunction	|	*Pred* $\wedge$ *Pred*	conjunction of preds
|	*Cond* $\vee$ *Cond*	disjunction	|	$*$	wildcard

Figure 7. The abstract syntax of *CSDN*. *F* is a first-order formula defined in Fig. 5.

$$
\begin{array}{l}
wp[\![\textbf{skip}]\!](Q) \stackrel{\text{def}}{=} Q \\
wp[\![\textbf{assume } F]\!](Q) \stackrel{\text{def}}{=} F \Rightarrow Q \\
wp[\![\textbf{assert } F]\!](Q) \stackrel{\text{def}}{=} F \wedge Q \\
wp[\![r.\textbf{insert } P]\!](Q) \stackrel{\text{def}}{=} Q[r(\vec{x}) \vee [\![P]\!]_{FO}(\vec{x})/r(\vec{x})] \\
wp[\![r.\textbf{remove } P]\!](Q) \stackrel{\text{def}}{=} Q[r(\vec{x}) \wedge \neg[\![P]\!]_{FO}(\vec{x})/r(\vec{x})] \\
wp[\![s.\textbf{flood}(p,i)]\!](Q) \stackrel{\text{def}}{=} Q[S.sent(P,I,O) \vee (S = s \wedge P = p \wedge I = i \wedge O \neq i \wedge O \neq null)/S.sent(P,I,O)] \\
wp[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!](Q) \stackrel{\text{def}}{=} (b \Rightarrow wp[\![c_1]\!](Q)) \wedge (\neg b \Rightarrow wp[\![(c_2]\!](Q)) \\
wp[\![c_1; c_2]\!](Q) \stackrel{\text{def}}{=} wp[\![c_1]\!](wp[\![c_2]\!](Q)) \\
\hline
wp[\![\textbf{pktIn}(s,p,i) \Rightarrow c]\!](Q) \stackrel{\text{def}}{=} (rcv^{this}(s,p,i) \wedge \neg\exists O : PR.\ s.ft(p, i \to O)) \Rightarrow wp[\![c]\!](Q) \\
wp[\![\textbf{pktFlow}(s,p,i,o) \Rightarrow \text{``}\textbf{forward}\text{''}]\!](Q) \stackrel{\text{def}}{=} (rcv^{this}(s,p,i) \wedge s.ft(p, i \to o)) \Rightarrow wp[\![s.\textbf{forward}(p,i,o)]\!](Q)
\end{array}
$$

Table 5. Rules for computing weakest (liberal) preconditions for *CSDN* programs. $Q[\psi/\varphi]$ denotes the substitution of all occurrences of φ in Q by ψ. The meaning of predicates $[\![P]\!]_{FO}$ is a first-order formula over r's columns defined in Table 6.

$$
\begin{array}{l}
[\![exp]\!]_{FO}(t) \stackrel{\text{def}}{=} exp = t \\
[\![*]\!]_{FO}(t) \stackrel{\text{def}}{=} \textbf{True} \\
[\![P_1 \wedge P_2]\!]_{FO}(t) \stackrel{\text{def}}{=} [\![P_1]\!]_{FO}(t) \wedge [\![P_2]\!]_{FO}(t) \\
\hline
[\![P_1, P_2, \ldots, P_k]\!]_{FO}(t_1, t_2, \ldots, t_k) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{k} [\![P_i]\!]_{FO}(t_i)
\end{array}
$$

Table 6. Converting predicates into first-order formulas. The meaning of predicates $[\![P]\!]$ is a first-order formula over the columns of the relation.

modified as

$$
\begin{array}{l}
wp[\![\textbf{pktFlow}(s,p,i,o) \Rightarrow \text{``}\textbf{forward}\text{''}]\!](Q) \stackrel{\text{def}}{=} \\
\quad (rcv^{this}(s,p,i) \wedge \exists \alpha\colon PRI.\ maxft(\alpha, s, p, i, o)) \Rightarrow \\
\quad wp[\![s.\textbf{forward}(p,i,o)]\!](Q)
\end{array}
$$

with $PRI = Nat$. Here, predicate $maxft(\alpha, s, p, i, o)$ denotes the fact that α is the maximal priority for the matching flow rules, i.e., $s.ft(\alpha, p, i \to o)$ and

$$\forall \alpha' \colon PRI, O' \colon PR.\ \ s.ft(\alpha', p, i \to O') \Rightarrow \alpha' \leq \alpha.$$

4.3 From Formulas to Theorems and Models

Our evaluation in Section 5 shows that the verification conditions can be solved by Z3. The invariants we encountered so far were all restricted to formulas with a quantifier prefix $\forall\exists$. Proving such invariants could easily be highly intractable and we explain the apparent ease based on the following observation:

Observation: Instantiation dependencies are shallow. By inspecting verification conditions from *CSDN* programs we observed that the formulas could be proved or disproved using relatively few instantiations. The reason is that the formulas do not contain opportunities for *pumping*. In other words, instantiations do not produce new opportunities for instantiations. For example, when Skolemizing $\mathcal{I}_3 \wedge \neg wp[\![c]\!](\mathcal{I}_3)$ we obtain a formula of the form (Skolem functions have hats, and the transition relation is abstracted as ρ):

$$\begin{aligned} & \mathtt{tr}(S, H) \Rightarrow S.sent(\widehat{Src}(S, H) \to H, prt(1) \to prt(2)) \\ \wedge \quad & \rho(\mathtt{tr}, sent, \mathtt{tr}', sent') \\ \wedge \quad & \mathtt{tr}'(\hat{S}, \hat{H}) \\ \wedge \quad & \forall Src \in HO : \neg \hat{S}.sent'(Src \to \hat{H}, prt(1) \to prt(2)) \end{aligned}$$

The opportunities for instantiating Src is limited to $\widehat{Src}(S, H)$.

4.4 Strengthening of Invariants

To ease the burden of writing inductive invariants, we implement a technique of strengthening for state invariants. The idea is to apply iteratively the weakest precondition operator *wp* to a given invariant I. Since the invariants are expressed as arbitrary first-order formulas, the stabilization checking is expensive in general. Therefore, we apply *wp* only finitely many times with the number of iterations limited by the user. Formally, for an event (a guarded command) e and a formula ϕ, we recursively define the n-strengthening by

$$\begin{aligned} Str^{(0)}(\phi, e) &= \phi \\ Str^{(n+1)}(\phi, e) &= Str^{(n)}(\phi, e) \wedge wp[\![e]\!](Str^{(n)}(\phi, e)). \end{aligned}$$

Given a set of events E, the function $Str^{(n)}(\phi, E)$ iteratively applies $Str^{(n)}(\phi, e)$, for all $e \in E$, in some order.

For each state invariant I, we instead try to verify its version n-strengthened by all events of *Prog*. All values of n are tried consequently from 0 to a bound provided by the user until all invariants are proven. If the tool fails to generate inductive invariants from the given ones within the bounded number of iterations, respective counterexamples are returned. In most of our experiments, $n = 1$ was sufficient to obtain invariants strong enough to prove their inductiveness.

5. Preliminary Experience

This section describes our preliminary experience with implementing VeriCon using Z3 and applying it to SDN programs. The program code and the invariants for both correct and incorrect programs are available online [2]. The rest of the section is organized as follows: Section 5.1 describes the actual implementation; Section 5.2 describes our experience applying VeriCon to correct programs; and Section 5.3 describes our experience applying VeriCon to incorrect programs and to programs with incorrect invariants.

All experiments were performed on an Intel i5 1.3GHz, 4GB, MacBook Air running OSX 10.8.5.

5.1 Implementation

We implemented the pseudocode shown in Fig. 8 in Python using the Z3 Python API. We use PLY (Python Lex-Yacc) to parse *CSDN* programs annotated with: (i) topology invariants; (ii) safety invariants; and (iii) transition invariants. The VC generator uses the rules of Table 5 to compute verification conditions.

The tool accepts as input a *CSDN* program *Prog* and a number $n_{\max}$ limiting the depth of invariant strengthening (default value is 0). First, we check that the topology constraints are consistent with the initial states of the network. Then, for each value n from 0 to a $n_{\max}$ we proceed as follows. We strengthen the safety invariants with n applications of *wp* for all events of *Prog*. We check that the strengthened safety invariants hold for the initial states under the topological assumptions. We then check that the topology, the strengthened safety invariants, and the transition invariants are preserved by the execution of arbitrary events executed on every admissible state. If this is not the case, we convert the model generated by Z3 into a readable counterexample in a graphical form, by means of the GraphViz library.

[2] `http://www.cs.tau.ac.il/~shachar`

VeriCon(*Prog*, $n_{\max}$)
  Let *Topo* be the set of topology invariants in *Prog*
  Let *Inv* be the set of safety invariants in *Prog*
  Let *Trans* be the set of transition invariants in *Prog*
  Let *Event* be the set of events of *Prog*
  Let *Init* be the formula describing the initial states of *Prog*
  if not $SAT(Init \wedge (\bigwedge Topo))$
    then return *topology and initial conditions are incompatible*
  for $n = 0$ **to** $n_{\max}$ **do**
    Let $Inv^{\#} = \{Str^{(n)}(I, Event) \mid I \in Inv\}$ *// strenghtened invs*
    Let $Ind \stackrel{\text{def}}{=} \bigwedge(Inv^{\#} \cup Topo)$ *// candidate inductive formula*
    if there exists $I \in Inv^{\#}$ s.t. $SAT(Init \wedge (\bigwedge Topo) \wedge \neg I)$
      then report I *does not hold on initial states*
    if there exist $ev \in Event$ and $I \in Inv^{\#} \cup Topo \cup Trans$
      s.t. $SAT(Ind \wedge \neg wp[\![ev]\!](I))$
      then report I *is not provable on event* ev *using Ind*
    else return *all proved*

Figure 8. A pseudcode for the VeriCon tool.

5.2 Verification Examples

Table 7 shows the running times of VeriCon on correct SDN programs. It was natural to express safety invariants using first-order logic. The table provides the number of goal safety and transition invariants that we verified, e.g., consistency of controller structures and that all the flows satisfy the intended access policy. Additionally, it shows the number of auxiliary invariants that make the goal invariants inductive. In most cases, the auxiliary invariants can be inferred automatically by the tool using one strengthening iteration. The only case in which one iteration did not suffice is the Resonance example.

5.2.1 A Stateless Firewall

Fig. 9 describes a firewall with the same functionality as the stateful firewall presented in Fig. 1, except it requires fewer interactions with the SDN program (via controller events), making it more scalable. The main idea is that the SDN program uses the power of OpenFlow to simultaneously install two rules at the switch for source and target packets.

5.2.2 A Firewall with Migration

One of the interesting issues in network management is coping with host migration. Fig. 10 shows a firewall implementation that supports migration of trusted hosts. We say that a host is trusted if it either sent/received (on some switch) a message through/from port 1. Thus, when a trusted host migrates to a new switch, the controller will remember it was trusted before and will allow communication from either port.

5.2.3 Network Authentication with Learning

The *CSDN* program in Fig. 11 is inspired by the code in Resonance [19]. It presents a composition of a learning switch with authentication. In general, composing two network protocols is hard. However, the code is written in a way that enables verification of the composed program with respect to many desired properties. Our approach allows us to verify all the usual properties of a learning switch, as well as the consistency of flow tables with the access directory relation maintained by the controller, and that all packet flows satisfy the intended access policy.

Program	Description	LOC		Rel	Inv			VC		Time
		TOT	MAX		goal	aux	auto	#	$\forall$	
Firewall	Simple stateful firewall, Fig. 1.	8	3	1	1	2	2	998	24	0.12s
StatellessFirewall	Simple stateless firewall, Fig. 9.	4	3	0	1	1	1	446	12	0.06s
FirewallMigration	Firewall with migrating hosts, Fig. 10.	9	4	1	1	2	2	1186	36	0.16s
Learning	Simple learning switch, Fig. 6	8	7	1	2	3	3	1251	18	0.16s
Auth	Authentication on the network with a learning controller switch, Section 5.2.3	15	14	4	6	3	3	2284	23	0.21s
Resonance	Learning switch with authentication from [19], Section 5.2.4.	93	92	16	5	3	0	6319	24	0.21s
Stratos	Forwarding traffic through a sequence of middleboxes [8, 21], Section 5.2.5	29	28	4	3	0	0	1493	16	0.09s

Table 7. Running times for *VeriCon* on correct SDN controller programs. LOC — the number of code lines where MAX is the maximal number of lines per event and TOT is the total number. Rel — the number of user provided relations in the controller code. Inv — specification safety and transition invariants: the number of goal invariants, the number of auxiliary invariants that make goal invariants inductive, and the number of auxiliary invariants automatically inferred by the tool, respectively. VC — the tool generated verification conditions: # — total number of sub-formulas, $\forall$ — quantifier nesting. Time — the running times for Z3.

pktIn$(s, src \to dst, prt(1)) \Rightarrow$ // packet from a trusted host
  s.**forward**$(src \to dst, prt(1) \to prt(2))$ // forward the packet to untrusted hosts
  s.**install**$(* \to dst, prt(1) \to prt(2))$ // insert a rule to forward future packets targeted at dst
  s.**install**$(dst \to *, prt(2) \to prt(1))$ // insert a rule to forward future packets coming from dst

Figure 9. A simple stateless firewall monitoring the traffic from untrusted to trusted hosts.

rel $\texttt{tr}(HO) = \{\}$ // declare a relation of trusted hosts (initially empty)
pktIn$(s, src \to dst, prt(1)) \Rightarrow$ // packet from a trusted host
  s.**forward**$(src \to dst, prt(1) \to prt(2))$ // forward the packet to untrusted hosts
  `tr`.*insert*(dst) // insert dst into trusted controller memory
  `tr`.*insert*(src) // insert src into trusted controller memory
  s.**install**$(src \to dst, prt(1) \to prt(2))$ // insert a per-flow rule to forward future packets
pktIn$(s, src \to dst, prt(2)) \Rightarrow$ // packet from a presumably untrusted host
  if $\texttt{tr}(src)$ **then**
    s.**forward**$(src \to dst, prt(2) \to prt(1))$ // forward the packet to trusted hosts
    s.**install**$(src \to dst, prt(2) \to prt(1))$ // insert a per-flow rule to forward future packets

Figure 10. A simple stateful firewall monitoring the traffic from untrusted to trusted hosts with migrating hosts.

var `authServ` : HO // a designated host is an authentication server
rel $\texttt{auth}(HO) = \{\texttt{authServ}\}$ // declare a relation of authenticated hosts (initially, contains `authServ` only)
rel $\texttt{connected}(SW, PR, HO) = \{\}$ // declare a relation with three arguments to store learned connections
pktIn$(s, src \to dst, i) \Rightarrow$ // unknown packets
  `connected`.*insert*(s, i, src) // learn a new connection
  if $src = \texttt{authServ}$ **then** // received a message from the authentication server
    `auth`.*insert*(dst) // destination is now authenticated
  if $\texttt{auth}(src) \wedge \texttt{auth}(dst)$ **then** // got a packet from an authenticated host to an authenticated host
    var $o : PR$ // a local variable for egress port
    if $\texttt{connected}(s, o, dst)$ **then** // destination of the packet is already learned
      s.**forward**$(src \to dst, i \to o)$ // forward the packet
      s.**install**$(src \to dst, i \to o)$ // install a new rule
    else s.**flood**$(src \to dst, i)$ // otherwise flood the packet
  else // the sender is not authenticated (hence, it is not `authServ`)
    if $dst = \texttt{authServ}$ **then** // the sender is a "normal" host trying to pass the authentication process
      s.**flood**$(src \to dst, i)$ // flood the packet

Figure 11. A learning switch controller code with authentication.

5.2.4 The Resonance Example

We implemented a simplified version of Resonance [19], an access control approach for host authentication in enterprises. Unlike the original work, we do not model redirection of the web-traffic.

In Resonance, a host could be in one of four states: (i) Registered, (ii) Authenticated, (iii) Operational, or (iv) Quarantined. For each state there are dedicated management servers, and a host is only allowed to communicate with the servers responsible for its current state. Once a host is marked as operational, we allow it to communicate with other operational hosts.

Transition between host states is controlled by the network management servers, which notify the controller on changes in the authentication procedure. The host becomes authenticated if the authentication servers approve its registration, and operational when it gets scanned, by some scanning server, and is found to be free of vulnerabilities. As the scanning servers perform random scans on hosts, a host may become quarantined at any given time (if it was authenticated before).

The key invariants we verified are (1) the installed flow rules satisfy the access policy, and (2) all packet flows in the network respect the policy, i.e., a packet is dropped if and only if it violates the policy.

5.2.5 The Middlebox Composition Example

We implemented a simple application to forward traffic through a sequence of middleboxes, similar to the traffic steering performed by Stratos [8] and SIMPLE [21]. However, unlike these more advanced frameworks, our implementation does not handle middleboxes that modify packet headers or terminate connections, nor does our application perform any weighted selection of middlebox instances. Furthermore, we reactively install rules for each middlebox when the first packet of each flow (where a flow is defined by a pair of source and destination IP addresses) is emitted by the previous middlebox in the sequence. Currently, we have only considered a simple case of one switch in the network. We have verified that the flow table is consistent with the specification of Stratos' chains which implies that (i) all packets of a flow traverse an instance of each middlebox in the sequence, and (ii) all packets of a flow (in both the forward and backward directions) traverse the same set of instances throughout the lifetime of the flow.

5.3 Buggy Examples

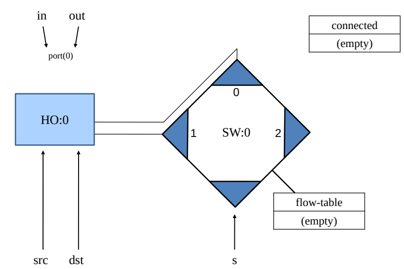
We also applied the tool to erroneous programs and to programs with incorrect assertions. The results, including run-time statistics, formula sizes, and topology sizes, are reported in Table 8.

A simple kind of bug which can occur in the learning switch (Program Learning-Nosend) is when the SDN program forgets to send a message when the destination is known, i.e., the send command is omitted. VeriCon detects that the controller event violates the L_4 transition invariant and generates the counterexample shown in Fig. 12.

Another type of bug that can occur in SDN programs is that the data structure of the SDN program is inconsistent with the forwarding tables of the switches. For example, in program Auth-NoFlowRemoval, we enhanced the network authentication controller with the ability to remove hosts. VeriCon uncovered a bug of a **pktIn** event violating the authentication protocol. This bug occurred because the forwarding rules were not removed from the switch forwarding tables, rendering re-authentication impossible.

6. Related Work

The past few years have witnessed a surge of interest in SDNs. We now discuss the work most relevant to ours along these lines.



$$pktIn(s, src \rightarrow dst, in)$$

Figure 12. A counterexample which shows a scenario where the programmer forgot the line **forward**(...) of Fig. 6, causing a black hole — a packet may be lost.

Language abstractions. [6, 25] introduce abstractions for programming controllers in order to simplify the task of programming controllers. [10] shows that the compiler from a high-level language, called NetCore, to OpenFlow generates semantically equivalent code. [20] defines a nice declarative language to ease the task of programming and verifying SDN programs. NetKAT [2] presents an equational calculus for imperative, finite state, SDN programs. Declarative programming is also successfully used for updating multi-tenant networks [16].

In contrast, our focus is on the orthogonal problem of verifying the safety of infinite state SDN programs. The ultimate goal is to verify SDN programs in stylized Java or Python, but we focus on a imperative language like *CSDN* which captures the essence of imperative SDN programming. In the future it is worthwhile to apply VeriCon to declarative programs.

Finite-state model checking of SDN programs. NICE [3] was the first system to use finite-state model checking to verify the correctness of SDN controllers. The SDN program is modeled as a state-transition system with events similar to those in VeriCon. The concrete network topology is also explicitly modeled. Finite state model checking has many advantages over Hoare style verification: it does not require inductive invariants, and it can employ simpler verification technology. It can be easily applied to arbitrary programs. However, it is unsound in the sense that it can never prove the absence of errors in the infinite state SDN program. Also, it is hard to scale finite state model checking to realistic networks. In contrast, we use first-order logic to model the admissible topology and network-wide invariants. Consequently, VeriCon is able to verify the absence of errors and can also potentially handle huge topologies. We note, however, that VeriCon relies on user-provided invariants and a first-order (potentially non-terminating) theorem prover. Our preliminary experience with Z3 is fairly positive.

Finite-state model checking has also been applied to verify SDN programs on large networks [23]. Two examples, the learning switch and the stateful firewall from this work, use manual abstractions. Our results establish that verification with VeriCon (with infinite states) is orders of magnitude faster than the approach in [23] ($0.13s$ vs. $68352s$ for the finite-state abstraction).

Verificare [24] also uses finite state CTL model checking. The FlowLog [20] system also employs finite-state model checking by limiting the number of packets sent. It also shows that for some specifications this bound suffices to obtain sound results.

In [17], NICE was extended to perform concolic testing [9] and thus reduce the number of missed bugs.

Checking invariants by analyzing snapshots of the network. [14] suggests a novel method for checking certain network proper-

Benchmark	Description	VC		CE size		Time
		#	$\forall$	# H	# SW	
Auth-NoFlowRemoval	Tried to add the ability to un-authenticate hosts, but forgot to remove hosts from the flow table.	2317	19	3	2	0.18s
Firewall-ForgotConsistency	Forgot part of the flow consistency invariant.	969	24	5	3	0.11s
Firewall-ForgotPortCheck	Forgot to check if trusted on events from port 2.	976	24	6	4	0.13s
Firewall-ForgotTrustedInvariant	Forgot to add an invariant defining what is a trusted host.	616	16	6	4	0.09s
Learning-NoSend	Forgot to forward the packets.	1248	18	1	1	0.15s
Resonance-StatesNotMutuallyExclusive	Forgot to add an invariant defining that states must be mutually exclusive.	4440	17	11	4	0.19s
StatelessFireWall-AllowAll2to1Traffic	Added a flow allowing all traffic from port 2 to 1.	444	12	4	2	0.07s

Table 8. Detection of several kinds of bugs. VC — the tool generated verification conditions: # — total number of sub-formulas, $\forall$ — quantifier nesting. CE size — sizes of the generated counterexamples: #H — number of hosts, #SW — number of switches. Time — the running times for Z3.

ties by analyzing packet headers. As with the above schemes, the approach described in [14] can establish the existence of bugs but not their absence.

Dynamically checking SDN programs. [13, 15] show how to monitor the correctness of certain properties of SDN programs in real time. The main challenge is to guarantee that this can be accomplished without harming network performance. VeriCon runs at compile-time, and so it verifies correctness or, alternatively, exhibits errors before the code is actually executed. We view controller code verification ala VeriCon and dynamic checking as in [13, 15] as two complementary approaches.

7. Conclusion

We presented VeriCon, the first verification tool for (infinite-state) SDN programs. VeriCon reflects two fundamental choices: (i) express controller programs as imperative event-driven programs that manipulate relations; and (ii) express network-wide invariants as first-order logic formulas. From a verification perspective, these two choices fit together well, since they guarantee that the generated verification conditions are simple enough to be expressible in a weak form of first-order logic, which enables complete verification via SMT solvers. Indeed, our results establish that Z3 is able to rapidly verify network-wide invariants of interesting controller programs or, alternatively, to quickly compute a concrete counterexample. VeriCon's encouraging ability to prove the functional correctness of controller programs of interest motivates further research along these lines.

Acknowledgments

We thank Ras Bodik, Ahmed Bouajjani, Nate Foster, Oded Padon, Brandon Heller, Ori Lahav, Aurojit Panda, Hossein Hojjat, Peyman Kazemian, Shriram Krishnamurthi, Teemu Koponen, Ratul Mahajan, Tim Nelson, Mark Reitblatt, Vyas Sekar and Sharon Shoham, for their insightful comments on earlier versions of this paper. The research of Itzhaky, and Sagiv has received funding from the European Research Council under the European Union's Seventh Framework Program (FP7/2007–2013) / ERC grant agreement n° [321174-VSSC]. The research of Karbyshev was funded by Technical University of Munich. Karbyshev thanks Prof. Sagiv for inviting him to visit Tel Aviv University. Part of this work was done while Sagiv was visiting Microsoft Research.

References

[1] *OpenFlow Switch Specification*, Oct. 2013. Version 1.4.0.

[2] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic foundations for networks. In *POPL* (2014), S. Jagannathan and P. Sewell, Eds., ACM, pp. 113–126.

[3] CANINI, M., VENZANO, D., PERES, P., KOSTIC, D., AND REXFORD, J. A NICE Way to Test OpenFlow Applications. In *NSDI* (2012).

[4] DE MOURA, L. M., AND BJØRNER, N. Z3: An Efficient SMT Solver. In *TACAS* (2008), C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 337–340.

[5] DIJKSTRA, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM 18*, 8 (1975), 453–457.

[6] FOSTER, N., GUHA, A., REITBLATT, M., STORY, A., FREEDMAN, M. J., KATTA, N. P., MONSANTO, C., REICH, J., REXFORD, J., SCHLESINGER, C., WALKER, D., AND HARRISON, R. Languages for software-defined networks. *IEEE Communications Magazine 51*, 2 (2013), 128–134.

[7] FRADE, M., AND PINTO, J. Verification conditions for source-level imperative programs. *Computer Science Review 5*, 3 (2011), 252–277.

[8] GEMBER, A., KRISHNAMURTHY, A., JOHN, S. S., GRANDL, R., GAO, X., ANAND, A., BENSON, T., AKELLA, A., AND SEKAR, V. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. Tech. Rep. arXiv:1305.0209, 2013.

[9] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: directed automated random testing. In *PLDI* (2005), pp. 213–223.

[10] GUHA, A., REITBLATT, M., AND FOSTER, N. Machine-verified network controllers. In *PLDI* (2013), pp. 483–494.

[11] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. B4: Experience with a Globally-deployed Software Defined WAN. In *ACM SIGCOMM* (2013), pp. 3–14.

[12] KATTA, N. P., REXFORD, J., AND WALKER, D. Logic programming for software-defined networks. In *ACM SIGPLAN Workshop on Cross-model Language Design and Implementation* (Sept. 2012).

[13] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real Time Network Policy Checking using Header Space Analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)* (2013).

[14] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header Space Analysis: Static Checking For Networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)* (2012).

[15] KHURSHID, A., ZHOU, W., CAESAR, M., AND GODFREY, B. Veriflow: verifying network-wide invariants in real time. *Computer Communication Review 42*, 4 (2012), 467–472.

[16] KOPONEN, T., AMIDON, K., BALLAND, P., CASADO, M., CHANDA, A., FULTON, B., GANICHEV, I., GROSS, J., GUDE, N., INGRAM, P.,

JACKSON, E., LAMBETH, A., LENGLET, R., LI, S.-H., PADMANABHAN, A., PETTIT, J., PFAFF, B., RAMANATHAN, R., SHENKER, S., SHIEH, A., STRIBLING, J., THAKKAR, P., WENDLANDT, D., YIP, A., AND ZHANG, R. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)* (2014).

[17] 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.

[18] MCCARTHY, J. Towards a mathematical science of computation. In *IFIP Congress* (1962), pp. 21–28.

[19] NAYAK, A. K., REIMERS, A., FEAMSTER, N., AND CLARK, R. Resonance: Dynamic Access Control for Enterprise Networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking (WREN '09)* (2009), pp. 11–18.

[20] NELSON, T., FERGUSON, A. D., SCHEER, M. J. G., AND KRISHNAMURTHI, S. A balance of power: Expressive, analyzable controller programming. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)* (2014).

[21] QAZI, Z. A., TU, C.-C., MIAO, R., SEKAR, V., AND YU, M. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *ACM SIGCOMM* (2013), pp. 27–38.

[22] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for network update. In *ACM SIGCOMM* (2012), pp. 323–334.

[23] SETHI, D., NARAYANA, S., AND MALIK, S. Abstractions for model checking sdn controllers. In *FMCAD* (2013).

[24] SKOWYRA, R., LAPETS, A., BESTAVROS, A., AND KFOURY, A. A verification platform for sdn-enabled applications. In *HiCoNS* (2013).

[25] VOELLMY, A., WANG, J., YANG, Y. R., FORD, B., AND HUDAK, P. Maple: simplifying SDN programming using algorithmic policies. In *ACM SIGCOMM* (2013), pp. 87–98.