Abstractions for Model Checking SDN Controllers

Divjyot Sethi, Srinivas Narayana, Sharad Malik
Princeton University

***Abstract*—Software defined networks (SDNs) are receiving significant attention in the computer networking community, with increasing adoption by the industry. The key feature of SDNs is a centralized controller which programs the packet forwarding behavior of a distributed underlying network. This centralized view of control—which is absent in traditional networks—opens up opportunities for full formal verification.**

While there is recent research in formal verification of these networks, model checking the controller behavior as it updates the underlying network has only seen limited application. Existing approaches are limited to verifying the controller for a small number of exchanged packets in the network. In this case study, we extend the state of the art by presenting abstractions for model checking controllers for an arbitrarily large number of packets exchanged in the network. We validate the utility of these abstractions through two applications: a learning switch and a stateful firewall.

I. Introduction

Software defined networks (SDNs) (such as ones based on Openflow [1]) have recently received significant attention in the computer networking community, with increasing relevance to and adoption by industry, *e.g.,* [2]. The key feature of SDNs is a centralized controller (*control plane*) which programs a distributed underlying network (*data plane*). While providing a centralized view of control, any bugs in the controller code can be an Achilles heel to the functioning of the entire network [3]. In this case study, we explore abstractions for proving the correctness of controllers using model checking.

Fig. 1 shows an example topology of an SDN. The data plane consists of three hosts H_A, H_B and H_C which exchange packets pkt_1 and pkt_2 with each other via the network switches S_1, S_2 and S_3[1]. These switches consist of ports p_0, p_1 and p_2. Each location in the network (switch ports and hosts) consists of an input and output buffer (referred to as the *data state*). Each switch enqueues packets in its input port buffers, and eventually forwards them to a set of output ports (or drops them) based on the packet processing logic.

The packet processing logic is encoded into the switches in the form of switch *flow tables* (also referred to as the *network state*). Based on the flow table, the switch applies one of the following 3 actions to an incoming packet: (1) it forwards the packet to a set of output ports of that switch, (2) drops the packet or (3) forwards it to the controller. The switch forwards packets to the controller by reporting them as events. The controller is a piece of software which updates switch flow tables through a standard interface (*e.g.,* Openflow [1]), either in response to events forwarded by switches or spontaneously.

[1]Following the approach of Zhang et al. [4], all the network logic like firewalls, routers etc. can be represented using switches.

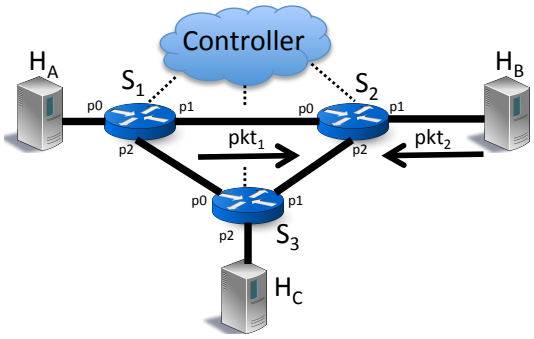

Fig. 1: An example topology.

In this paper, we prove the correctness of the network controller program for a given network topology, and an arbitrarily large number of packets. We focus on *per-packet properties*, which assert the correctness of packet processing in the presence of updates from the controller. Note that these updates may themselves occur in response to events reported due to other packets, *i.e.*, due to *interference* from other packets. Examples of such properties include *no forwarding loop* (*i.e.*, a packet does not loop back to a switch which it has already visited) and *no invalid drop* (*i.e.*, a packet is not dropped due to an invalid controller update to some switch).

Challenges in Verification: The key challenge in model checking SDNs is the state space explosion resulting from the following factors. (1) There can be a large number of packets alive in the network, resulting in a large buffer state—commodity switches can have of the order of ten megabytes of packet buffering per switch (*e.g.,* [5]). Further, these packets create interference for each other by sending events to the controller which then trigger updates to the network state. (2) These packets and the corresponding events can have arbitrary interleavings. (3) The switch flow tables store a mapping from the packet header and the input port to the output port, resulting in a large network state—modern switches can have tens of thousands of flow table entries [6].

Related Work: Existing work in the verification of SDNs exploits the fact that the time between updates to the network state by the controller is much larger than the lifetime of a packet through the network. Thus, the state evolution of the network can be viewed as updates from one network configuration to another via intermediate (*transient*) states, as per the commands from the controller.

Static verification verifies a fixed configuration of the network by using either symbolic simulation [7], by reduction to SAT [4], [8] or, by model checking [9].

Incremental verification approaches extend the static verification approach and incrementally verify the network for

all the network configurations [10], [11]. The property may however be violated in the transient stage.

Safe update builds upon the incremental verification approach by guaranteeing that the property under check holds during the transient stage by using specific update protocols [12]. This work is specific to enforcing properties for which the update protocols are designed—more complex properties may not be enforced with this approach.

Dynamic checking seeks to verify the properties on the network in the presence of arbitrary updates from the controller, even when no specific update protocol has been implemented to ensure the property. This is the space of our work—the other works we are aware of in this space are NICE [3] and FlowLog [13]. They use a model checking based approach to successfully find important bugs in controller code. However, they check the controller for a bounded number of exchanged packets. *In this work we extend dynamic checking and scale to an arbitrarily large number of packets.*

Key Contribution: This work addresses the challenges outlined above by first constructing a data state abstraction, and then, a network state abstraction which builds upon the data state abstraction to significantly reduce the model size.

The data state abstraction is based on the standard data type reduction [14] and addresses challenges (1) and (2) by keeping just one packet (*concrete* packet), (pkt^c), in the system and replacing the effect of all the other packets on the network state by *non-deterministically* injecting an arbitrarily large number of *environment packets* (pkt^e). These packets have arbitrary header values, unless constrained by user-added lemmas, and can be injected at any port of any switch. When injected, these environment packets may be forwarded as events to the controller and trigger updates to the network state. Thus, these packets simulate the updates to the network state triggered by an arbitrarily large number of other packets. Further, the environment packets need not traverse the links and occupy data state as they are directly injected at arbitrary ports—this enables abstracting away all the buffers.

Since there is only one alive packet (pkt^c) in the system after the data state abstraction, the network state abstraction exploits this to address challenge (3) by *case splitting* on the source and destination hosts of this packet: the switch flow table can then be abstracted to contain information specific to only these hosts, which are fixed for a concrete packet[2]. This significantly reduces the network state.

Mechanical Construction: The data state abstraction is constructed by adding a special host for injecting environment packets—this host is independent of the controller application under verification. The network state abstraction adds a runtime check to the flow table to only allow updates corresponding to the selected source and destination hosts.

Experimental Setup: We model the controller code to closely resemble the original (typically Python) code in the Murphi language (CMurphi 5.4.6). To verify the controller for a specified topology and property, we use a Murphi model with (1) the controller code, and (2) the switches and hosts connected according to the specified topology with the property specified on it. The abstractions are also implemented on this model.

We demonstrate the utility of the abstraction by verifying *no forwarding loop* for a learning switch application (controller), `Pyswitch` [15] and *no invalid drop* property for a stateful firewall application. We were able to find bugs in the buggy version of these applications and prove correctness for the correct ones for an arbitrary number of packets.

Limitation: While our approach verifies topologies larger than the state of the art for dynamic checking, it cannot scale to realistically large sizes (*e.g.,* data centers). However, in certain cases it is possible to extend controllers proved on smaller topologies to larger ones through topology abstractions [16].

II. Modeling Network Controllers

The network consists of a controller, switches (switch S_i has an id i) with ports (port p_i has an id i) and hosts (host H_i has an id i). Packets traverse the network by hopping from one location to another (unless they get dropped). Each packet consists of a header and payload (data information). The source and destination host id of a packet pkt are denoted by $pkt.src$ and $pkt.dst$, respectively.

When a packet arrives at a switch port, the switch processes the packet in accordance with the *flow table*. For sake of brevity, we assume that the flow table of a switch S (denoted by $S.ft$) matches on the source, destination, and input port (p_i) of the incoming packet and applies a set of actions to the packet—in general our approach extends to cases where it matches other fields as well. Formally, a flow table is a mapping $S.ft : (pkt.src, pkt.dst, p_i) \rightarrow A$, where A is a set containing one or more actions from the following: (1) forward the packet to a set of output ports P_o of the switch (denoted by $Forward\ (P_o)$), (2) drop the packet ($Drop$), or (3) forward it to the controller ($SendToController$). We denote the set of the above three actions by $\mathcal{A}$.

Controller-switch interaction: The controller switch interaction happens in accordance with the Openflow [1] specification: (1) switches report events to the controller which are enqueued in a per-switch event buffer at the controller, and (2) the controller sends commands to the switch which are enqueued in a command buffer at the switch. Following the approach of Foster et al. [17], the relevant events and commands are described below.

Events: The event used in this paper is the $packet\_in(swID, portID, packet)$ event, which tells the controller that $packet$ has arrived at port with id $portID$ of switch with id $swID$.

Controller Commands: The controller can either react to events reported by the switch through event handlers, or spontaneously program the switch. The controller commands can be one of the following: (1) $install(swID,\ match,\ actions)$: this command updates the flow table of switch $swID$ to apply $actions$ (a subset of $\mathcal{A}$) to all packets which match the pattern specified by $match$. The string $match$ is of the form $\{src : H_1, dst : H_2, inport : p_i\}$, *i.e.*, match all incoming packets with $pkt.src = H_1$, $pkt.dst = H_2$, and ingress port p_i at switch $swID$. (2) $send(swID,\ packet,\ action)$ sends the packet to switch $swID$ where $action$ is

[2]Packet rewrites are discussed at the end of §III.

```
packet_in (swID, inport, pkt):
 1: mactable = ctrlState[swID]
 2: mactable[pkt.src] = inport
 3: if (mactable[pkt.dst] != null)
 4:    outport = mactable[pkt.dst]
 5:    if (outport != inport)
 6:       match = {src:pkt.src, dst:pkt.dst, inport:inport}
 7:       action = {Forward (outport)}
 8:       install (swID, match, action)
 9:       send (swID, pkt, action)
10:       return
11: send (swID, pkt, Flood)
12: return
```

Fig. 2: The `Pyswitch` controller algorithm.

applied to it at the switch. Here $action$ is one of $\{Forward, Drop, Flood\}$, where $Flood$ instructs the switch to forward the packet on all its ports except the packet's ingress port.

Property: We verify invariants of the form $\forall pkt : \phi(B_{pkt})$, where ϕ is a propositional logic formula and B_{pkt} is some per-packet book-keeping state. For example, this state can log the packet history, *i.e.*, switches which the packet has visited, in order to detect loops.

Running example (MAC learning switch): We use a layer 2 MAC address learning switch application, `Pyswitch` [15] with topology as shown in Fig. 1, as a running example to describe key concepts. The controller algorithm is shown in Fig. 2. At a high level, for each switch $swID$, the controller learns a mapping (denoted by $ctrlState[swID]$) from host MAC addresses to ports. This allows the switch to forward packets destined to these hosts. As an example, suppose the packet pkt_1, with source H_A and destination H_B, arrives at port p_0 of switch S_2. In case no match exists for the packet, it is matched to a default flow table entry which forwards a $packet\_in$ event to the controller. The `packet_in` event handler learns that the host H_A is reachable through port p_0 on switch S_2 (line 2). In case the port leading to the destination H_B is unknown, the `if` condition on line 3 evaluates to false, and the packet is flooded on all ports, except the incoming port (line 11). However, if the destination is found in $ctrlState$, the flow table is updated to forward all subsequent packets with the same src, dst and $inport$ to p_1 (line 8).

Property: We verify the *no forwarding loop* property for the `Pyswitch` controller. Due to flooding on a topology (Fig. 1) with a loop, the property is violated. However, if the controller only sends packets along a spanning tree (*e.g.,* no packets between port p_2 of S_2 and p_1 of S_3), the property holds.

III. Abstraction

Data state abstraction: As discussed in §I, the data state abstraction exploits the fact that the property under check is a per-packet property: it checks the property on one concrete packet pkt^c, and abstracts away all the other packets. Continuing on the `Pyswitch` example from §II, suppose pkt^e is injected at port p_2 of switch S_2 in Fig. 1, such that $pkt^e.src = H_A$. This leads the switch to send a $packet\_in$ to the controller which updates $ctrlState[S_2][H_A]$ to p_2. Next, suppose the concrete packet pkt^c with source H_B and destination H_A arrives at port p_1 of S_2. The switch sends this packet to the controller, and since $ctrlState[S_2][H_A] = p_2$, the controller commands the switch to forward the packet out port p_2 (line 9 of Fig. 2) to S_3. If there are no matching entries for pkt^c at S_3 and S_1 both in flow tables and $ctrlState$, the packet gets flooded at both switches (line 11). Thus, pkt^c loops back to S_2, which is a violation of the forwarding loop property.

Refinement: Since the header of pkt^e can take arbitrary values, the updates triggered by pkt^e are highly unconstrained. This leads to both scalability bottlenecks as well as functional incorrectness due to *over-abstraction*, *i.e.*, the model exhibiting more behaviors than are realistically expected. We follow the approach of the CMP (CoMPositional) method [18]: the model is iteratively model checked and refined by the user by adding non-interference lemmas in order to constrain pkt^e. The non-interference lemmas we used typically constrain the model according to reachability in the topology. For example a packet with source H_A cannot be injected at port p_1 of S_2, *i.e.*, $((port = p_1)\&(switch = S_2)) \rightarrow (pkt.src \neq H_A)$. (Since these lemmas are application-independent, they were added pre-emptively to mechanically constrain the data state abstraction.) As per the CMP method, these non-interference lemmas do not over-constrain the environment packet pkt^e: they are also model checked by validating them on pkt^c [18].

Network state abstraction: As discussed in §I, the network state abstraction case splits on the source and destination of pkt^c to abstract the flow tables. Suppose we assume that pkt^c has source H_A and destination H_B. Then, for each switch S, the flow table mapping $S.ft$ can be abstracted to $S.ft^{abs}$ where $S.ft^{abs}(pkt^c, p) = S.ft(pkt^c, p)$ for all ports p of the switch, and $S.ft^{abs}(pkt^e, p) = \{SendToController\}$ for all other packets (*i.e.*, pkt^e). The $Forward$ action is not applied to pkt^e as it is non-deterministically injected at all ports.

Packet rewrites: In order to handle packet rewrites, the network state abstraction can be refined by including flow rules for rewritten header values as well. These rules are needed to process the concrete packet when its header is rewritten.

IV. Experiments

We verified two applications: `Pyswitch` and a stateful firewall on a 2.40 GHz Intel Core 2 Quad processor, with 3.74 GB RAM. Murphi source code is available online [19].

MAC learning switch (`Pyswitch`): For the `Pyswitch` application, we verified loop freedom for the star topology shown in Fig. 1, with non-interference lemmas from §III added pre-emptively for scalability. The loop was found in 0.1 sec with 159 states explored. Next, on constraining the topology to forward packets only along the spanning tree, the model checker proved correctness with an arbitrary number of packets exchanged between H_A and H_B in 600s with 1.45M states. We note that model checking did not finish in a day without the abstractions. Finally, we ran a stress test with a larger *fat tree* topology with 20 switches, 16 hosts and 48 links. While model checking did not finish for an arbitrarily large number of packets, it finished in 68352s for the single packet case with network state abstraction.

Stateful firewall: We consider a simple firewall policy which may be used to prevent direct connections from the Internet into an enterprise network, *e.g.,* to implement Network Address Translation (NAT). Fig. 3 shows an enterprise network

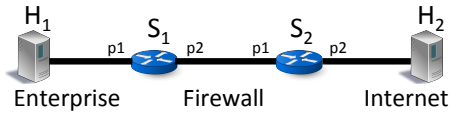

Fig. 3: Two switches acting together as a stateful firewall.

```
1: packet_in (swId, inport, pkt):
2:   if swId = 1 and inport = 1:
3:     match_S1 = {src:pkt.src, dst:pkt.dst, inport:1}
4:     action_S1 = {Forward ({2})}
5:     install (1,match_S1,action_S1)
6:     match_S2 = {src:pkt.dst, dst:pkt.src, inport:2}
7:     action_S2 = {Forward ({1})}
8:     install (2,match_S2,action_S2)
9:   else if swId = 2 and inport = 2:
10:    match = {src:pkt.src, inport:2}
11:    action = {Drop }
12:    install (2,match,action)
```

Fig. 4: Firewall controller for the network shown in Fig. 3.

connected to the Internet via a firewall implemented on two switches S_1 and S_2. The controller (not shown in Fig. 3 for brevity) initializes the *default* behavior of S_1 and S_2 as follows: (1) S_1 sends all incoming traffic at port p_1 to the controller, in addition to forwarding out of p_2, (2) S_1 forwards incoming traffic at p_2 directly to the enterprise host H_1, (3) S_2 sends all incoming traffic at p_1 directly to the Internet, and (4) S_2 forwards incoming traffic at p_2 to the controller. As packets arrive, the controller dynamically updates the switch flow tables to implement the following high-level policy (Fig. 4): (a) All traffic originating from any enterprise host H_1 and destined to any Internet host H_2 is allowed to pass freely. In particular S_1 forwards all traffic to port p_2 bypassing the controller after the first `packet_in` (line 5). (b) Traffic from an Internet host H_2 destined to an enterprise host H_1 is only allowed if the communication was initiated by H_1 first (line 8). (c) If H_2 attempts to communicate with H_1 without prior initiation by H_1, then H_2 is considered malicious and is explicitly blacklisted (line 12).

Verification: We verify the *no invalid drop* property by checking if traffic from Internet host H_2 replying to a request sent by enterprise host H_1 does not get dropped. Due to a race condition between the events reported by S_1 and S_2, the controller erroneously blacklists host H_2. This happens when H_1 sends a first request to H_2. The request goes to S_1 first which forwards it via S_2 to H_2, and also forwards an event e_1 reporting the request to the controller. However, this event gets delayed and in the meantime H_2 replies, and S_2 forwards this to the controller as an event e_2. Since e_2 is processed by the controller before e_1, H_2 is erroneously blacklisted. (Note that events across switches can be processed out of order.) Our approach found this race condition in 0.13 sec with 482 states, without requiring any lemmas.

Fixing the violation: This bug can be fixed by requiring S_1 to wait for the controller before forwarding requests from H_1 to H_2 via S_2. Our approach was able to prove correctness for an arbitrarily large number of packets in 0.19 sec with 613 states, without requiring any lemmas.

V. Conclusion and Ongoing Work

We have presented abstractions for model checking controllers for software defined network applications. These abstractions extend the state of the art by enabling correctness proofs for SDN controllers for an arbitrarily large number of packets and their ensuing controller state updates. As a next step, we plan to explore abstractions to further scale model checking for larger topologies. In particular, since properties are typically violated along a particular path taken by packets in the network, we plan to focus on validating properties for packets taking fixed paths in the topology instead of all possible paths.

Acknowledgment: We thank Jennifer Rexford and Muralidhar Talupur for their helpful ideas and feedback. This work was supported by NSF grant 1111520 and by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, 2008.

[2] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined WAN," in *SIGCOMM*, 2013.

[3] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in *Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[4] S. Zhang, S. Malik, and R. McGeer, "Verification of computer switching networks: an overview," ser. ATVA'12. Springer-Verlag, 2012, pp. 1–16.

[5] Broadcom, 2012. [Online]. Available: http://www.broadcom.com/collateral/etp/SBT-ETP100.pdf

[6] IBM G8264 switch, 2012. [Online]. Available: http://www.openflow.org/wp/ibm-switch/

[7] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: static checking for networks," in *Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[8] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *SIGCOMM*, 2011.

[9] 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*, ser. SafeConfig '10, 2010.

[10] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *NSDI*, 2013.

[11] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[12] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *SIGCOMM*, 2012.

[13] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "A balance of power: Expressive, analyzable controller programming," in *Hot topics in Software Defined Networks (HotSDN)*, 2013.

[14] K. L. McMillan, "Verification of infinite state systems by compositional model checking," in *CHARME*, 1999.

[15] "PySwitch NOX application," http://nddi.googlecode.com/svn/nox/src/nox/coreapps/examples/pyswitch.py.

[16] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *Symposium on Networked Systems Design and Implementation*, 2013.

[17] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: a network programming language," in *International Conference on Functional Programming (ICFP)*, 2011.

[18] C.-T. Chou, P. K. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *Proc. FMCAD*, 2004.

[19] Murphi source code, [Online] https://github.com/ngsrinivas/sdnverify.