Simulation of a Software-Defined Network as One Big Switch

ABSTRACT

Software-defined networking (SDN) technology promises centralized and rapid network provisioning, holistic management, low operational cost, and improved network visibility. Researchers have developed multiple SDN simulation and emulation platforms to expedite the adoption of many emerging SDN-based applications to production systems. However, the scalability of those platforms is often limited by the underlying physical hardware resources, which inevitably affects the simulation fidelity in large-scale network settings. In this paper, we present a model abstraction technique that effectively transforms the network devices in an SDN-based network to one virtualized switch model. While significantly enhancing the simulation scalability, our abstracted model also preserves the end-to-end forwarding behavior of the original network. To achieve this, we first classify packets with the same forwarding behavior into smaller and disjoint groups by analyzing the OpenFlow rules installed on the SDN devices. We then create a graph model for each group representing its forwarding behavior. We traverse those graphs to aggregate the forwarding rules across the network, and finally generate the effective rules for the big switch model. Experimental results demonstrate that the network forwarding logic equivalence is well preserved between the abstracted model and the original SDN network. The model abstraction process is fast, e.g., 3.15 seconds to transform a medium-scale tree network consisting of 53,260 rules. The big-switch model is able to speed up the simulation by 4.3 times in average maybe give a range here, or mention average and best case.

Keywords

Network Simulation; Model Abstraction; Software-Defined Networking;

1. INTRODUCTION

Software defined networking (SDN) centralizes and simplifies control of network management, and has been increasingly adopted in data centers and internet exchange points [8, 11, 15]. Similar to traditional computer network systems, it is crucial to perform appropriate testing and evaluation of SDN-based applications before deploying on a real system. Researchers in the simulation community have extended various existing network simulators to support SDN capability [3, 4, 21]. To improve experimental fidelity, researchers have also developed network emulation testbeds that utilize Linux containers over shared hardware resources and real network stack to run high-fidelity SDN experiments, such as Mininet [13]. However, container-based emulators cannot reproduce the correct behavior of a real network with a large network topology and high traffic load because of the limited underlying physical resources. For example, on a commodity machine with 2.98 GHz CPU, 4 GB RAM, and 3 Gbps internal bandwidth, Mininet can only emulate a network up to 30 hosts, each with a 100 MHz CPU, 100 MB RAM and connected by 100 Mb/s links [14]. Therefore, increasing SDN testbed scalability and speed without losing the desired fidelity is essential.

In this paper, we present a model abstraction technique to transform an SDN-based network model to a "one big switch" based network model. The idea was inspired by the work on rule placement optimization in [17]. With the highly abstracted network, SDN application developers now only need to consider simple end-to-end policy when programming a network, and are shielded from the details on routing policy, switch memory limits, and distributing rules across switches. Our work applies the idea of one big switch abstraction for enhancing the scalability of network simulation and emulation, while preserving the end-to-end forwarding logic. The technique is useful if users only care about the end-to-end behavior rather than the details within the network, such as hop-by-hop routing, memory consumption on switches.

We develop a three-step approach to transform an SDN network to a big OpenFlow switch based network, while still preserving the network forwarding logic equivalence. The high-level idea is illustrated in Figure 1, and the details are discussed in Section 3. We first group all packets into equivalence classes by analyzing the matching fields (e.g., source/destination MAC address/IP address/port, VLAN id, etc.) of the OpenFlow rules installed on the switches. An equivalence class represents a set of packets of the same network forwarding behavior. We then create a graph-based model for each equivalence class to model its packet forwarding behavior. Finally, we traverse all the forwarding graph models to generate rules for the big switch, and the number

SIGSIM-PADS '17, May 24–26, 2017, Singapore.

DOI: http://dx.doi.org/xx.xxxx/xxxxxxxxxx.xxxxxx

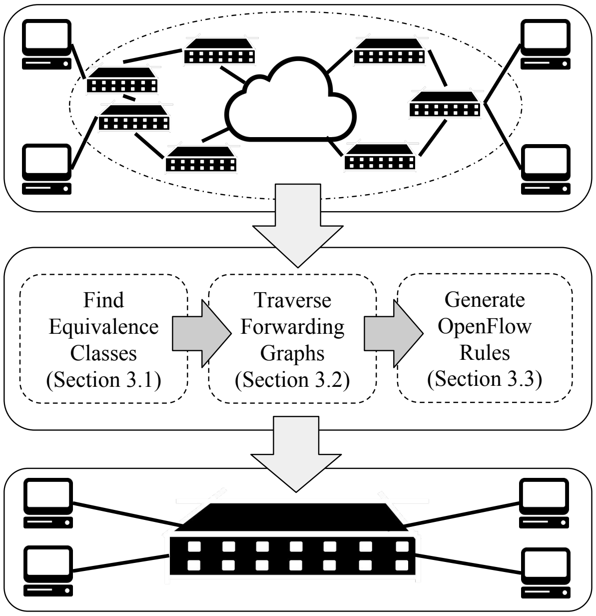


Figure 1: Transforming an SDN network to a big OpenFlow switch based network while preserving the network forwarding logic equivalence.

of rules is largely reduced. This way, we reduce the SDN network to a big-switch-based network to improve the scalability of SDN simulation or emulation.

The reduction in the number of switches and the number of rules significantly enhances the testbed scalability and reduces the experiment running time. may mention some results here. We can also reuse the abstracted network model. For example, after one complete experimental run of a complex network, users can abstract (possibly part of) the network, and reproduce the simulation results with a much simpler configuration, including link connectivity and flow tables. We can partition a large-scale network model, and abstract each partition in parallel. By combining those abstracted network models, a testing platform with limited hardware resources now can afford such network simulation/emulation experiments. As the network state evolves, the abstracted big switch model may also need to be frequently updated. Our approach is lightweight. For example, we can reduce 50,000 rules in a large tree-topology network to 5,000 rules in a big-switch-based network in three seconds while preserving the network forwarding rule equivalence. In addition, our approach allows incrementally updating the big switch model, i.e., modifying the rules that are only affected by the current network changes.

In this work, we present a model abstraction technique to reduce networked SDN switches to a one-big-switch model. We mainly focus on preserving the end-to-end network forwarding logic. Our long term goal is to investigate systematic model abstraction approaches that preserve end-to-end performance equivalence as well, such as latency and packet drop, to further enhance the model fidelity.

The remainder of this paper is organized as follows. Section 2 illustrates the problem and the approach using a simple motivating example. Section 3 describes the details of the three-step model abstraction design. Section 4 presents the evaluation results in terms of forwarding logic equivalence, simulation time, reduction in flow rules, and model abstraction execution time. Section 5 summarizes the related works, and Section 6 concludes the paper with future works.

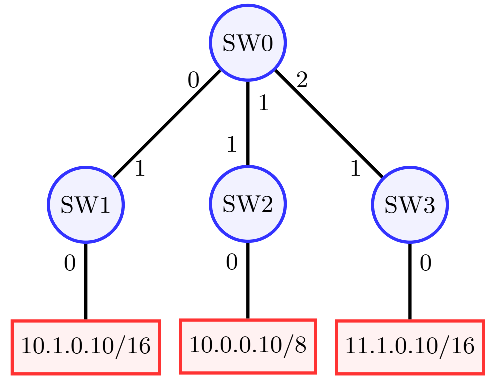


Figure 2: A Tree-Topology SDN Network

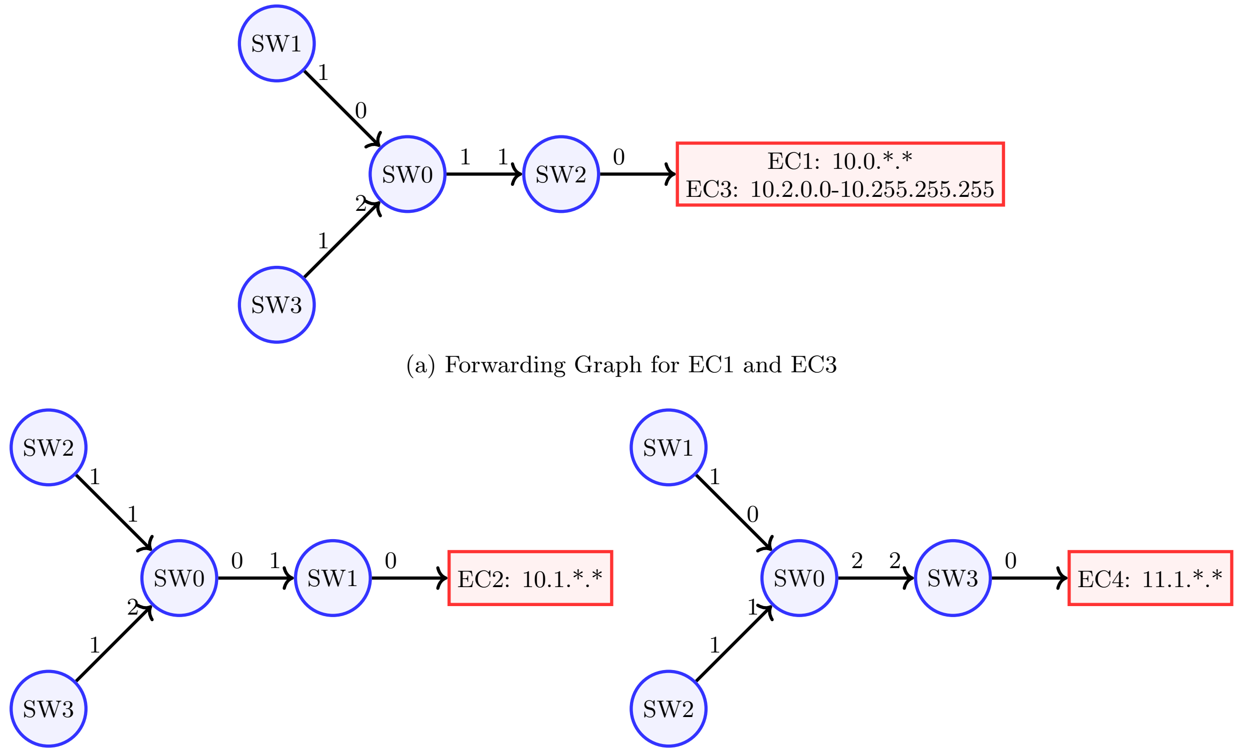
2. MOTIVATING EXAMPLE

In this section, we describe our model abstraction technique to transform an SDN network model to a "big switch" model with a concrete network example. Let us consider a tree-topology network connected by four OpenFlow switches, as shown in Figure 2. The centralized SDN controller (not shown in the figure for simplicity) installs the forwarding rules on each switch to establish connections for all three subnets. All the switch rules are shown in Table 1. We assume that during the process of model abstraction, the rules have been installed on each OpenFlow switch, and there is no link down or rule modification. OpenFlow switch 0 (i.e., SW0) works as an **aggregation switch** that provides connectivity for other switches. SW1, SW2 and SW3 work as **edge switches** that provide connectivity for each subnet, and each edge switch connects to one end-host.

Our approach abstracts the network to one "big switch" that has logically equivalent forwarding behavior.

The first step in the abstraction process is to extract *equivalence classes* through the OpenFlow rules installed on network devices, i.e., aggregation and edge switches. Equivalence class (EC) is the set of packets that experience identical forwarding action at **any** network device. We utilize EC to merge all the rules on a set of switches. For example, the flow rules shown in Table 1 can be sliced into four **disjoint** ECs based on the NW_DST field as follows. Note that the matching field IN_PORT cannot be used in identifying ECs, since it is not a packet-dependent, but topology-dependent field.

- Packets in EC1 are destined to the network address 10.0.*.*.
- Packets in EC2 are destined to hosts with address 10.1.*.*.
- Packets in EC3 are destined to the address range from 10.2.0.0 to 10.255.255.255
- Packets in EC4 are destined to the subnet 11.1.*.*.



(a) Forwarding Graph for EC1 and EC3

(b) Forwarding Graph for EC2

(c) Forwarding Graph for EC4

Figure 3: Forward Graph for Each EC

After identifying all the ECs from the rule set, we generate *forwarding graph* for each EC, which models how packets within an EC are forwarded through the network [19]. The node in a forwarding graph represents a network device, and the directed edge represents how the network device forwards the packets. The sink nodes, i.e., the red rectangle nodes in Figure 3, indentifies the EC that this forwarding graph belongs to. Each equivalence class will have exactly one forwarding graph, as shown in Figure 3. Note that {EC1, EC2, EC3, EC4} is not yet the minimal set of ECs in the network. In fact, EC1 and EC3 can be merged because the forwarding behaviors of both ECs are identical at any device in the network, as depicted in Figure 3a that EC1 and EC3 share the same forwarding graph.

We finish the model abstraction by generating a new set of forwarding rules that are to be installed on the "big switch". To make the process more efficient, we only have to consider those ECs whose packets traverse edge switches in the network.

Table 2 shows the resulting rules that will be installed in the big switch, i.e., SW in Figure 4. The resulting one big switch network has the identical forwarding functions to the original tree network from the end-to-end communication perspective. The number of switches we need to simulate or emulate is now reduced from four to one, and the number of rules in the network is reduced from twelve to four. If we only consider OpenFlow rules that match the NW_DST field and the action is always forwarding, then the total number of rules in the big switch is proportional to the number of ECs, whereas in the original SDN network, the total number of rules is $O(S \times P)$, where S is the number of switches and P is the number of address prefixes.

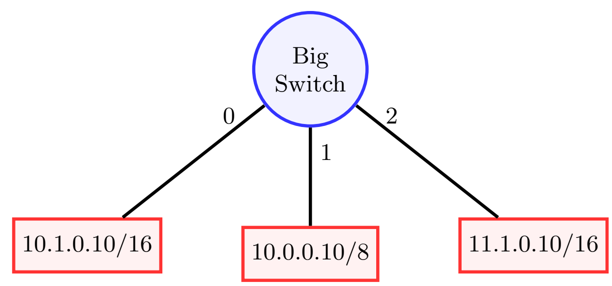


Figure 4: Compressed SDN Network for Scalable Simulation

3. SDN MODEL ABSTRACTION

Our objective is to effectively transform a static SDN data plane configuration (i.e., a snapshot of the network state) to one "big switch" model, which preserves the same end-to-end forwarding behavior. To achieve this objective, we need to identify how every packet is processed in the snapshot, and how to correctly configure the big switch model to reflect the identical forwarding logic. In this paper, we develop a three-step model abstraction method, which is summarized as follows.

- **Identifying Equivalence Classes.** We partition all possible packets in the network into mutually exclusive sets (i.e., equivalence class, as formally defined in Section 3.1), and the packets belongs to the same set are

Table 1: Forwarding Rules on Each OpenFlow Switch in the 3-ary Tree Network

Switch	Priority	Match Field	Action
SW0	10	NW_DST=10.1.*.*	FWD: OUT_PORT=0
	1	NW_DST=10.*.*.*	FWD: OUT_PORT=1
	1	NW_DST=11.1.*.*	FWD: OUT_PORT=2
SW1	10	IN_PORT=1, NW_DST=10.1.*.*	FWD: OUT_PORT=0
	1	IN_PORT=0, NW_DST=10.*.*.*	FWD: OUT_PORT=1
	1	IN_PORT=0, NW_DST=11.1.*.*	FWD: OUT_PORT=1
SW2	10	IN_PORT=0, NW_DST=10.1.*.*	FWD: OUT_PORT=1
	1	IN_PORT=1, NW_DST=10.*.*.*	FWD: OUT_PORT=0
	1	IN_PORT=0, NW_DST=11.1.*.*	FWD: OUT_PORT=1
SW3	10	IN_PORT=1, NW_DST=11.1.*.*	FWD: OUT_PORT=0
	1	IN_PORT=0, NW_DST=10.*.*.*	FWD: OUT_PORT=1
	1	IN_PORT=0, NW_DST=10.1.*.*	FWD: OUT_PORT=1

Table 2: Forwarding Rules on the "Big OpenFlow Switch"

Switch	Priority	Match Field	Action
SW	10	NW_DST=10.0.*.*	FWD OUT_PORT=1
	10	NW_DST=10.1.*.*	FWD OUT_PORT=0
	10	NW_DST=11.1.*.*	FWD OUT_PORT=2
	10	NW_DST=10.2.0.0-10.255.255.255	FWD OUT_PORT=1

processed in the same way. Those sets are identified according to the matching field of **all** the OpenFlow rules on **all** the SDN switches in the original network.

- **Creating Forwarding Graphs.** We model the forwarding behavior of each packet set using the topology information as well as the local information stored on SDN switches (e.g., port mapping, rule priorities, etc), and generate a graph-based model to represent the forwarding behavior.

- **Generating OpenFlow Rules of the Big Switch.** We generate the OpenFlow rules for the big switch in order to preserve the end-to-end forwarding logic. This step includes (1) constructing the port-to-host mapping, (2) generating the rules by matching the packet header of each set, and (3) forwarding the packet to the correct output port, which is determined by traversing the forwarding graph acquired in step 2.

Our three-step approach has two assumptions. First, the controller can dynamically change the configuration of each network device, but we assume that the frequency of issuing such control messages is far less than the rate of the incoming packets. Between two configuration updates, the data plane remains unchanged. Therefore, we can exclude the SDN controller from the abstracted network model. Second, we do not consider packet header modification actions on the network device. We describe each step in details in the remainder of the section.

3.1 Identifying Equivalence Classes

We first give the definition of equivalence class (EC), and then present the data structure and algorithms to partition the packets into ECs.

Definition 1. An equivalence class is a set of packets that experience identical forwarding action at **any** network device in the network.

Each packet is uniquely identified by its header field values, which are matched against the forwarding rules in the OpenFlow switches to determine the appropriate action. Since the matching fields of the OpenFlow rules typically contain the wildcard suffix (e.g., longest prefix match of IP source/destination addresses), a group of packets with consecutive header values are often processed by the same rule. We use a trie structure, originally proposed by VeriFlow [19], to maintain the matching fields of all the OpenFlow rules in the network. The trie is composed of several sub-tries, and each sub-trie stands for a matching field (e.g., source/destination MAC address/IP address/port, etc.). Each node in sub-trie presents one bit in the corresponding matching field, and each node has three edges to the next node (i.e., next bit in the matching field). The edges represent three possible bit-to-bit rule matching conditions: zero, one, or wildcard (i.e., don't care). The rule metadata are stored in the corresponding leaf node, including the rule's location (switch#), action (forwarding to an out port or dropping the packet), priority, etc.

Having all the OpenFlow rules inserted in the aforementioned trie structure, we perform the following three steps to identify the equivalence classes in the network. (1) We traverse the trie to obtain the consecutive header values for each rule. (2) After having a collection of header value intervals, which are denoted by the starting and end values, we develop an algorithm to split the existing intervals into smaller and non-overlapping intervals. Each non-overlapping interval identifies the packets belonging to an equivalence class. (3) We merge certain equivalence classes in order to reduce the time and space complexity for the forwarding graph gen-

ALGORITHM 1: Splitting Overlapping Intervals

Data: I = a set of packet header intervals from the leaves of the trie

Result: EC = a set of equivalence classes as disjoint intervals

1 EC has special meaning as equivalence class. Please use another symbol. $cnt \leftarrow 0$
2 $S = \{\text{starting points of } \forall i \in I\}$, $E = \{\text{end points of } \forall i \in I\}$
3 $A \leftarrow Sort(S \bigcup E)$ in a non-decreasing order
4 $EC \leftarrow \emptyset$
5 **foreach** $x \in A$ **do**
6   **if** $x \in S$ **then**
7   **if** $cnt \neq 0$ **then**
8       $EC \leftarrow EC \bigcup [prev, x-1]$
9   **end**
10     $prev \leftarrow x$
11     $cnt \leftarrow cnt + 1$
12   **else** $x \in E$
13     $EC \leftarrow EC \bigcup [prev, x]$
14     $prev \leftarrow x + 1$
15     $cnt \leftarrow cnt - 1$
16   **end**
17 **end**

eration (Section 3.2) and the big switch rule generation (Section 3.3). The details of the second and third steps for EC identification are presented as follows.

3.1.1 Splitting Overlapping Intervals

By traversing from the root node to all leaf nodes, we obtain a set of packet header intervals that match all the rules along the traversal. Each interval is represented by a pair of starting and ending values as A, B and C as shown in Figure 5. We split this set of intervals, I, to a list of non-overlapping intervals, each of which forms an EC. We develop Algorithm 1 to generate a set of disjoint intervals, and show that the generation can be accomplished in $O(N \times M \log M)$ time, where M is the number of intervals in I, and N is the number of header bits.

First, we place I into an array A of $2M$ elements. Each element is either a starting point or an end point of an interval. We visit each point in a sorted order, and maintain the difference d between the number of visited starting points and the number of visited end points.

- If the current element $x \in S$ and $d > 0$, we end the previous interval with the ending value $x - 1$; Start a new interval with a starting value x (line 7-9).
- If the current element $x \in E$, we end the previous interval with the ending value x. (line 13).
- In either case, we update the potential new interval's starting value *prev* (line 10 and 14).

Please define S and E, and "end an interval" or "start an interval" does not sound right to me, usually we start/end an action.

Updating the network forwarding rules will change the EC set. By maintaining the rules in a trie, we can efficiently update ECs in an incremental way. An insertion of a new rule requires us to do a depth first traversal. This process automatically narrows down the set of affected rules by ignoring those non-overlapping branches with the new rule. The output is the set of affected intervals, and we can run Algorithm 1 to update only those affected ECs.

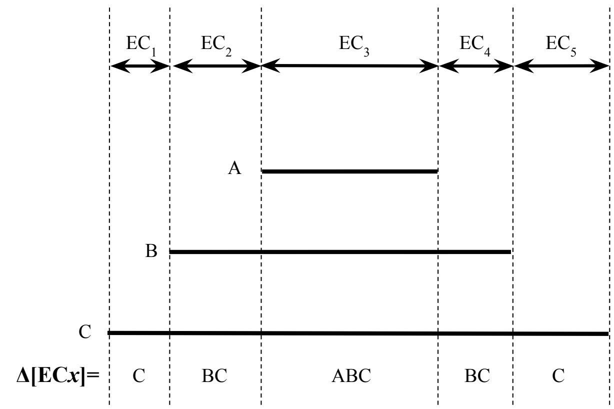


Figure 5: A set of packets are identified by an interval of packet header values. Five equivalence classes, EC_1 to EC_5, can be obtained via splitting three intervals A, B and C. Finding $\Delta[EC_x]$ (i.e., the rules that intersect with EC_x) is instrumental for merging ECs, as shown in the bottom of the figure.

3.1.2 Combining Equivalence Classes

We can further union certain ECs obtained from Algorithm 1, if they essentially represent identical packet forwarding behavior (see the definition of ECs). For example, EC_2 and EC_4 in Figure 5 can be combined as one EC, since the packets in both ECs experience the same set of forwarding rules in the network.

LEMMA 1. *If packets in EC α and EC β experience the same forwarding actions on all network devices, then $\alpha \cup \beta$ is also an EC.*

Combining two EC into a single one reduces the running time in the next two phases, i.e., generating the forwarding graph and populating the final OpenFlow rules. The number of the resulting forwarding rules in the "big switch" can also be reduced. We present the following lemma to identify whether two ECs can be unioned.

LEMMA 2. *EC α and EC β can be unioned into one EC, if both packet header values are covered by the same set of rules in the network.*

For example, both EC_2 and EC_4 are covered by interval B and C in Figure 5, and therefore, we can treat them as one EC. The explanation is illustrated below.

First, we define a function $\Delta(x)$ that maps an EC x to a set of forwarding rules, whose matching fields cover the header values of all the packets in x. Assume $\Delta(\alpha) = \Delta(\beta)$, and let $\delta \in \Delta(\alpha)$ be the rule on a network device d with the highest priority. If no such δ exists, packets from both α and β are dropped on d. Otherwise, packets in both α and β match the rule δ and are processed with the same action specified in δ. Note that in another device d', the highest priority rule that covers both α and β may be different, i.e., $\delta' \neq \delta$. However, as long as δ is unique at a given d, the forwarding behavior at d for both α and β are always identical.

Given an EC, we can efficiently calculate $\Delta(\alpha)$ using two data structures: an array of pointers and a central interval tree. Each of them is responsible for one of the two cases specified in [2].

- Case 1: A rule δ overlaps with an EC α with its starting and/or end point in α. We can reuse the sorted array A in Algorithm 1. We augment each value, either a starting point or an end point of an interval in A with a pointer to the corresponding rule. By doing a binary search, we can find the minimum and maximum values in A, which bound the interval of α. Therefore, we can ignore two types of rules: the ones with end points smaller than the minima and the ones with starting point larger than the maxima. We then perform a linear search in the new set of rules, and check one-by-one whether the interval overlaps with α. The total time complexity for both the linear search and the binary search are $O(\log M + K)$, where K is the number of reported intervals in $\Delta(\alpha)$.

- Case 2: Rule δ covers α entirely. We can build a central interval tree [10] with all the available intervals. We pick a random value $x \in \alpha$ and query the central interval tree for all the ranges that intersect with x, which can be done in $O(\log M + K)$ time. It takes $O(MlogM)$ time to build the central interval tree. Since the central interval tree supports efficient incremental operations (i.e., insertion and deletion), our design also supports dynamic changes of the rule set.

Using the interval tree and the ordered list, for each EC α, we calculate $\Delta(\alpha)$ by mapping each rule $\delta \in \Delta(\alpha)$ to a unique binary ID c_δ of length $\log_2 M$. We can encode $\Delta(\alpha)$ to a string of c_δs, starting with small IDs. This string of unique IDs, named C_α, has a $M \log_2 M$ upper bound in length. We then use a hash table H to combined the ECs by hashing each EC x to C_x. The minimal size of ECs is the number of unique keys in H. Note that in the subsequent algorithmic designs, iterating through all ECs refers to iterating through the first ECs in each set $H[key]$.

3.2 Generating Forwarding Graphs

In the second step, we compute a forwarding graph for each EC, and then effectively reduce the size of the forwarding graph to improve efficiency for the third step.

First, we define a function $FG(\alpha)$ that maps an EC α to a corresponding forwarding graph. A forwarding graph is a directed graph that represents how packets belonging to the same EC are processed by the network. A node u in the forwarding graph is a networking device, and an edge (u, v) in the graph means that device u forwards the packets to device v in the network. A forwarding graph not only concatenates the forwarding behavior for each EC, but also visualizes the data flow of the EC in the network. Since our objective is to abstract the network forwarding logic into a big switch, our end-to-end modeling focuses on the sources and sinks of the graph. Figure 6 depicts the generalized forwarding graph $FG(x)$ for EC x.

3.2.1 Network Traversal for Forwarding Graph Generation

We develop a forwarding graph generation algorithm as shown in Algorithm 2. The notations are defined as follows. $FG(x)$ denotes the forwarding graph for a particular equivalence class x. A *edge switch* is defined as a switch that has at least one link to a node outside SN. SN first appeared here, need definition. A *non-edge switch* is defined as a switch with all connected nodes inside SN. The forwarding behavior of the non-edge switches are not considered in the big switch model abstraction. Let src and snk denote the source and sink nodes of the forwarding path for an EC. Note that all src in $FG(x)$ are edge switches, and snk in $FG(x)$ can be either edge switches or non-edge switches. Let $curr$ denote the current traversed node in the network. We add a super-source node, SRC^x, and a super-sink node, SNK^x, as the boundaries of $FG(x)$.

Algorithm 2 is designed to generate $FG(x)$. We start the process from each src that connects to SRC^x, and then traverse EC x's forwarding graph using a depth-first-based search and follow the action specified in the forwarding rule with the highest priority for EC x at each node along the traversal.

We distinguish two kinds of port on an edge switch:

- *end port* that connects to a node that is either the forwarding end point or outside the target network;

- *inner port* that connects to a node inside the target network.

We add an edge from SRC^x to a src, if the source node has a forwarding rule r that matches EC x, or the IN_PORT field of rule r on the source node is an end port. Otherwise, we do not initiate a traverse (see line 22 to 27 in Algorithm 2). Correspondingly, we add an edge from a snk to the super sink SNK^x, if the following two conditions are satisfied:

1. the sink node is an edge switch in the network;

2. the OUT_PORT field determined by the rule's action on the sink node is an end port.

3.2.2 Network Traversal Outcomes

After running Algorithm 2, we can discover three kinds of "paths" in $FG(x)$ that are useful for the forwarding rule generation process for the big switch model, i.e., the third step of our model abstraction process (see Section 3.3).

- **Forwarding path** (line 10-14). The path from the super source node to the super sink node. This is a normal forwarding path for packets in EC x.

- **Dropping packets in the network** (line 4-8). The path ends at a device inside the network, and fails to reach the super sink node. This indicates that the packets in EC x are dropped inside the network.

- **Forwarding loop** (line 18). There is a directed cycle in the graph. One can simulate a forwarding loop in the network by (1) adding a rule in the big switch to drop the looping packets; or (2) dynamically monitoring the volume of the looping packets and adjusting the delay of looping packets and other packets sharing the communication path. We choose the first method since the model abstraction in the paper is focus on the forwarding logic equivalence, and will leave the second method as future work when investigating end-to-end performance equivalence.

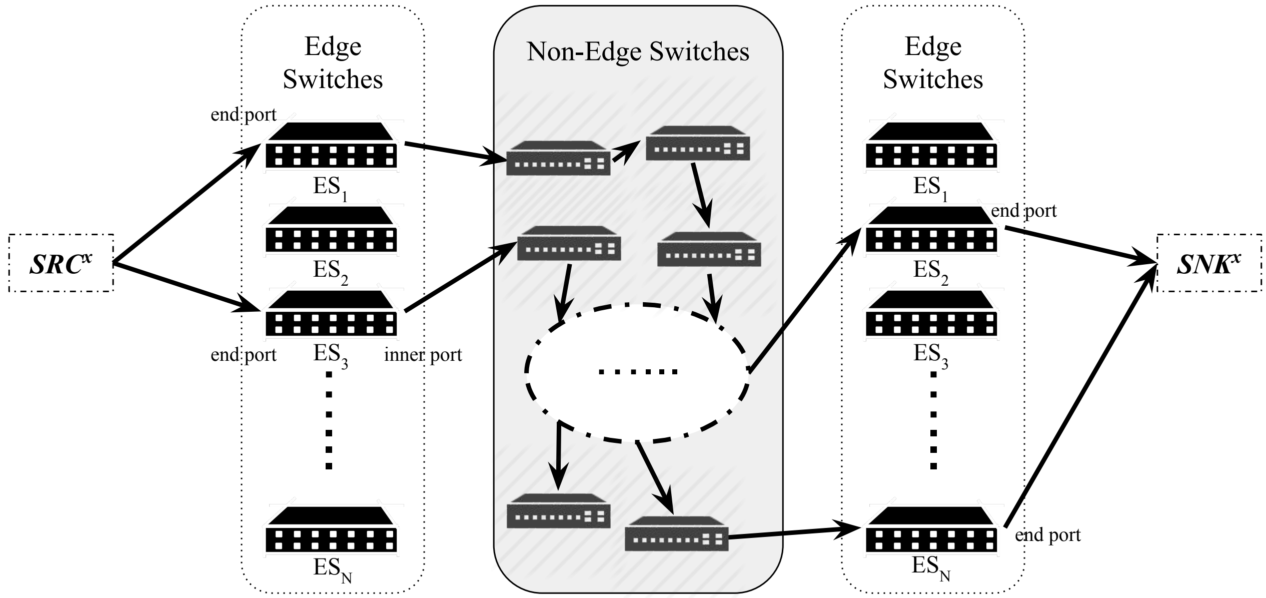


Figure 6: Modeling a Forwarding Graph of an Equivalence Class

3.3 Populating Flow Tables on the Big Switch Model

We develop an algorithm to generate OpenFlow rules on the big switch to abstract the forwarding behavior (see Algorithm 3). We maintain a hash table $PortMap$ to map the end ports of the edge switches to the ports of the big switch. This table is configured using the *global_port* variable during the rule generation procedure. Algorithm 3 generates the mandatory fields in an OpenFlow rule:

- The $MATCH$ field is given by the EC x, i.e., the range of matching packets header (line 2);
- The IN_PORT field is the mapped port number of *src.port* (line 6);
- Depending on the *dst* port, we generate either a packet drop action (line 8) or a packet forwarding action with the appropriate mapped port number of *dst.port* (line 10-14).

4. EVALUATION

We perform experimental evaluation of our network model abstraction technique that transforms an SDN-based network to one big switch. The evaluation results show that our approach significantly saves simulation/emulation resources (e.g., number of forwarding rules) and simulation execution time, while still preserving the forwarding behavior of the original network.

Our experiments simulate and emulate networks of type tree topology. The tree network is described by two topological parameters: depth d and fanout f. Such network $tree(d, f)$ can connect f^d hosts with $\frac{f^d-1}{f-1}$ switches in total. All end-hosts in a tree network are fully-connected with at most $2d$ hops.

4.1 Network Forwarding Logic Equivalence

We first demonstrate that the forwarding logic of the original software-defined network is exactly preserved by the abstracted big switch model. We created a tree-topology network net_1 in Mininet [13], and connected all the switches to an SDN controller running a layer-two learning switch application [5]. After performing the `ping` tests between randomly selected pairs of end-hosts, the controller application generated all the network forwarding rules and installed them on the switches. We then took a snapshot of the network, including (1) the host-to-switch and switch-to-switch connections, and (2) the rules on all the switches using the `ovs-ofctl dump-flows` command. The snapshot was used to generate the rules for the big switch model as well as the port mapping according to the algorithms presented in Section 3.

We then created another emulated network net_2 in Mininet, consisting of one OpenFlow switch and the same number of hosts as net_1. The switch was connected to f^d hosts with the port numbers derived from both the $PortMap$ (Algorithm 3) and the link information (net_1's topology). The rules generated by Algorithm 3 were installed on the switch using the `ovs-ofctl add-flow` command.

To validate that the big-switch-network preserved the network forwarding logic of the original network, we recorded the connectivity between *every* host pair in both net_1 and net_2, and compared the results. Specially, the original network net_1 is a tree network with f^d hosts, where d is the depth and f is the fanout of a tree network. Each host sent a number of `ping` packets to every other host in net_1, and the amount of packet was randomly selected between 1 and 10. We repeated the experiments in net_2 with the same traffic pattern. The result was represented in a matrix $\mathcal{R}$, where $\mathcal{R}[i][j]$ denotes the numbers of successfully received `ping` packets from host i to host j, where $i \neq j$, and $i, j \in [1, f^d]$.

We repeated the experiment for different combinations of d and f, i.e., $(d, f) \in \{(2, 3), (2, 4), (3, 3), (3, 4), (4, 3)\}$. For each network scenario, we saved the experimental results in $\mathcal{R}_1$ and $\mathcal{R}_2$, and compared the two matrices using

ALGORITHM 2: Generating a Forwarding Graph for EC x

```
Input: nodes = Switches containing rules for EC x
       topo = Network topology
Result: Forwarding graph FG(x) for EC x
Function traverse(curr, src, snk)
    if curr is NOT visited then
        r ← highest-priority rule on curr that processes EC x
        if r is NULL or r.action is DROP then
            snk ← (curr, NULL)
            generate_rules(x, src, snk)
            return
        end
        next ← topo[curr][r.action.outport]
        if next ∉ nodes then
            snk ← (curr, r.action.outport)
            generate_rules(x, curr, src, snk)
            return
        end
        mark curr as visited
        traverse(next, src, snk)
    else
        report forwarding loop
    end
return

foreach n ∈ neighbors of SRC^x do
    if n is NOT visited then
        inport ← input port number from SRC^x to n
        traverse(n, src =(n, inport), snk =NULL)
    end
end
```

ALGORITHM 3: Generating Forwarding Rules for EC x on the Big Switch Model

```
Data: PortMap, which maps a port on sw to a port on the big
      switch
      global_port, for port number assignment, and is
      initialized to 0
Result: A new rule r to install on the big switch
Function generate_rules(x, src, dst)
    r.match ← x
    if src.port ∉ PortMap[src.sw] then
        PortMap[src.sw][src.port] ← global_port++
    end
    r.inport = PortMap[src.sw][src.port]
    if dst.port is NULL then
        r.action ← drop_action
    else
        if dst.port ∉ PortMap[dst.sw] then
            PortMap[dst.sw][dst.port] ← global_port++
        end
        r.action ← forward_action
        r.action.outport ← PortMap[dst.sw][dst.port]
    end
return
```

the `diff` command. We found that $\mathcal{R}_1 = \mathcal{R}_2$ holds true for all five network scenarios. We visualized $\mathcal{R}_1$ and $\mathcal{R}_2$ for the $(d = 2, f = 3)$ and $(d = 4, f = 3)$ cases in Figure 7. We can see that the original SDN-based network and the abstracted one-big-switch-based network have the identical network forwarding logic, measured by the connectivity and the number of receiving packets for each connection. Note that the brightness of the element in the matrix is proportional to $\mathcal{R}[i][j]$, i.e., the number of successfully delivered packets from host i to host j.

4.2 Performance Gain

Number of OpenFlow Rules. We compare the total number of rules installed on the switches in both net_1 and net_2 with the same experimental settings in Section 4.1. The results are plotted in Figure 8 for networks with various topological parameter settings. The number of rules needed to preserve the forwarding logic is significantly less in the one-big-switch-based network as compared with the original SDN-based network for all scenarios in the range of x% to y% reduction. For example, in the case of a network with depth = 4, and fanout = 3, 53,260 rules in the original network were reduced to 5,766 rules in the big-switch-based network.

Simulation Time. Our approach significantly reduces network simulation model complexity in terms of the number of switches and the number of rules. A key benefit is to reduce the time to run simulation experiments.

We performed the same set of experiments on a network simulator, S3FNet [21]. We simulated two SDN-based networks: one models a tree-topology network $net_1(d, f)$, and the other models the corresponding big-switch-based network net_2. We set half of the hosts as TCP clients and the other half as TCP servers, and conducted one-to-one

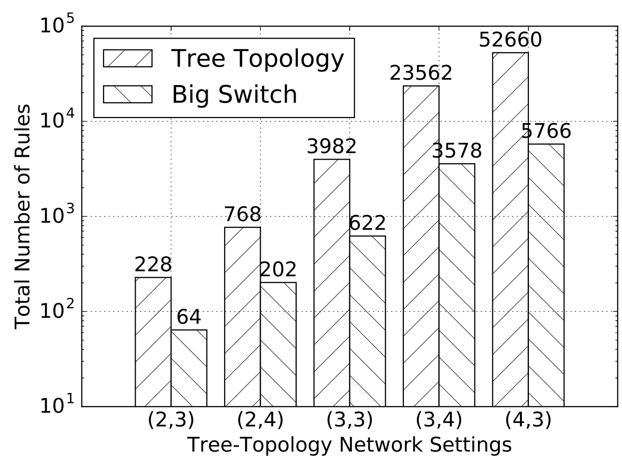


Figure 8: Number of rules needed to preserve the network forwarding logic. The number of rules on the big switch is about x% to y% less than the number of rules in the original tree-topology network. The x-axis label (d, f) represents the depth and fanout parameters in a tree topology network.

communication among them. We sent each traffic flow for 100 seconds in simulation time. We repeated each experiment ten times and recorded the simulation execution time for both net_1 and net_2 in Figure 9 for comparison. modify Figure 9, y-axis, simulation execution time or simulation running time. The error bars indicate the standard deviations of the running time for all ten independent simulation runs. We can see that simulating the big-switch-based network is 4.63 give a range times faster than simulating the original SDN-based network.

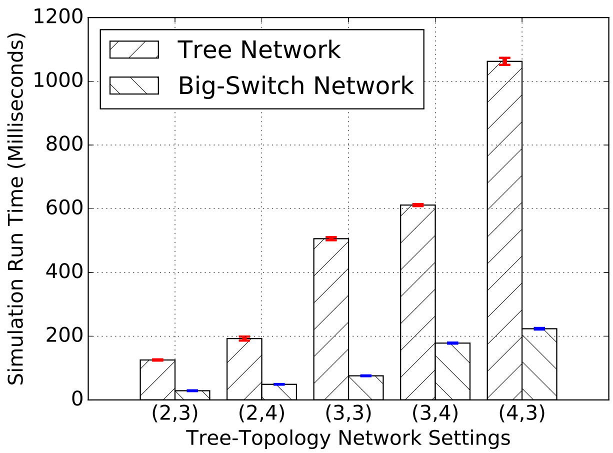


Figure 9: Comparison of simulation execution time. The big-switch-based network model saves about 80% give a range running time as compared to simulating the corresponding SDN-based network. The x-axis label (d, f) represents the depth and fanout parameters in a tree topology network.

4.3 Model Abstraction Execution Time

We discussed the asymptotic time complexity of our model abstraction technique in Section 3. We now evaluate the execution time for transforming an SDN-based network to a big-switch-based network. We recorded the running time for converting various tree networks, i.e., $(d, f) \in \{(2, 3), (2, 4), (3, 3), (3, 4), (4, 3)\}$ in Figure 10. We can see that the model abstraction process is lightweight. For example, it took 3.15 seconds to abstract a medium-scale tree network (depth 4 and fanout 3) that consists of 53,260 rules (maybe give another example for the smallest tree network). The fast model abstraction execution time is useful. As the network state keeps evolving, it is essential to constantly update the abstracted big switch model to reflect the changes, preferably in an online fashion. In fact, the three-step approach allows us to incrementally update the big switch model and requires far less execution time, i.e., we only need to update a small set of rules that are different in the new network snapshot.

about the model abstraction execution time figure, why the x-axis and y-axis use log2? I thought no log or log 10 is better. even with log2, the y value should be 1, 2, 4, 8; and x-axis should be X, not log2(X); let us remove the blue line (overhead trend) as for now. In fact, I think the execution time does not increase linearly as the number of rules grows according to the plot.

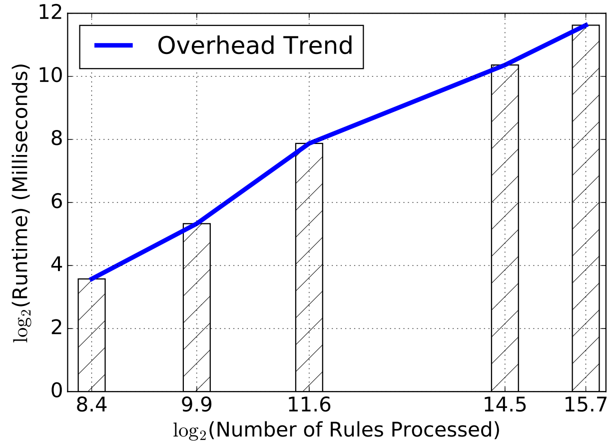


Figure 10: Execution time to transform an SDN-based network to a big-switch-based network.

5. RELATED WORK

5.1 SDN Forwarding Rules Abstraction

The "one big switch" concept was proposed in [17] for a different purpose. In their work, the network abstraction is used to reduce conflicting rules generated by various high-level SDN applications, which often simultaneously run on one or even multiple controllers. Application developers only need to specify the end-to-end connectivity policies on the big switch, and the SDN infrastructure is responsible for hop-by-hop routing and rule placement mechanisms. Our work applies the concept in a reversed direction for enhancing the scalability of network simulation and emulation. We collect the forwarding rules on each device and use the rules to correctly infer the end-to-end forwarding policies. Our model abstraction approach is based on statically analyzing snapshots of the network state. There exists a line of research on network fault detection by analyzing software, configuration and network-wide data-plane state [6, 7, 20, 22]. Those approaches typically operate offline on timescales of seconds to hours. Real time network verification tools are developed to enforce correctness in connectivity [18, 19]. Our work leverages the idea of slicing the entire network into equivalence classes in [19] to reduce the problem space, which enables fast model abstraction execution speed.

5.2 SDN Emulation and Simulation

There are a number of SDN emulation and simulation testbeds based on the OpenFlow protocol. Examples include Mininet [13], EstiNet [1], ns-3 [3], S3FNet [16], fs-sdn [12] and OpenNet [9]. Mininet [13] applies container-based virtualization technique and cgroup based resource isolation to provide a lightweight and high fidelity emulation platform. Its functional fidelity is guaranteed by executing real SDN switch/controller software. ns-3 [3] offers simulation models of SDN networks and emulation of SDN controllers via the direct code execution (DCE) technique. S3FNet [16] is a hybrid OpenFlow-based SDN testing platform that integrates a parallel network simulator with an OpenVZ-based network emulator. fs-sdn [12] extends fs, a flow-level discrete event network simulator, with the SDN capability. We develop a model abstraction method in this paper to transform a large scale and complicated SDN network to a one-big-switch-based network. We can use the resulting abstracted network model in all the aforementioned simulation and emulation environment for performance gain while still preserving the network forwarding logic.

6. CONCLUSION AND FUTURE WORK

We present a three-step model abstraction technique to transform an SDN-based network to a one-big-switch based network without losing the forwarding behavior as defined by the OpenFlow rules in the network devices. Experimental results demonstrate that the big switch abstraction correctly models the end-to-end forwarding logic of the original SDN network, and the abstracted model significantly saves the experiment running time and resources. The ultimate goal of the one-big-switch abstraction is to enhance simulation and emulation scalability while preserving packet-level fidelity. This paper mainly focuses on the end-to-end forwarding logic equivalence, and we will investigate end-to-end performance equivalence, such as latency and packet drop in the future.

7. REFERENCES

[1] Estinet. http://www.estinet.com/. Accessed: 2016-12-31.

[2] Interval tree. https://en.wikipedia.org/wiki/Interval_tree#With_an_interval. Accessed: 2016-12-20.

[3] ns-3. https://www.nsnam.org. Accessed: 2016-12-29.

[4] Software defined network OPNET simulation SDN. https://www.youtube.com/watch?v=MPpu8blePfc. Accessed: 2016-12-29.

[5] The POX controller. https://github.com/noxrepo/pox. Accessed: 2016-12-31.

[6] E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *Proc. of the 3rd ACM Workshop on Assurable and Usable Security Configuration (SafeConfig)*, pages 37–44, 2010.

[7] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *Proc. of the 17th IEEE International Conference on Network Protocols (ICNP)*, pages 123–132, 2009.

[8] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey, and G. Wang. Meridian: an SDN platform for cloud network services. *IEEE Communications Magazine*, 51(2):120–127, February 2013.

[9] M. C. Chan, C. Chen, J. X. Huang, T. Kuo, L. H. Yen, and C. C. Tseng. OpenNet: A simulator for software-defined wireless local area network. In *Proceedings of the 2014 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 3332–3336, April 2014.

[10] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. Interval tree. *Computational Geometry*, pages 220–226, 2008.

[11] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett. SDX: A software defined internet exchange. In *Proceedings of ACM SIGCOMM*, pages 551–562, 2014.

[12] M. Gupta, J. Sommers, and P. Barford. Fast, accurate simulation for sdn prototyping. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 31–36, New York, NY, USA, 2013. ACM.

[13] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[14] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, pages 253–264, New York, NY, USA, December 2012. ACM.

[15] 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 *Proceedings of ACM SIGCOMM*, pages 3–14, 2013.

[16] D. Jin, Y. Zheng, and D. Nicol. S3F/S3FNet: Simpler scalable simulation framework. University of Illinois at Urbana-Champaign. https://s3f.iti.illinois.edu/, 2013.

[17] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 13–24, New York, NY, USA, 2013. ACM.

[18] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 99–112, Berkeley, CA, USA, 2013. USENIX Association.

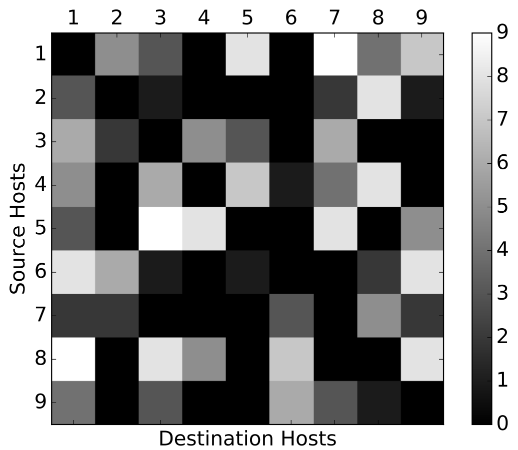
[19] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'13, pages 15–27, Lombard, IL, 2013.

[20] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Proc. of ACM SIGCOMM Computer Communication Review*, SIGCOMM'11, pages 290–301, New York, NY, USA, 2011. ACM.

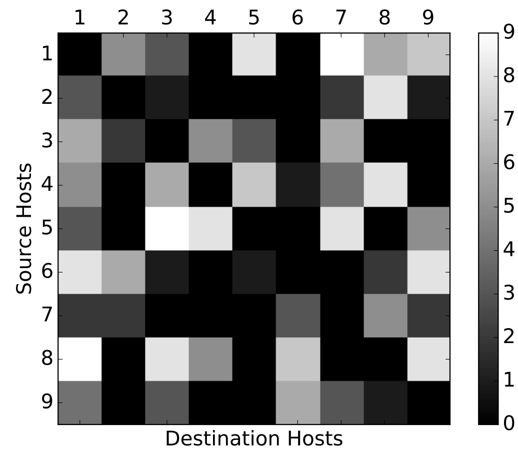
[21] D. M. Nicol, D. Jin, and Y. Zheng. S3F: The scalable simulation framework revisited. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 3288–3299, 2011.

[22] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of IP networks. In *Proc. of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*

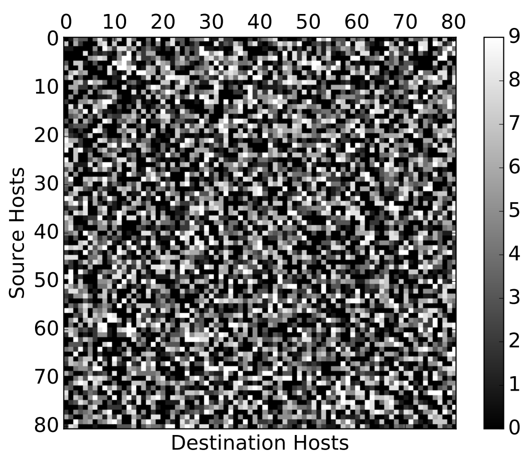
(INFOCOM), volume 3, pages 2170–2183. IEEE, March 2005.



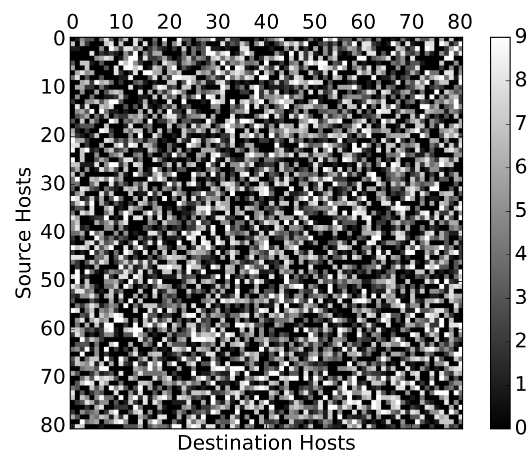
(a) $\mathcal{R}_1$, packets received per host in $net_1(d = 2, f = 3)$



(b) $\mathcal{R}_2$, Packets received in $net_2(d = 2, f = 3)$



(c) $\mathcal{R}_1$, Packets received in $net_1(d = 4, f = 3)$



(d) $\mathcal{R}_2$, Packets received in $net_2(d = 4, f = 3)$

Figure 7: Matrix $\mathcal{R}$ represents the number of packets received at each host. net_1 is the original SDN network with a tree topology (d, f), where d is depth and f is fanout. net_2 is the corresponding one-big-switch-based network. The gradient legend visualizes the number of received packets. $\mathcal{R}_1$ and $\mathcal{R}_2$ are identical, which indicate that our model abstraction technique preserves the network forwarding logic.