VeriFlow: Verifying Network-Wide Invariants in Real Time

Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, P. Brighten Godfrey
University of Illinois at Urbana-Champaign
{*khurshi1, xuanzou2, wzhou10, caesar, pbg*}*@illinois.edu*

Abstract

Networks are complex and prone to bugs. Existing tools that check network configuration files and the data-plane state operate offline at timescales of seconds to hours, and cannot detect or prevent bugs as they arise.

Is it possible to *check network-wide invariants in real time*, as the network state evolves? The key challenge here is to achieve extremely low latency during the checks so that network performance is not affected. In this paper, we present a design, VeriFlow, which achieves this goal. VeriFlow is a layer between a software-defined networking controller and network devices that checks for network-wide invariant violations dynamically as each forwarding rule is inserted, modified or deleted. VeriFlow supports analysis over multiple header fields, and an API for checking custom invariants. Based on a prototype implementation integrated with the NOX OpenFlow controller, and driven by a Mininet OpenFlow network and Route Views trace data, we find that VeriFlow can perform rigorous checking within hundreds of microseconds per rule insertion or deletion.

1 Introduction

Packet forwarding in modern networks is a complex process, involving codependent functions running on hundreds or thousands of devices, such as routers, switches, and firewalls from different vendors. As a result, a substantial amount of effort is required to ensure networks' correctness, security and fault tolerance. However, faults in the network state arise commonly in practice, including loops, suboptimal routing, black holes and access control violations that make services unavailable or prone to attacks (e.g., DDoS attacks). Software-Defined Networking (SDN) promises to ease the development of network applications through logically-centralized network programmability via an open interface to the data plane, but bugs are likely to remain problematic since the complexity of software will increase. Moreover, SDN allows multiple applications or even multiple users to program the same physical network simultaneously, potentially resulting in conflicting rules that alter the intended behavior of one or more applications [25].

One solution is to rigorously check network software or configuration for bugs prior to deployment. Symbolic execution [12] can catch bugs through exploration of all possible code paths, but is usually not tractable for large software. Analysis of configuration files [13, 28] is useful, but cannot find bugs in router software, and must be designed for specific configuration languages and control protocols. Moreover, using these approaches, an operator who wants to ensure the network's correctness must have access to the software and configuration, which may be inconvenient in an SDN network where controllers can be operated by other parties [25]. Another approach is to statically analyze snapshots of the network-wide data-plane state [9, 10, 17, 19, 27]. However, these previous approaches operate offline, and thus only find bugs after they happen.

This paper studies the question, *Is it possible to check network-wide correctness in real time as the network evolves?* If we can check each change to forwarding behavior before it takes effect, we can raise alarms immediately, and even prevent bugs by blocking changes that violate important invariants. For example, we could prohibit changes that violate access control policies or cause forwarding loops.

However, existing techniques for checking networks are inadequate for this purpose as they operate on timescales of seconds to hours [10, 17, 19]. [1] Delaying updates for processing can harm consistency of network state, and increase reaction time of protocols with real-time requirements such as routing and fast failover; and processing a continuous stream of updates in a large

[1] The average run time of reachability tests in [17] is 13 seconds, and it takes a few hundred seconds to perform reachability checks in Anteater [19].

network could introduce scaling challenges. Hence, we need some way to perform verification at very high speeds, i.e., within milliseconds. Moreover, checking network-wide properties requires obtaining a view of network-wide state.

We present a design, VeriFlow, which demonstrates that the goal of real-time verification of network-wide invariants is achievable. VeriFlow leverages software-defined networking (SDN) to obtain a picture of the network as it evolves by sitting as a layer between the SDN controller and the forwarding devices, and checks validity of invariants as each rule is inserted, modified or deleted. However, SDN alone does not make the problem easy. In order to ensure real-time response, VeriFlow introduces novel incremental algorithms to search for potential violation of key network invariants — for example, availability of a path to the destination, absence of forwarding loops, enforcement of access control policies, or isolation between virtual networks.

Our prototype implementation supports both OpenFlow [21] version 1.1.0 and IP forwarding rules, with the exception that the current implementation does not support actions that modify packet headers. We microbenchmarked VeriFlow using a stream of updates from a simulated IP network, constructed with Rocketfuel [7] topology data and real BGP traces [8]. We also evaluated its overhead relative to the NOX controller [14] in an emulated OpenFlow network using Mininet [3]. We find that VeriFlow is able to verify network-wide invariants within hundreds of microseconds as new rules are introduced into the network. VeriFlow's verification phase has little impact on network performance and inflates TCP connection setup latency by a manageable amount, around 15.5% on average.

We give an overview of data plane verification and SDN (§ 2) before presenting VeriFlow's design (§ 3), implementation (§ 4), and evaluation (§ 5). We then discuss future (§ 6) and related work (§ 7), and conclude (§ 8).

2 Overview of Approach

VeriFlow adopts the approach of *data plane verification*. As argued in [19], verifying network correctness in the data plane offers several advantages over verifying higher-level code such as configuration files. First, it is closely tied to the network's actual behavior, so that it can catch bugs that other tools miss. For example, configuration analysis [13, 28] cannot find bugs that occur in router software. Second, since data-plane state has relatively simple formats and semantics that are common across many higher-layer protocols and implementations, it simplifies rigorous analysis of a network.

Early data plane verification algorithms were developed in [27], and systems include FlowChecker [9], Anteater [19], and Header Space Analysis [17]. The latter two systems were applied to operational networks and uncovered multiple real-world bugs, validating the data plane analysis approach. However, as noted previously, these are offline rather than real-time systems.

VeriFlow performs real-time data plane verification in the context of software defined networks (SDNs). An SDN comprises, at a high level, (1) a standardized and open interface to read and write the data plane of network devices such as switches and routers; (2) a *controller*, a logically centralized device that can run custom code and is responsible for transmitting commands (forwarding rules) to network devices.

SDNs are a good match for data plane verification. First, a standardized data plane interface such as OpenFlow [6] simplifies unified analysis across all network devices. Second, SDNs ease *real-time* data plane verification since the stream of updates to the network is observable at the controller.

SDN thus simplifies VeriFlow's design. Moreover, we believe SDNs can benefit significantly from VeriFlow's data plane verification layer: the network operator can verify that the network's forwarding behavior is correct, without needing to inspect (or trust) relatively complex controller code, which may be developed by parties outside the network operator's control.

3 Design of VeriFlow

Checking network-wide invariants in the presence of complex forwarding elements can be a hard problem. For example, packet filters alone make reachability checks NP-Complete [19]. Aiming to perform these checks in real-time is therefore challenging. Our design tackles this problem as follows. First, we monitor all the network update events in a live network as they are generated by network control applications, the devices, or the network operator. Second, we confine our verification activities to only those parts of the network whose actions may be influenced by a new update. Third, rather than checking invariants with a general-purpose tool such as a SAT or BDD solver as in [10, 19] (which are generally too slow), we use a custom algorithm. We now discuss each of these design decisions in detail.

VeriFlow's first job is to track every forwarding-state change event. For example, in an SDN such as OpenFlow [21], a centralized controller issues forwarding rules to the network devices to handle flows initiated by users. VeriFlow must intercept all these rules and verify them before they reach the network. To achieve this goal, VeriFlow is implemented as a shim layer between the controller and the network, and monitors all communication in either direction.

For every rule insertion/deletion message, VeriFlow must verify the effect of the rule on the network at very high speed. VeriFlow cannot leverage techniques used by past work [9, 17, 19], because these operate at timescales of seconds to hours. Unlike previous solutions, we do not want to check the entire network on each change. We solve this problem in three steps. First, using the new rule and any overlapping existing rules, we slice the network into a set of *equivalence classes* (ECs) of packets (§ 3.1). Each EC is a set of packets that experience the same forwarding actions throughout the network. Intuitively, each change to the network will typically only affect a very small number of ECs (see § 5.1). Therefore, we find the set of ECs whose operation could be altered by a rule, and verify network invariants only within those classes. Second, VeriFlow builds individual *forwarding graphs* for every modified EC, representing the network's forwarding behavior (§ 3.2). Third, VeriFlow traverses these graphs (or runs custom user-defined code) to determine the status of one or more invariants (§ 3.3). The following subsections describe these steps in detail. Figure 1 shows the placement and operations of VeriFlow in an SDN.

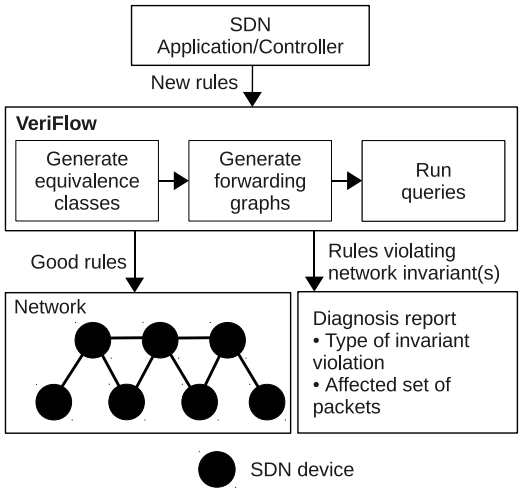

Figure 1: VeriFlow sits between the SDN applications and devices to intercept and check every rule entering the network.

3.1 Slicing the network into equivalence classes

One way to verify network properties is to prepare a model of the entire network using its current data-plane state, and run queries on this model [9, 19]. However, checking the entire network's state every time a new flow rule is inserted is wasteful, and fails to provide real-time response. Instead, we note that most forwarding rule changes affect only a small subset of all possible packets. For example, inserting a longest-prefix-match rule for the destination IP field will only affect forwarding for packets destined to that prefix. In order to confine our verification activities to only the affected set of packets, we slice the network into a set of equivalence classes (ECs) based on the new rule and the existing rules that overlap with the new rule. An equivalence class is defined as follows.

Definition (Equivalence Class): An equivalence class (EC) is a set P of packets such that for any $p_1, p_2 \in P$ and any network device R, the forwarding action is identical for p_1 and p_2 at R.

Separating the entire packet space into individual ECs allows VeriFlow to pinpoint the affected set of packets if a problem is discovered while verifying a newly inserted forwarding rule.

Let us look at an example. Consider an OpenFlow switch with two rules matching packets with destination IP address prefixes 11.1.0.0/16 and 12.1.0.0/16, respectively. If a new rule matching destination IP address prefix 11.0.0.0/8 is added, it may affect packets belonging to the 11.1.0.0/16 range depending on the rules' priority values [6] (the longer prefix may not have higher priority). However, the new rule will not affect packets outside the range 11.0.0.0/8, such as 12.1.0.0/16. Therefore, VeriFlow will only consider the new rule (11.0.0.0/8) and the existing overlapping rule (11.1.0.0/16) while analyzing network properties. These two overlapping rules produce three ECs (represented using the lower and upper bound range values of the destination IP address field): 11.0.0.0 to 11.0.255.255, 11.1.0.0 to 11.1.255.255, and 11.2.255.255 to 11.255.255.255.

VeriFlow needs an efficient data structure to quickly store new network rules, find overlapping rules, and compute the affected ECs. For this we utilize a *multi-dimensional prefix tree (trie)* inspired by traditional packet classification algorithms [26].

A trie is an ordered tree data structure that stores an associative array. In our case, the trie associates the set of packets matched by a forwarding rule with the forwarding rule itself. Each level in the trie corresponds to a specific bit in a forwarding rule (equivalently, a bit in the packet header). Each node in our trie has three branches, corresponding to three possible values that the rule can match: 0, 1, and * (wildcard). The trie can be seen as a composition of several *sub-tries* or dimensions, each corresponding to a packet header field. We maintain a sub-trie in our multi-dimensional trie for each of the mandatory match and packet header fields supported by OpenFlow 1.1.0.[2] (Note that an optimization in our implementation uses a condensed set of fields in the trie;

[2](DL_SRC, DL_DST, NW_SRC, NW_DST, IN_PORT, DL_VLAN, DL_VLAN_PCP, DL_TYPE, NW_TOS, NW_PROTO, TP_SRC, TP_DST, MPLS_LABEL and MPLS_TC).

see § 4.2.) For example, the sub-trie representing the IPv4 destination corresponds to 32 levels in the trie. One of the sub-tries (DL_SRC in our design) appears at the top, the next field's sub-tries are attached to the leaves of the first, and so on (Figure 2). A path from the trie's root to a leaf of one of the bottommost sub-tries thus represents the set of packets that a rule matches. Each leaf stores the rules that match that set of packets, and the devices at which they are located (Figure 2).

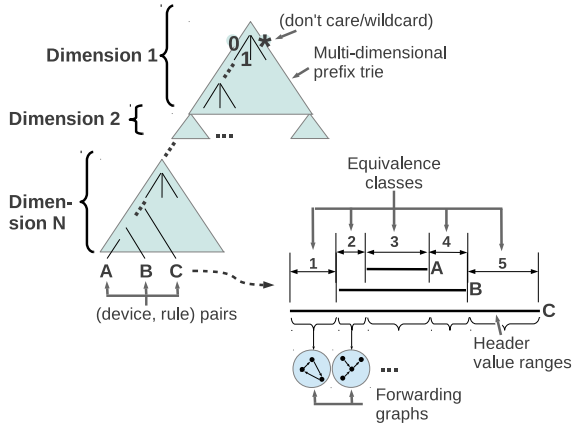

Figure 2: VeriFlow's core algorithmic process.

When a new forwarding rule is generated by the application, we perform a lookup in our trie, by traversing it dimension by dimension to find all the rules that intersect the new rule. At each dimension, we narrow down the search area by only traversing those branches that fall within the range of the new rule using the field value of that particular dimension. The lookup procedure results in the selection of a set of leaves of the bottommost dimension, each with a set of forwarding rules. These rules collectively define a set of packets (in particular, their corresponding forwarding rules) that could be affected by the incoming forwarding rule. This set may span multiple ECs. We next compute the individual ECs as illustrated in Figure 2. For each field, we find a set of disjoint ranges (lower and upper bound) such that no rule splits one of the ranges. An EC is then defined by a particular choice of one of the ranges for each of the fields. This is not necessarily a minimal set of ECs; for example, ECs 2 and 4 in Figure 2 could have been combined into a single EC. However, this method performs well in practice.

3.2 Modeling forwarding state with forwarding graphs

For each EC computed in the previous step, VeriFlow generates a *forwarding graph*. Each such graph is a representation of how packets within an EC will be forwarded through the network. In the graph, a node represents an EC at a particular network device, and a directed edge represents a forwarding decision for a particular (EC, device) pair. Specifically, an edge $X \rightarrow Y$ indicates that according to the forwarding table at node X, packets within this EC are forwarded to Y. To build the graph for each EC, we traverse our trie a second time to find the devices and rules that match packets from that EC. The second traversal is needed to find all those rules that were not necessary to compute the affected ECs in the first traversal, yet can still influence their forwarding behavior. For example, for a new rule with 10.0.0.0/8 specified as the destination prefix, an existing 0.0.0.0/0 rule will not contribute to the generation of the affected ECs, but may influence their forwarding behavior depending on its priority. Given the range values of different fields of an EC, looking up matching rules from the trie structure can be performed very quickly. Here, VeriFlow only has to traverse those branches of the trie having rules that can match packets of that particular EC.

3.3 Running queries

Above, we described how VeriFlow models the behavior of the network using forwarding graphs, building forwarding graphs only for those equivalence classes (ECs) whose behavior may have changed. Next, we answer queries (check invariants) using this model.

VeriFlow maintains a list of invariants to be checked. When ECs have been modified, VeriFlow checks each (invariant, modified EC) pair. An invariant is specified as a verification function that takes as input the forwarding graph for a specific EC, performs arbitrary computation, and can trigger resulting actions. VeriFlow exposes an API (Application Programming Interface), the implementation of which is described in § 4.3, so that new invariants can be written and plugged in.

Up to a certain level of detail, the forwarding graph is an exact representation of the forwarding behavior of the network. Therefore, invariant modules can check a large diversity of conditions concerning network behavior. For example:

- Basic reachability: The verification function traverses the directed edges in the forwarding graph (using depth-first search in our implementation) to determine whether packets will be delivered to the destination address specified in the rule.
- Loop-freeness: The verification function traverses the given EC's forwarding graph to check that it does not contain a loop.
- Consistency: Given two (pre-specified) routers R_1, R_2 that are intended to have identical forwarding operations, the verification function traverses

the forwarding graph starting at R_1 and R_2 to test whether the fate of packets is the same in both cases. (Any difference may indicate a bug.)

Further examples include detecting "black holes" where packets are dropped, ensuring isolation of multiple VLANs, verifying access control policies, checking whether a new rule conflicts with an existing rule, checking whether an EC changes its next hop due to the insertion/deletion of a rule, ensuring that packets always traverse a firewall, and so on.

There are two key limitations on what invariants can be feasibly implemented. First, VeriFlow's forwarding graph construct must include the necessary information. Our current implementation of VeriFlow does not, for example, incorporate information on buffer sizes that would be necessary for certain performance invariants. (There is not, however, any fundamental reason that VeriFlow could not be augmented with such metadata.) Second, the invariant check must be implementable in the *incremental* manner described above where only the modified ECs are considered at each step.

If a verification function finds a violated invariant, it can choose to trigger further actions within VeriFlow. Two obvious actions are dropping the rule that was being inserted into the network, or installing the rule but generating an alarm for the operator. For example, the operator could choose to drop rules that cause a security violation (such as packets leaking onto a protected VLAN), but only generate an alarm for a black hole. Since verification functions are arbitrary code, they may take other actions as well, such as maintaining statistics (e.g., rate of forwarding behavior change) or writing logs.

3.4 Dealing with high verification time

VeriFlow achieves real-time response by confining its verification activities within those parts of the network that are affected when a new forwarding rule is installed. In general, the effectiveness of this approach will be determined by numerous factors, such as the complexity of verification functions, the size of the network, the number of rules in the network, the number of unique ECs covered by a new rule, the number of header fields used to match packets by a new rule, and so on.

However, perhaps the most important factor summarizing verification time is the *number of ECs modified*. As our later experiments will show, VeriFlow's verification time is roughly linear in this number. In other words, VeriFlow has difficulty verifying invariants in real-time when large swaths of the network's forwarding behavior are altered in one operation.

When such disruptive events occur, VeriFlow may need to let new rules be installed in the network without waiting for verification, and run the verification process in parallel. We lose the ability to block problematic rules before they enter the network, but we note several mitigating facts. First, the most prominent example of a disruptive event affecting many ECs is a link failure, in which case VeriFlow anyway cannot block the modification from entering the network. Second, upon (eventually) detecting a problem, VeriFlow can still raise an alarm and remove the problematic rule(s) from the network. Third, the fact that the number of affected ECs is large may itself be worthy of an immediate alarm even before invariants are checked for each EC. Finally, our experiments with realistic forwarding rule update traces (§ 5) show that disruptive events (i.e., events affecting large number of ECs) are rare: in the vast majority of cases (around 99%), the number of affected ECs is small (less than 10).

4 Implementation

We describe three key aspects of our implementation: our shim layer to intercept network events (§ 4.1), an optimization to accelerate verification (§ 4.2), and our API for custom invariants (§ 4.3).

4.1 Making deployment transparent

In order to ease the deployment of VeriFlow in networks with OpenFlow-enabled devices, and to use VeriFlow with unmodified OpenFlow applications, we need a mechanism to make VeriFlow transparent so that these existing OpenFlow entities may remain unaware of the presence of VeriFlow. We built two versions of VeriFlow. One is a *proxy process* [25] that sits between the controller and the network, and is therefore independent of the particular controller. The second version is *integrated* with the NOX OpenFlow controller [14] to improve performance; our performance evaluation is of this version. We expect one could similarly integrate VeriFlow with other controllers, such as Floodlight [2], Beacon [1] and Maestro [11], without significant trouble.

We built our implementation within NOX version 0.9.1 (full beta single-thread version). We integrated VeriFlow within NOX, enabling it to run as a transparent rule verifier sitting between the OpenFlow applications implemented using NOX's API, and the switches and routers in the network. SDN applications running on NOX use the NOX API to manipulate the forwarding state of the network, resulting in OFPT_FLOW_MOD (flow table modification) and other OpenFlow messages generated by NOX. We modified NOX to intercept these messages, and redirect them to our VeriFlow module. This ensures that all messages are intercepted by VeriFlow before they are dispatched to the network. Veri-

Flow then processes and checks the forwarding rules contained in these messages for correctness, and can block problematic flow rules.

To integrate the VeriFlow module, we extend two parts of NOX. First, within the core of NOX, the *send_openflow_command()* interface is responsible for adding (relaying) flow rules from OpenFlow applications to the switches. At the lower layers of NOX, *handle_flow_removed()* handles events that remove rules from switches, due to rule timeouts or commands sent by applications. Our implementation intercepts all messages sent to these two function calls, and redirects them to VeriFlow. To reduce memory usage and improve running time, we pass these messages via shallow copy.

There are five types of flow table modification messages that can be generated by OpenFlow applications: OFPFC_ADD, OFPFC_MODIFY_STRICT, OFPFC_DELETE_STRICT, OFPFC_MODIFY and OFPFC_DELETE. These rules differ in terms of whether they add, modify or delete a rule from the flow table. The strict versions match all the fields bit by bit, whereas the non-strict versions allow wildcards. Our implementation handles all these message types appropriately.

4.2 Optimizing the verification process

We use an optimization technique that exploits the way certain match and packet header fields are handled in the OpenFlow 1.1.0 specification. 10 out of 14 fields in this specification do not support arbitrary wildcards.[3] One can only specify an exact value or the special *ANY* (wildcard) value in these fields. We do not use separate dimensions in our trie to represent these fields, because we do not need to find multiple overlapping ranges for them. Therefore, we only maintain the trie structure for the other four fields (DL_SRC, DL_DST, NW_SRC and NW_DST). Due to this change, we generate the set of affected equivalence classes (ECs) in three steps. First, we use the trie structure to look for network-wide overlapping rules, and find the set of affected packets determined by the four fields that are represented by the trie. Each individual packet set we get from this step is actually a set of ECs that can be distinguished by the other 10 fields. Second, for each of these packet sets, we extract all the rules that can match packets of that particular class from the location/device of the newly inserted rule. We linearly go through all these rules to find non-overlapping range values for the rest of the fields that are not maintained in the trie structure. Thus, each packet set found in the first step breaks into multiple finer packet sets spanning all the 14 mandatory OpenFlow match and packet header fields. Note that in this step we only consider the rules present at the device of the newly inserted rule. Therefore, in the final step, as we traverse the forwarding graphs, we may encounter finer rules at other devices that will generate new packet sets with finer granularity. We handle them by maintaining sets of excluded packets as described in the next paragraph.

Each forwarding graph that we generate using our trie structure represents the forwarding state of a group of packet sets that can be distinguished using the 10 fields that do not support arbitrary wildcards. Therefore, while traversing the forwarding graphs, we only work on those rules that overlap with the newly inserted rule on these 10 fields. As we move from node to node while traversing these graphs, we keep track of the ECs that have been served by finer rules and are no longer present in the primary packet set that was generated in the first place. For example, in a device, a subset of a packet set may be served by a finer rule having higher priority than a coarser rule that serves the rest of that packet set. We handle this by maintaining a set of excluded packets for each forwarding action. Therefore, whenever we reach a node that answers a query (e.g., found a loop or reached a destination), the primary packet set minus the set of excluded packets gives the set of packets that experiences the result of the query.

4.3 API to write general queries

We expose a set of functions that can be used to write general queries in C++. Below is a list of these functions along with the required parameters.

GetAffectedEquivalenceClasses: Given a new rule, this function computes the set of affected ECs, and returns them. It also returns a set of sub-tries from the last dimension of our trie structure. Each sub-trie holds the rules that can match packets belonging to one of the affected ECs. This information can be used to build the forwarding graphs of those ECs. This function takes the following parameters.
- Rule: A newly inserted rule.
- *Returns:* Affected ECs.
- *Returns:* Sub-tries representing the last dimension, and holding rules that can match packets of the affected ECs.

GetForwardingGraph: This function generates and returns the forwarding graph for a particular EC. It takes the following parameters.
- EquivalenceClass: An EC whose forwarding graph will be computed.
- TrieSet: Sub-tries representing the last dimension, and holding rules that match the EC supplied as the first argument.
- *Returns:* Corresponding forwarding graph.

ProcessCurrentHop: This function allows the user to

[3] IN_PORT, DL_VLAN, DL_VLAN_PCP, DL_TYPE, NW_TOS, NW_PROTO, TP_SRC, TP_DST, MPLS_LABEL and MPLS_TC.

traverse a forwarding graph in a custom manner. Given a location and EC, it returns the corresponding next hop. It handles the generation of multiple finer packet sets by computing excluded packet sets that need to be maintained because of our optimization strategy (§ 4.2). Due to this optimization, this function returns a set of (next hop, excluded packet set) tuples — effectively, an annotated directed edge in the forwarding graph. With repeated calls to this function across nodes in the forwarding graphs, custom invariant-checking modules can traverse the forwarding graph and perform arbitrary computation on its structure. This function takes the following parameters.
- ForwardingGraph: The forwarding graph of an EC.
- Location: The current location of the EC.
- *Returns:* (Next hop, excluded packet set) tuples.

Let us look at an example that shows how this API can be used in practice. A network operator may want to ensure that packets belonging to a certain set always pass through a firewall device. This invariant can be violated during addition/deletion of rules, or during link up/down events. To check this invariant, the network operator can extend VeriFlow using the above API to incorporate a custom query algorithm that generates an alarm when the packet set under scrutiny bypasses the firewall device. In fact, the network operator can implement any query that can be answered using the information present in the forwarding graphs.

5 Evaluation

In this section, we present a performance evaluation of our VeriFlow implementation. As VeriFlow intercepts every rule insertion message whenever it is issued by an SDN controller, it is crucial to complete the verification process in real time so that network performance is not affected, and to ensure scalability of the controller. We evaluated the overhead of VeriFlow's operations with the help of two experiments. In the first experiment (§ 5.1), our goal is to microbenchmark different phases of VeriFlow's operations and observe their contribution to the overall running time. The goal of the second experiment (§ 5.2) is to assess the impact of VeriFlow on TCP connection setup latency and throughput as perceived by end users of an SDN.

In all of our experiments, we used our basic reachability algorithms to test for loops and black holes for every flow modification message that was sent to the network. All of our experiments were performed on a Dell Optiplex 9010 machine with an Intel Core i7 3770 CPU with 4 physical cores and 8 threads at 3.4 GHz, and 32 GB of RAM, running 64 bit Ubuntu Linux 11.10.

5.1 Per-update processing time

In this experiment, we simulated a network consisting of 172 routers following a Rocketfuel [7] topology (AS 1755), and replayed BGP (Border Gateway Protocol) RIB (Routing Information Base) and update traces collected from the Route Views Project [8]. We built an OSPF (Open Shortest Path First) simulator to compute the IGP (Interior Gateway Protocol) path cost between every pair of routers in the network. A BGP RIB snapshot consisting of 5 million entries was used to initialize the routers' FIB (Forwarding Information Base) tables. Only the FIBs of the border routers were initialized in this phase. We randomly mapped Route Views peers to border routers in our network, and then replayed RIB and update traces so that they originate according to this mapping. We replayed a BGP update trace containing 90,000 updates to trigger dynamic changes in the network. Upon receiving an update from the neighboring AS, each border router sends the update to all the other routers in the network. Using standard BGP polices, each router updates its RIB using the information present in the update, and updates its FIB based on BGP AS path length and IGP path cost. We fed all the FIB changes into VeriFlow to measure the time VeriFlow takes to complete its individual steps described in § 3. We recorded the run time to process each change individually. Note that in this first set of experiments, only the destination IP address is used to forward packets. Therefore, only this one field contributes to the generation of equivalence classes (ECs). We initialize the other fields with ANY (wildcards).

The results from this experiment are shown in Figure 3(a). VeriFlow is able to verify most of the updates within 1 millisecond (ms), with mean verification time of 0.38ms. Moreover, of this time, the query phase takes only 0.01ms on an average, demonstrating the value of reducing the query problem to a simple graph traversal for each EC. Therefore, VeriFlow would be able to run multiple queries of interest to the network operator (e.g., black hole detection, isolation of multiple VLANs, etc.) within a millisecond time budget.

We found that the number of ECs that are affected by a new rule strongly influences verification time. The scatter plot of Figure 3(b) shows one data point for each observed number of modified ECs (showing the mean verification time across all rules, which modified that number of ECs). The largest number of ECs affected by a single rule was 574; the largest verification latency was 159.2ms due to an update affecting 511 ECs. However, in this experiment, we found that for most updates the number of affected ECs is small. 94.5% of the updates only affected a single EC, and 99.1% affected less than 10 ECs. Therefore, only a small fraction of rules (0.9%) affected large numbers of ECs. This can be observed by

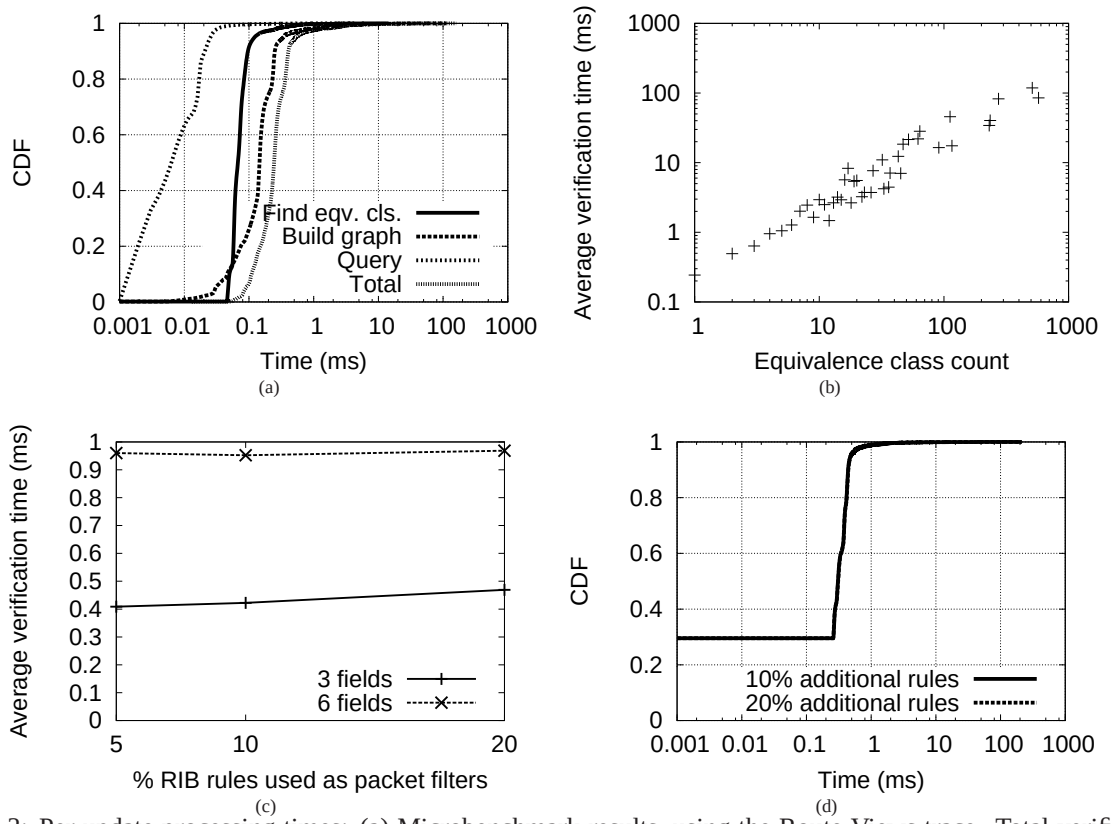

Figure 3: Per-update processing times: (a) Microbenchmark results, using the Route Views trace. Total verification time of VeriFlow remained below 1ms for 97.8% of the updates. (b) Scatter plot showing the influence of number of equivalence classes on verification time. (c) Results from multi-field packet filter experiment using the Route Views trace. As more fields are used in forwarding rules, the running time of VeriFlow increases. The average verification latency is not significantly influenced as we increase the number of filters present in the network. (d) Results from the conflict detection test. VeriFlow is fast enough to compute all the conflicting rules within hundreds of microseconds for 99% of the updates.

looking at the long tail of Figure 3(a).

In the above experiment, we assumed that the network topology remains unchanged, i.e., there are no link or node failures. In case of a link failure or node failure (which can be thought of as failure of multiple links connected to the failed node), the packets that were using that link or node will experience changes in their forwarding behavior. When this happens, VeriFlow's job is to verify the fate of those affected packets. In order to evaluate VeriFlow's performance in this scenario, we used the above topology and traces to run a new experiment. In this experiment, we fed both the BGP RIB trace and update trace to the network. Then we removed each of the packet-carrying links (381 in total) of the network one by one (restoring a removed link before removing the next), and computed the number of affected ECs and the running time of VeriFlow to verify the behavior of those classes. We found that most of the link removals affected a large number of ECs. 254 out of 381 links affected more that 1,000 ECs. The mean verification time to verify a link failure event was 1.15 seconds, with a maximum of 4.05 seconds. We can deal with such cases by processing the forwarding graphs of different ECs in parallel on multi-core processors. This is possible because the forwarding graphs do not depend on each other, or on any shared data structure. However, as link or node failures cannot be avoided once they happen, this may not be a serious issue for network operators.

In order to evaluate VeriFlow's performance in the presence of more fields, we changed the input data set to add packet filters that will selectively drop packets after matching them against multiple fields. We randomly selected a subset of the existing RIB rules currently present in the network, and inserted packet filter rules by specifying values in some of the other fields that were not present in the original trace. We ran this experiment with two sets of fields. In the first set we used TP_SRC and TP_DST in addition to NW_DST (3 fields in total), which was already present in the trace. For each randomly selected RIB rule, we set random values to those

two fields (TP_SRC and TP_DST), and set its priority higher than the original rule. The remaining 11 fields are set to ANY. While replaying the updates, all the 14 fields except NW_DST are set to ANY.

In the second set we used NW_SRC, IN_PORT, DL_VLAN, TP_SRC and TP_DST in addition to NW_DST (6 fields in total). For each randomly selected RIB rule, we set random values to IN_PORT, DL_VLAN, TP_SRC and TP_DST, a random /16 value in NW_SRC, and set the priority higher than the original rule. The remaining 8 fields are set to ANY. While replaying the updates, all the 14 fields except NW_SRC and NW_DST are set to ANY. In the updates, the NW_SRC is set to a random /12 value and the NW_DST is the original value present in the trace. We ran this experiment multiple times varying the percentage of RIB rules that are used to generate random filter rules with higher priority.

Figure 3(c) shows the results of this experiment. Verification time is heavily affected by the number of fields used to classify packets. This happens because as we use more fields to classify packets at finer granularities, more unique ECs are generated, and hence more forwarding graphs need to be verified. We also note from Figure 3(c) that VeriFlow's overall performance is not affected much by the number of filters that we install into the network.

In all our experiments thus far, we kept a fixed order of packet header fields in our trie structure. We started with DL_SRC (DS), followed by DL_DST (DD), NW_SRC (NS) and NW_DST (ND). In order to evaluate the performance of VeriFlow with different field orderings, we re-ran the above packet filter experiment with reordered fields. In all the runs we used random values for the NW_SRC field and used the NW_DST values present in the Route Views traces. All the other fields were set to ANY. We installed random packet filter rules for 10% of the BGP RIB entries. As our dataset only specified values for the NW_SRC and NW_DST fields, there were a total of 12 different orderings of the aforementioned 4 fields. Table 1 shows the results from this experiment.

Table 1: Effect of different field orderings on total running time of VeriFlow.

Order	Time (ms)	Order	Time (ms)
DS-DD-NS-ND	1.001	DS-DD-ND-NS	0.090
DS-NS-DD-ND	1.057	DS-ND-DD-NS	0.096
NS-DS-DD-ND	1.144	ND-DS-DD-NS	0.101
NS-DS-ND-DD	1.213	ND-DS-NS-DD	0.103
NS-ND-DS-DD	1.254	ND-NS-DS-DD	0.15
DS-NS-ND-DD	1.116	DS-ND-NS-DD	0.098

From Table 1, we can see that changing the field order in the trie structure greatly influences the running time of VeriFlow. Putting the NW_DST field ahead of NW_SRC reduced the running time by an order of magnitude (from around 1ms to around 0.1ms). This happens because a particular field order may produce fewer unique ECs compared to other field orderings for the same rule. However, it is difficult to come up with a single field order that works best in all scenarios, because it is highly dependent on the type of rules present in a particular network. Changing the field order in the trie structure dynamically and efficiently as the network state evolves would be an interesting area for future work.

Checking non-reachability invariants: Most of our discussion thus far focused on checking invariants associated with the inter-reachability of network devices. To evaluate the generality of our tool, we implemented two more invariants using our API that were not directly related to reachability: *conflict detection* (whether the newly inserted rule violates isolation of flow tables between network slices, accomplished by checking the output of the EC search phase), and *k-monitoring* (ensuring that all paths in the network traverse one of several deployed monitoring points, done by augmenting the forwarding graph traversal process). We found that the overhead of these checks was minimal. For the conflict detection query, we ran the above filtering experiment using the 6-field set with 10% and 20% newly inserted random rules. However, this time instead of checking the reachability of the affected ECs as each update is replayed, we only computed the set of rules that overlap/conflict with the newly inserted rule. The results from this experiment are shown in Figure 3(d).

From this figure, we can see that conflicting rule checking can be done quickly, taking only 0.305ms on average. (The step in the CDF is due to the fact that some withdrawal rules did not overlap with any existing rule.)

For the k-monitoring query experiment, we used a snapshot of the Stanford backbone network data-plane state that was used in [17]. This network consists of 16 routers, where 14 of these are internal routers and the other 2 are gateway routers used to access the outside network. The snapshot contains 7,213 FIB table entries in total. In this experiment, we used VeriFlow to test whether *all* the ECs currently present in the network pass through one of the two gateway routers of the network. We observed that at each location the average latency to perform this check for all the ECs is around 68.06ms with a maximum of 75.39ms.

5.2 Effect on network performance

In order to evaluate the effect of VeriFlow's operations on user-perceived TCP connection setup latency and the network throughput, we emulated an OpenFlow network consisting of 172 switches following the aforementioned Rocketfuel topology using Mininet [3]. Mininet creates

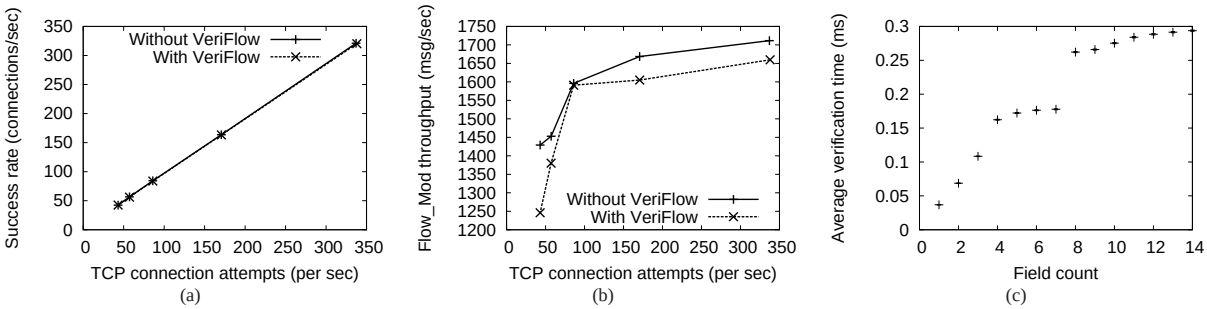

Figure 4: Effect on network performance: (a) TCP connection setup throughput, and (b) Throughput of flow modification (Flow_Mod) messages, with and without VeriFlow. For different loads, VeriFlow imposes minimal overhead. (c) Effect of the number of packet header fields on VeriFlow's verification speed. As we increase the number of fields, overhead of VeriFlow increases gradually.

a software-defined network (SDN) with multiple nodes on a single machine. We connected one host to every switch in this emulated network. We ran the NOX OpenFlow controller along with an application that provides the functionality of a learning switch. It allows a host to reach any other host in the network by installing flow rules in the switches using flow modification (Flow_Mod) messages. We implemented a simple TCP server program and a simple TCP client program to drive the experiment. The server program accepts TCP connections from clients and closes the connection immediately. The client program consists of two threads. The primary thread continuously sends connect requests to a random server using a non-blocking socket. To vary the intensity of the workload, our TCP client program generates connections periodically with a parameterized sleep interval (*S*). The primary thread at each client sleeps for a random interval between 0 to *S* seconds (at microsecond granularity) before initiating the connection request, and iterating. The secondary thread at each client uses the *select* function to look for connections that are ready for transmission or experienced an error. A user supplied polling interval (*P*) is used to control the rate at which the select call will return. We set *P* inversely proportional to the *S* value to avoid busy waiting and to allow the other processes (e.g., Open vSwitch [5]) to get a good share of the CPU. We ran the server program at each of the 172 hosts, and configured the client programs at all the hosts to continually connect to the server of random hosts (excluding itself) over a particular duration (at least 10 minutes). In the switch application, we set the rule eviction idle timeout to 1 second and hard timeout to 5 seconds.

We ran this experiment first with NOX alone, and then with NOX and VeriFlow. We used the same seed in all the random number generators to ensure similar loads in both the runs. We also varied the *S* value to monitor the performance of VeriFlow under a range of network loads.

Figure 4(a) shows the number of TCP connections that were successfully completed per second for different workloads both with and without VeriFlow. From this figure, we can see that in all the cases VeriFlow imposes negligible overhead on the TCP connection setup throughput in our emulated OpenFlow network. The largest reduction in throughput that we observed in our experiments was only 0.74%.

Figure 4(b) shows the number of flow modification (Flow_Mod) messages that were processed and sent to the network per second for different workloads both with and without VeriFlow. From this figure, again we can see that in all the cases VeriFlow imposes minimal overhead on the flow modification message throughput. The largest reduction in throughput that we observed in our experiments was only 12.8%. This reduction in throughput is caused by the additional processing time required to verify the flow modification messages before they are sent to the network.

In order to assess the impact of VeriFlow on end-to-end TCP connection setup latency, we ran this experiment with *S* set to 30 seconds. We found that in the presence of VeriFlow, the average TCP connection setup latency increases by 15.5% (45.58ms without VeriFlow versus 52.63ms with VeriFlow). As setting up a TCP connection between two hosts in our emulated 172 host OpenFlow network requires installing flow rules into more than one switch, the verification performed by VeriFlow after receiving each flow rule from the controller inflates the end-to-end connection setup latency to some extent.

Lastly, we ran this experiment after modifying VeriFlow to work with different numbers of OpenFlow packet header fields. Clearly, if we restrict the number of fields during the verification process, there will be less work for VeriFlow, resulting in faster verification time. In this experiment, we gradually increased the number of OpenFlow packet header fields that were used during the verification process (from 1 to 14). VeriFlow simply ignored the excluded fields, and it reduced the number of dimensions in our trie structure. We set *S* to 10

seconds and ran each run for 10 minutes. During the runs, we measured the verification latency experienced by each flow modification message generated by NOX, and computed their average at each run.

Figure 4(c) shows the results from this experiment. Here, we see that with the increase in the number of packet header fields, the verification overhead of VeriFlow increases gradually but always remains low enough to ensure real-time response. The 5 fields that contributed most in the verification overhead are DL_SRC, DL_DST, NW_SRC, NW_DST and DL_TYPE. This happened because these 5 fields had different values at different flow rules, and contributed most in the generation of multiple ECs. The other fields were mostly wildcards, and did not generate additional ECs.

Comparison with related work: Finally, we compared performance of our technique with two pieces of related work: the Hassel tool presented in [17] (provided to us by the authors), and a BDD-based analysis tool that we implemented from scratch following the strategy presented in [10] (the original code was not available to us). The authors of [17] provided two copies of their tool, one in Python and one in C, and we evaluated using the better-performing C version. While we note these works solve different problems from our work (e.g., HSA performs static verification, and does it between port pairs), we present these results to put VeriFlow's performance in context. First, we ran Hassel over the snapshot of the Stanford backbone network dataplane state that was used in [17]. We found that Hassel's average time to check reachability between a pair of ports (effectively exploring all ECs for that source-destination pair) was 578.62ms, with a maximum of 6.24 seconds. In comparison, VeriFlow took only 68.06ms on average (with a maximum of 75.39ms) to test the reachability of all the ECs currently present at a single node in the network. Next, in the BDD-based approach, we used the NuSMV [4] model checker to build a BDD using a new rule and the overlapping existing rules, and used CTL (Computation Tree Logic) to run reachability queries [10]. Here, we used the Rocketfuel topology and Route Views traces that we used in our earlier experiments. We found that this approach is quite slow and does not provide real-time response while inserting and checking new forwarding rules. Checking an update took 335.71ms on an average with a maximum of 67.16 seconds.

6 Discussion and Future Work

Deciding when to check: VeriFlow may not know when an invariant violation is a true problem rather than an intermediate state during which the violation is considered acceptable by the operator. For example, in an SDN, applications can install rules into a set of switches to build an end-to-end path from a source host to a destination host. However, as VeriFlow is unaware of application semantics, it may not be able to determine these rule set boundaries. This may cause VeriFlow to report the presence of temporary black holes while processing a set of rules one by one. One possible solution is for the SDN application to tell VeriFlow when to check. Moreover, VeriFlow may be used with consistent update mechanisms [20, 24], where there are well-defined stages during which the network state is consistent and can be checked.

Handling packet transformations: We can extend our design to handle rules that perform packet transformation such as Network Address Translation. A transformation rule has two parts – the match part determines the set of packets that will undergo the transformation, and the transformation part represents the set of packets into which the matched packets will get transformed. We can handle this case by generating additional equivalence classes and their corresponding forwarding graphs, to address the changes in packet header due to the transformations. In the worst case, if we have transformations at every hop (e.g., in an MPLS network), then we may need to traverse our trie structure multiple times to build an end-to-end path of a particular packet set. We leave a full design and implementation to future work.

Multiple controllers: VeriFlow assumes it has a complete view of the network to be checked. In a multi-controller scenario, obtaining this view in real time would be difficult. Checking network-wide invariants in real time with multiple controllers is a challenging problem for the future.

7 Related Work

Recent work on debugging general networks and SDNs focuses on detecting network anomalies [10, 19], checking OpenFlow applications [12], ensuring data-plane consistency [20, 24], and allowing multiple applications to run side-by-side in a non-conflicting manner [22, 23, 25]. However, unlike VeriFlow, none of the existing solutions provides real-time verification of network-wide invariants as the network experiences dynamic changes.

Checking OpenFlow applications: Several tools have been proposed to find bugs in OpenFlow applications and to allow multiple applications run on the same physical network in a non-conflicting manner. NICE [12] performs symbolic execution of OpenFlow applications and applies model checking to explore the state space of an entire OpenFlow network. Unlike VeriFlow, NICE is a proactive approach that tries to figure out invalid system

states by using a simplified OpenFlow switch model. It is not designed to check network properties in real time. FlowVisor [25] allows multiple OpenFlow applications to run side-by-side on the same physical infrastructure without affecting each others' actions or performance. Unlike VeriFlow, FlowVisor does not verify the rules that applications send to the switches, and does not look for violations of key network invariants.

In [22], the authors presented two algorithms to detect conflicting rules in a virtualized OpenFlow network. In another work [23], Porras et al. extended the NOX OpenFlow controller with a live rule conflict detection engine called FortNOX. Unlike VeriFlow, both of these works only detect conflicting rules, and do not verify the forwarding behavior of the affected packets. Therefore, VeriFlow is capable of providing more useful information compared to these previous works.

Ensuring data-plane consistency: Static analysis techniques using data-plane information suffer from the challenge of working on a consistent view of the network's forwarding state. Although this issue is less severe in SDNs due to their centralized controlling mechanism, inconsistencies in data-plane information may cause transient faults in the network that go undetected during the analysis phase. Reitblatt et al. [24] proposed a technique that uses an idea similar to the one proposed in [15]. By tagging each rule by a version number, this technique ensures that switches forward packets using a consistent view of the network. This same problem has been addressed in [20] using a different approach. While these works aim to tackle transient inconsistencies in an SDN, VeriFlow tries to detect both transient and long-term anomalies as the network state evolves. Therefore, using these above mechanisms along with VeriFlow will ensure that whenever VeriFlow allows a set of rules to reach the switches, they will forward packets without any transient and long-term anomalies.

Checking network invariants: The router configuration checker (rcc) [13] checks configuration files to detect faults that may cause undesired behavior in the network. However, rcc cannot detect faults that only manifest themselves in the data plane (e.g., bugs in router software and inconsistencies between the control plane and the data plane; see [19] for examples).

Anteater [19] uses data-plane information of a network, and checks for violations of key network invariants (absence of routing loops and black holes). Anteater converts the data-plane information into boolean expressions, translates network invariants into instances of boolean satisfiability (SAT) problems, and checks the resultant SAT formulas using a SAT solver. Although Anteater can detect violations of network invariants, it is static in nature, and does not scale well to dynamic changes in the network (taking up to hundreds of seconds to check a single invariant). Header Space Analysis [17] is a system with goals similar to Anteater, and is also not real time.

Concurrent with our work, NetPlumber [16] is a tool based on Header Space Analysis (HSA) that is capable of checking network policies in real time. NetPlumber uses HSA in an incremental manner to ensure real-time response. Unlike VeriFlow, which allows users to write their own custom query procedures, NetPlumber provides a policy language for network operators to specify network policies that need to be checked.

ConfigChecker [10] and FlowChecker [9] convert network rules (configuration and forwarding rules respectively) into boolean expressions in order to check network invariants. They use Binary Decision Diagram (BDD) to model the network state, and run queries using Computation Tree Logic (CTL). VeriFlow uses graph search techniques to verify network-wide invariants, and handles dynamic changes in real time. Moreover, unlike previous solutions, VeriFlow can *prevent* problems from hitting the forwarding plane, whereas FlowChecker find problems after they occur and (potentially) cause damage. ConfigChecker, like rcc, cannot detect problems that only affect the data plane.

An early version of VeriFlow was presented in [18] but only supported checking for a single header field with a basic reachability invariant, had comparatively high overhead, and had a limited evaluation.

8 Conclusion

In this paper, we presented VeriFlow, a network debugging tool to find faulty rules issued by SDN applications, and optionally prevent them from reaching the network and causing anomalous network behavior. VeriFlow leverages a set of efficient algorithms to check rule modification events in real time before they are sent to the live network. To the best of our knowledge, VeriFlow is the first tool that can verify network-wide invariants in a live network in real time. With the help of experiments using a real world network topology, real world traces, and an emulated OpenFlow network, we found that VeriFlow is capable of processing forwarding table updates in real time.

Acknowledgments

We thank our shepherd, Richard Mortier, and the anonymous reviewers for their valuable comments. We gratefully acknowledge the support of the NSA Illinois Science of Security Lablet, and National Science Foundation grants CNS 1040396 and CNS 1053781.

References

[1] Beacon OpenFlow Controller. https://openflow.stanford.edu/display/Beacon/Home.

[2] Floodlight Open SDN Controller. http://floodlight.openflowhub.org.

[3] Mininet: Rapid prototyping for software defined networks. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.

[4] NuSMV: A new symbolic model checker. http://nusmv.fbk.eu.

[5] Open vSwitch. http://openvswitch.org.

[6] OpenFlow switch specification. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[7] Rocketfuel: An ISP topology mapping engine. http://www.cs.washington.edu/research/networking/rocketfuel.

[8] University of Oregon Route Views Project. http://www.routeviews.org.

[9] AL-SHAER, E., AND AL-HAJ, S. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig* (2010).

[10] AL-SHAER, E., MARRERO, W., EL-ATAWY, A., AND ELBADAWI, K. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *ICNP* (2009).

[11] CAI, Z., COX, A. L., AND NG, T. S. E. Maestro: A system for scalable openflow control. http://www.cs.rice.edu/~eugeneng/papers/TR10-11.pdf.

[12] CANINI, M., VENZANO, D., PERESINI, P., KOSTIC, D., AND REXFORD, J. A NICE way to test OpenFlow applications. In *NSDI* (2012).

[13] FEAMSTER, N., AND BALAKRISHNAN, H. Detecting BGP configuration faults with static analysis. In *NSDI* (2005).

[14] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an operating system for networks. In *SIGCOMM CCR* (2008).

[15] JOHN, J. P., KATZ-BASSETT, E., KRISHNAMURTHY, A., ANDERSON, T., AND VENKATARAMANI, A. Consensus routing: The Internet as a distributed system. In *NSDI* (2008).

[16] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013).

[17] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *NSDI* (2012).

[18] KHURSHID, A., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying network-wide invariants in real time. In *HotSDN* (2012).

[19] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the data plane with Anteater. In *SIGCOMM* (2011).

[20] MCGEER, R. A safe, efficient update protocol for OpenFlow networks. In *HotSDN* (2012).

[21] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., AND SHENKER, S. OpenFlow: Enabling innovation in campus networks. In *SIGCOMM CCR* (2008).

[22] NATARAJAN, S., HUANG, X., AND WOLF, T. Efficient conflict detection in flow-based virtualized networks. In *ICNC* (2012).

[23] PORRAS, P., SHIN, S., YEGNESWARAN, V., FONG, M., TYSON, M., AND GU, G. A security enforcement kernel for OpenFlow networks. In *HotSDN* (2012).

[24] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for network update. In *SIGCOMM* (2012).

[25] SHERWOOD, R., GIBB, G., YAP, K.-K., APPENZELLER, G., CASADO, M., MCKEOWN, N., AND PARULKAR, G. Can the production network be the testbed? In *OSDI* (2010).

[26] VARGHESE, G. Network Algorithmics: An interdisciplinary approach to designing fast networked devices, 2004.

[27] XIE, G., ZHAN, J., MALTZ, D., ZHANG, H., GREENBERG, A., HJALMTYSSON, G., AND REXFORD, J. On static reachability analysis of IP networks. In *INFOCOM* (2005).

[28] YUAN, L., MAI, J., SU, Z., CHEN, H., CHUAH, C.-N., AND MOHAPATRA, P. FIREMAN: A toolkit for firewall modeling and analysis. In *SnP* (2006).