VeriFlow: Verifying Network-Wide Invariants in Real Time

Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, P. Brighten Godfrey
Department of Computer Science
University of Illinois at Urbana-Champaign
201 North Goodwin Avenue
Urbana, Illinois 61801-2302, USA
{khurshi1, wzhou10, caesar, pbg}@illinois.edu

ABSTRACT

Networks are complex and prone to bugs. Existing tools that check configuration files and 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 preliminary design, VeriFlow, which suggests that this goal is achievable. 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. Based on an implementation using a Mininet OpenFlow network and Route Views trace data, we find that VeriFlow can perform rigorous checking within hundreds of microseconds per rule insertion.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management, Network monitoring*

General Terms

Algorithms, Design, Experimentation, Management, Performance, Security, Verification

Keywords

Software-defined networking, OpenFlow, forwarding, debugging, real time

1. INTRODUCTION

Network forwarding behaviors are complex, including code-dependent 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 and security. 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) will ease the development of network applications, 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 [15].

One solution is to rigorously check network software or configuration for bugs prior to deployment. Symbolic execution [7] can catch bugs through exploration of all possible code paths, but is usually not tractable for large software. Analysis of configuration files [8,18] 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 not be true in an SDN network where controllers can be operated by other parties [15]. Another approach is to statically analyze snapshots of the network-wide data-plane state [5, 6, 11, 12]. These tools operate offline, and thus only find bugs after they happen.

This paper studies the following question: *Is it possible to check network-wide invariants, such as absence of routing loops, in real time as the network evolves?* This would enable us to check updates before they hit the network, allowing us to raise alarms, or even prevent bugs as they occur by blocking problematic changes. However, existing techniques for checking networks are not adequate for this purpose as they operate on timescales of seconds to hours [6,11,12] [1]. As current SDN controllers are capable of handling around 30K new flow installs per second while maintaining a sub-10ms flow install time [16], rule verification latency in the order of seconds is not enough to ensure real-time response, and will affect controller throughput immensely. Delaying updates for processing can harm consistency of network state, and reduce reaction time of protocols with real-time requirements such as routing and fast failover. Moreover, checking network-wide properties seems to require network-wide state, and processing churn of a large network could introduce scaling challenges. Hence, we need some way to perform this checking at very high speeds.

We present a preliminary design, VeriFlow, which demonstrates that the goal of real-time verification is achievable.

[1]The average run time of reachability tests in [11] is 13 seconds.

HotSDN'12, August 13, 2012, Helsinki, Finland.

VeriFlow leverages SDN to obtain a picture of the network as it evolves by sitting as a layer between the controller and the devices, and checks validity of invariants as each rule is inserted. However, SDN in isolation 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 routing loops, access control policies, or isolation between virtual networks.

Our prototype implementation of VeriFlow checks OpenFlow [13] and IP forwarding rules. We microbenchmarked VeriFlow by simulating a real IP network using real BGP traces collected from Route Views [4]. We also evaluated its overhead relative to NOX [10] in an emulated OpenFlow network using Mininet [1]. We find that VeriFlow is able to verify network 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 7% on average. In summary, our key contribution is to present the first tool that can check network-wide invariants in real time.

2. DESIGN OF VERIFLOW

Checking network-wide invariants in the presence of complex forwarding elements (such as routers, firewalls, packet transformers) can be a hard problem to solve. For example, in [12], it was shown that packet filters make reachability checks NP-Complete, and if arbitrary programs are allowed in the data plane, then reachability becomes undecidable. Aiming to perform these checks in real-time makes the problem even harder. Our design tackles this problem as follows. First, we monitor all the network update events in a live network. 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 [6, 12] (which are generally too slow), we use a custom algorithm that is sufficient to verify many kinds of invariants. 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 [13], a centralized controller issues forwarding rules to the network devices to handle flows initiated by users. VeriFlow has to intercept all these rules and verify them before they reach the network. To achieve this goal, VeriFlow sits as a shim layer between the controller and the network (similar to FlowVisor [15]), and monitors all communication in either direction.

For every rule insertion/deletion message, VeriFlow must verify the effect of the rule on the entire network at very high speeds. We solve this problem in three steps. First, we slice the network into a set of *equivalence classes* of packets (Section 2.1). Packets belonging to an equivalence class experience the same forwarding actions throughout the network. Intuitively, each change to the network will typically only affect a very small number of equivalence classes. Therefore, we find the set of equivalence classes whose operation could be altered by a rule, and verify network invariants only within those classes. Second, VeriFlow builds individual *forwarding graphs* for every equivalence class using the current network state (Section 2.2). Third, VeriFlow traverses these graphs to determine the status of one or more invariants (Section 2.3). The following subsections describe these steps in detail. Figure 1 shows the placement and operations of VeriFlow in an SDN.

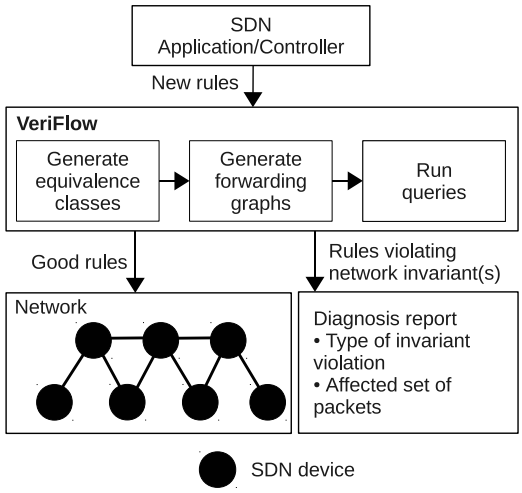

Figure 1: VeriFlow sits between the SDN applications and devices to intercept and check every rule that the network experiences.

2.1 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 [5, 12]. However, checking the entire network's state every time a new flow rule is inserted is wasteful and fails to provide real-time response [12]. Instead, we note that most forwarding rule changes affect only a small subset of all possible packets. For example, inserting a IP longest-prefix-match rule will only affect forwarding for packets destined for that prefix. However, there can be rules in a network device that overlap with a newly inserted rule and match the same set of packets. If this happens then it is possible that the new rule may (inadvertently) alter the path of a set of packets. In order to confine our verification activities only on the affected set of packets, we slice the network into a set of equivalence classes based on the new rule and the existing rules that overlap with the new rule. An equivalence class 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 equivalence classes allows VeriFlow to pinpoint the affected set of packets if a problem is discovered.

Let us look at an example. Assume that a switch in an OpenFlow network has two flow rules matching two disjoint sets of packets. The first rule matches all packets whose destination IP address falls within the range 11.1.0.0/16, and the second rule matches all packets whose destination IP address falls within the range 12.1.0.0/16. Now, if a new rule with destination IP address prefix 11.0.0.0/8 is added into the switch, it may affect packets belonging to the 11.1.0.0/16 range depending on the priority values of these two rules [2]. However, the new rule will never affect packets that belong to the 12.1.0.0/16 range. 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 will result in the following three equivalence classes: (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 has to utilize an efficient data structure to quickly store new network rules and find overlapping rules. We achieve this goal with the help of a *prefix tree* or *trie* data structure inspired by the packet classification algorithms presented in [17]. A trie is an ordered tree data structure that is used to store an associative array. We use the packet header fields as keys in our trie. These include the MAC and IP addresses (both source and destination) and transport protocol ports. We use a single trie to store all the rules present in the network. In the current version of VeriFlow, we only use the destination IP address to build the forwarding state of the entire network. More details on our trie implementation are presented in Section 3.2.

2.2 Modeling forwarding state with forwarding graphs

In order to determine the forwarding behavior of each equivalence class, VeriFlow generates individual forwarding graphs for all the equivalence classes computed in the previous step. This graph is a representation of how packets within an equivalence class will be forwarded through the network. It consists of nodes representing network devices and directed edges representing forwarding decisions for that equivalence class at each node. Hence, a node represents an equivalence class at a particular device. We put a directed edge from node A to node B if according to the forwarding table at node A, the next hop for the equivalence class is node B. For each equivalence class, we traverse the trie structure to find the devices and rules that match packets from that equivalence class, and build the graph using this information. A forwarding graph contains all the information needed to answer queries posted by network operators.

2.3 Running queries

Above, we described how we model the network's behavior. Next, we need some way to answer queries (check invariants), using this model. To do this, we provide an algorithm, which takes as input an invariant to be checked, traverses the forwarding graphs of the affected equivalence classes, and outputs information about whether the invariant holds.

There exists a large diversity of queries that can be expressed as network reachability [6] (e.g., detecting black holes and routing loops, ensuring isolation of multiple VLANs, verifying access control policies). Hence, we focus on supporting reachability queries. However, to support other types of queries, it would be straightforward to extend VeriFlow to accept user-provided modules that compute arbitrary properties, given access to the equivalence classes and their forwarding graphs.

Basic reachability algorithm: Given a snapshot of the network data-plane state and a set of new rules, network-wide invariants can be verified by tracing the traversal paths of all the affected equivalence classes. VeriFlow performs this step by traversing every forwarding graph (computed in the previous step) using depth-first search. During this traversal, VeriFlow tries to detect violations of network-wide invariants. The outcome of this traversal can be a set of possible destinations, including none (black hole), or may result in a routing loop. In particular, while traversing a forwarding graph, if VeriFlow encounters a node twice, then it concludes that insertion of the new rule will result in a routing loop. On the other hand, if VeriFlow encounters a node that does not have any outgoing edge and hence does not lead to the intended destination, then it concludes that there is a black hole in the network.

Verification actions: If VeriFlow determines that an invariant is violated, it executes an associated action that is pre-configured for each invariant by the network operator. Two obvious actions the operator could choose are dropping the rule, 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.

3. IMPLEMENTATION

In this section, we describe three key implementation challenges of our design. We start with a description of our interfacing module that helps VeriFlow to intercept all network events in an OpenFlow network in a transparent manner (Section 3.1). Section 3.2 provides some details on the use of our trie structure. Finally, in Section 3.3, we discuss a graph-cache based strategy that we use to speed up the verification process.

3.1 Interfacing with OpenFlow entities

In order to ease the deployment of VeriFlow in any OpenFlow network and use VeriFlow with unmodified OpenFlow applications, we need a mechanism to make VeriFlow transparent so that OpenFlow entities remain completely unaware of the presence of VeriFlow. We do this by implementing VeriFlow as a proxy application that sits between OpenFlow switches and the controller. OpenFlow switches need to be configured to connect to VeriFlow instead of the OpenFlow controller. The switches consider VeriFlow as their controller. For every connection VeriFlow receives from the OpenFlow switches, it initiates a new connection towards the actual controller and simply copies all the bytes sent from the switches to the controller and vice versa. However, simply copying all the bytes from one end to another will not serve our main purpose, i.e., verification of newly inserted rules. VeriFlow has to determine message boundaries within this stream of bytes and filter out rule insertion/deletion messages. To achieve this, VeriFlow buffers the bytes it receives from either end and checks whether it received a complete OpenFlow message or not. Whenever VeriFlow detects a *Flow Modification* message, it invokes its rule verification module.

3.2 Performing rule verification

As we mentioned in Section 2, we maintain a trie data structure to store all the forwarding rules present in all the devices in the network. This allows us to quickly look-up the existing rules that overlap with a newly inserted rule. We consider each rule as a binary string and use individual bits to prepare the trie. Each level in our trie represents a single bit in a particular rule. For example, for traditional destination prefix-based routing, there are 32 levels in the trie. Each node in our trie has three branches – the first branch is taken if the corresponding rule bit is 0, the second is taken if the bit is 1, and the third is taken if the bit is

don't care (i.e., a wildcard). The leaves in the trie store the actual rules that are represented by the path that leads to a particular leaf starting from the root of the trie. Once we build the trie, searching for overlapping rules becomes pretty simple and extremely fast. Given a new rule, we start with the first bit of the rule and traverse the trie starting from its root. We examine each bit and take the branch that the bit value points to. For don't care bits, we explore all the branches of the current node, as a don't care bit can take any value. For the 0 (or, 1) bit, we explore both the 0 (or, 1) branch and the don't care branch. Once we reach the leaves of all the paths that we explore, we get a list of rules that overlap with the new rule. We use these rules to construct the equivalence classes and forwarding graphs to be used for verifying network properties.

3.3 Speeding up VeriFlow with a graph-cache

As the network experiences new updates, new equivalent classes of packets will be produced due to the interactions between existing and new rules. However, some equivalent classes will remain common across multiple rule insertions and their forwarding graphs can be reused by updating the graphs with the new rules. Therefore, we maintain a cache of forwarding graphs indexed by their equivalence classes. This saves time and allows VeriFlow to perform rule verification very quickly resulting in real-time response.

4. EVALUATION

In this section, we present performance results 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 (Section 4.1), our goal is to microbenchmark different phases of VeriFlow's operations and observe their contribution to the overall run time. This allows us to focus on those parts of VeriFlow that can be further optimized to reduce the verification latency.

The goal of the second experiment (Section 4.2) is to assess the impact of VeriFlow on TCP connection setup latency as perceived by end users of an SDN. This is important because setting up an end-to-end TCP connection requires multiple flow rules to be installed in a multi-hop network. As VeriFlow will be intercepting each of these rules and verify their effects one by one, it is important to keep the overhead as low as possible so that end users do not experience significant delay while setting up a new TCP flow. Both of our experiments were performed on a Dell Optiplex 990 machine with an Intel Core i7 2600 CPU with 4 cores at 3.4 GHz and 16 GB of RAM running 64 bit Ubuntu Linux 10.10.

4.1 Microbenchmarking VeriFlow run time

In this experiment, we simulated a network consisting of 172 routers following a Rocketfuel [3] topology (AS 1755), and replayed BGP (Border Gateway Protocol) RIB (Routing Information Base) and update traces collected from the Route Views Project [4]. We used 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 network FIB tables. Then we replayed a BGP update trace containing 90,000 updates to trigger dynamic changes in the network. We randomly mapped Route Views peers to border routers in our network, and then replayed updates so that they originate according to this mapping. 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 (Forwarding Information Base). We fed all the FIB changes into VeriFlow to measure the time VeriFlow takes to complete its individual steps. The results from this experiment are shown in Figure 2.

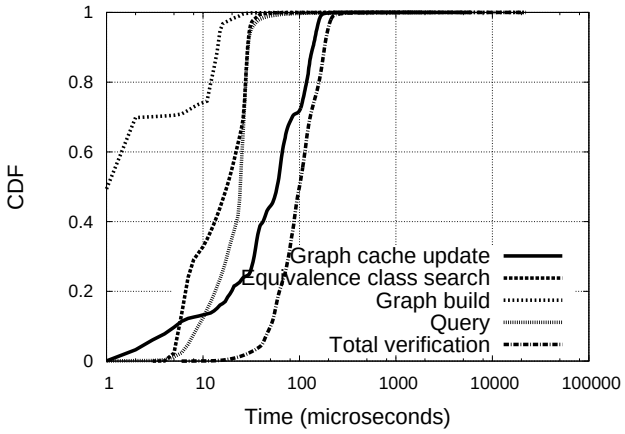

Figure 2: Results from simulation using the Route Views trace. Total verification time of VeriFlow remained well below 1 millisecond for most of the updates.

From Figure 2, we see that VeriFlow is able to verify most of the updates within 1 millisecond. Please note that the X-axis in Figure 2 is plotted in a log scale. The mean verification time for the updates is only 108 microseconds and each query takes only 21 microseconds on an average. By limiting the verification latency within hundreds of microseconds, VeriFlow ensures real-time response while processing rule insertion messages. Moreover, this allows network operators to run multiple queries (e.g., black hole detection, isolation of multiple VLANs, etc.) within a millisecond time budget.

4.2 Effect on TCP connection setup latency

In order to evaluate the effect of VeriFlow's operations on user-perceived TCP connection setup latency, we emulated a 20 node OpenFlow network using Mininet. Mininet creates a software-defined network (SDN) with multiple nodes on a single machine. Our network consists of 10 OpenFlow switches arranged in a chain-like topology with a host connected to every switch. We ran a NOX 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. 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 just connects to a given server and closes the connection immediately. We ran the server program at each of the 10 hosts and configured the client programs at all the hosts to connect to the server

of a random host (excluding itself) as many times as possible over a given duration (10 seconds in our experiment). We set the rule eviction hard timeout to its minimum possible value (1 second) so that VeriFlow experiences the maximum number of new rules being sent by the controller at the arrival of new connection requests. We measured the time it takes to complete each connection.

While using VeriFlow, we encounter two types of overhead. First, there is the overhead imposed by the proxy module, including overhead to buffer bytes, re-assemble complete rule messages, and kernel overheads of context switching and socket interfaces. In order to assess this overhead, we first ran our experiment with VeriFlow placed between the controller and the switches, but disabled the verification module.

The second overhead is the overhead imposed by the verification module itself. To assess this overhead, we ran our experiment a second time with the verification module enabled. This allows us to see the delta increase caused by running our verification algorithms. As a comparison, we ran our experiment a third time without placing VeriFlow between the controller and the switches. In all the runs, we varied the number of hosts and ran each combination 5 times. Figure 3 shows the results from this experiment averaged over 5 runs.

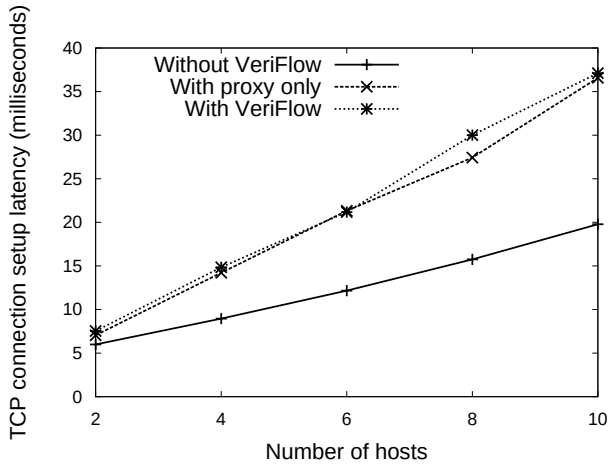

Figure 3: TCP connection setup experiment using Mininet and NOX. The verification phase of VeriFlow incurs minimal overhead compared to the proxy operations.

From Figure 3 we can see that in the presence of VeriFlow the TCP connection setup latency is affected by a significant amount (latency is increased by around 69% on average) compared to the case when VeriFlow is not in action. However, we observe that a major share of this overhead is actually contributed by the proxy module (increases latency by around 62% on average). The overhead imposed by the verification module is rather low and inflates the connection setup latency by only 7% on average. From this observation, we can conclude that implementing VeriFlow as a pluggable module for SDN applications or controller frameworks (such as NOX) will allow VeriFlow to verify network-wide invariants with minimal increase in TCP connection setup latency. Moreover, the overhead caused by proxy operations will be present in any tool that acts as a proxy in an SDN, such as FlowVisor [15].

5. DISCUSSION AND FUTURE WORK

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. We leave a full design and implementation to 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.

Handling queries other than reachability: We can extend our design to answer queries that do not fall into the reachability category. For example, in a data center, the network operator may want to ensure that certain flows do not use the same links, or that the number of flows on a link always remains below a threshold. As mentioned in Section 2.3, plug-in modules could check other properties; performance may, however, depend on the property being checked and its implementation.

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.

6. RELATED WORK

Recent work on debugging general networks and SDNs focuses on detecting network anomalies [6,12], checking OpenFlow applications [7], ensuring data-plane consistency [14], and allowing multiple applications to run side-by-side in a non-conflicting manner [15]. However, unlike VeriFlow, none of the existing solutions provide real-time verification of network-wide invariants as the network experiences dynamic changes.

Programming OpenFlow networks: NOX [10] is a "network operating system" that provides a programming interface to write controller applications for an OpenFlow network. NOX provides an API that is used by the applications to register for OpenFlow events and send OpenFlow commands to the switches. Frenetic [9] is a high-level programming language that can be used to write OpenFlow applications running on top of NOX. Frenetic allows OpenFlow application developers to express packet processing policies at a higher-level manner than the NOX API. However, Fre-

netic and NOX only provide the language and the associated run-time. Unlike VeriFlow, neither NOX nor Frenetic perform correctness checking of updates, limiting their ability to help in detecting bugs in the application code or other issues that may occur while the network is in operation.

Checking OpenFlow applications: NICE [7] 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 [15] allows multiple OpenFlow applications to run side-by-side on the same physical infrastructure without affecting each others' actions or performance. Like VeriFlow, FlowVisor acts as a proxy. Unlike VeriFlow, FlowVisor does not look for violations of key network invariants.

Checking network invariants: The router configuration checker (rcc) [8] 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 [12] for examples).

Anteater [12] uses data-plane information of a network and checks for violations of key network invariants. 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). Concurrent with our work, [11] is a system with goals similar to Anteater, and is also not real time.

ConfigChecker [6] and FlowChecker [5] 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.

7. 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. 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.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their invaluable feedback. We gratefully acknowledge the support of the NSA Illinois Science of Security Lablet, and National Science Foundation grants CNS 1040396 and CNS 1053781.

9. REFERENCES

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

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

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

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

[5] Al-Shaer, E., and Al-Haj, S. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig* (2010).

[6] 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).

[7] Canini, M., Venzano, D., Peresini, P., Kostic, D., and Rexford, J. A NICE way to test OpenFlow applications. In *NSDI* (2012).

[8] Feamster, N., and Balakrishnan, H. Detecting BGP configuration faults with static analysis. In *NSDI* (2005).

[9] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., and Walker, D. Frenetic: A network programming language. In *ICFP* (2011).

[10] 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).

[11] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *NSDI* (2012).

[12] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P. B., and King, S. T. Debugging the data plane with Anteater. In *SIGCOMM* (2011).

[13] 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).

[14] Reitblatt, M., Foster, N., Rexford, J., and Walker, D. Consistent updates for software-defined networks: Change you can believe in! In *HotNets* (2011).

[15] 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).

[16] Tavakoli, A., Casado, M., Koponen, T., and Shenker, S. Applying NOX to the datacenter. In *HotNets* (2009).

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

[18] Yuan, L., Mai, J., Su, Z., Chen, H., Chuah, C.-N., and Mohapatra, P. FIREMAN: A toolkit for firewall modeling and analysis. In *IEEE SnP* (2006).