MONITORING AND VERIFYING NETWORK BEHAVIOR USING DATA-PLANE STATE

BY

AHMED KHURSHID

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

ABSTRACT

Modern computer networks are complex, incorporating hundreds or thousands of network devices from multiple vendors performing diverse functions such as routing, switching, and access control across physical and virtual networks (VPNs and VLANs). As in any complex computer system, these networks are prone to a wide range of errors such as misconfigurations, software bugs, or unexpected interactions across protocols. Previous tools to assist operators in debugging network anomalies primarily focus on analyzing control plane configuration. Configuration analysis is limited in that it cannot find bugs in router software, and is harder to generalize across protocols since it must model complex configuration languages and dynamic protocol behavior.

This thesis studies an alternate approach: diagnosing problems through static analysis of a network's data-plane state. We call it *data-plane verification*. This approach can catch bugs that are invisible at the level of configuration files, and simplifies unified analysis of a network across many protocols and implementations. To prove the applicability and usefulness of data-plane verification, we designed and implemented two tools to rigorously check important network invariants, such as absence of routing loops, routing consistency of replicated devices, and other reachability properties. Our first tool, called *Anteater*, translates a network's data-plane state and invariants into boolean satisfiability problems, and checks them using a SAT solver. Our second tool, called *VeriFlow*, creates a device independent graph model of the network state, and uses standard graph traversal algorithms to detect invariant violations.

We tested our tools with real world network data-plane traces, and with large emulated networks. Both of our tools were able to detect real bugs that went unnoticed to network operators for more than a month. Our tools helped them to narrow down the faulty configurations, and resolve those quickly. Results from emulated larger networks showed that the running time performance of our tools, especially that of VeriFlow, is good enough to detect bugs quickly before they can be exploited by outside attackers. Due to the fast response time of VeriFlow, it can be used in the emerging Software-Defined Networking (SDN) setting as a proactive tool to detect and filter out faulty configurations before they reach network devices.

To my parents, for their love, care, support and belief in me.

ACKNOWLEDGMENTS

A Ph.D. program is a long journey filled with challenges, stress, innovation, occasional failures, accomplishments and of course publications. Mine was no different. All praises go to the Almighty who gave me the opportunity, strength and determination to work through it. In the past 7 years, I got the opportunity to work on real world challenges in computer networking towards the goal of making networks secure and dependable. And at every step of my research and development I was guided with great patience and care by my advisor Professor Matthew Caesar. Without his guidance and continuous support, this thesis would not have been possible. Prof. Caesar always encouraged me to think critically while working in new research topics, and assigned me projects that have immediate real world impact. This helped me immensely to develop my research skills. He also helped me a lot to improve my writing skills by giving detailed constructive comments on my paper drafts. All these supports made my Ph.D. journey smooth and highly productive. It was an honor and privilege to work closely with Prof. Caesar. I want to thank him from the bottom of my heart for all his advises and guidance.

My research received a heavy boost when I got the opportunity to join a project administered by Professor Brighten Godfrey. This project eventually became the core of my Ph.D. thesis. With his extraordinary vision towards next generation network architecture, Prof. Godfrey guided me to work on systems that has the potential to revolutionize the way we manage our networks. I want to thank Prof. Godfrey for his continuous support and guidance that made my thesis stronger.

I want to thank my Ph.D. committee members Professor William Sanders, Professor Nikita Borisov and Professor Jennifer Rexford for their valuable comments. They provided me with useful directions to extend my thesis to make it better in terms of usability and applicability. I highly admire their selfless support and guidance.

Finally, I would like to thank my family, friends and colleagues for their support and encouragement. Especially my friends here at Champaign-Urbana did not let me feel homesick. My dear friends, if I am able to do something useful for the community, please know that you are an integral part of that contribution.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Overview of data-plane verification 2
1.2 Proposed contributions of this thesis 3

CHAPTER 2 STATIC ANALYSIS OF DATA-PLANE STATE 5
2.1 Overview of architecture 6
2.2 Modeling network behavior 7
2.2.1 Running reachability queries 9
2.2.2 Checking forwarding loops, packet loss, and consistency 10
2.2.3 Packet transformation 12
2.3 Complexity 15
2.4 Implementation 17
2.5 Evaluation 19
2.5.1 Bugs found in a deployed network 20
2.5.2 Applicability to router bugs 23
2.5.3 Performance and scalability 24
2.6 Discussion 27
2.7 Related work 30
2.8 Conclusion and future work 31

CHAPTER 3 REAL-TIME VERIFICATION OF NETWORK-WIDE INVARIANTS 33
3.1 Design of VeriFlow 34
3.1.1 Slicing the network into equivalence classes 35
3.1.2 Modeling forwarding state with forwarding graphs 40
3.1.3 Handling packet transformations 40
3.1.4 Running queries 41
3.1.5 Dealing with high verification time 42
3.2 Implementation 43
3.2.1 Making deployment transparent 43
3.2.2 Optimizing the verification process 45
3.2.3 API to write general queries 46

3.3 Evaluation 47
3.3.1 Per-update processing time 48
3.3.2 Effect on network performance 53
3.3.3 Comparison with related work 56
3.3.4 Bugs found in a real network 58
3.4 Discussion 58
3.5 Related work 59
3.6 Current status and future work 62

CHAPTER 4 LIST OF USEFUL INVARIANTS 63
4.1 Basic reachability 63
4.2 Loop-freeness 64
4.3 Black hole detection 64
4.4 Consistency 64
4.5 VLAN isolation 65
4.6 Loose path routing 65
4.7 Strict path routing 65
4.8 Path length constraint 65
4.9 Detect overlapping rules 66
4.10 Change of next hop 66

CHAPTER 5 CONCLUSION AND FUTURE WORK 67

REFERENCES 68

CHAPTER 1

INTRODUCTION

Computer networks, especially the Internet, have become a part and parcel of our day-to-day life. We depend heavily on these networks for our education, communication, business and other activities. From a network user's point of view, a computer network is a communication medium that carries data from the *sender* to the *receiver*. However, from a network operator or service provider's perspective, the same network is a complex ecosystem consisting of multiple parties trying to use the network resources in diverse ways.

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, like any other real world systems, computer networks suffer from errors [29, 31, 32, 34, 58, 71, 78, 81] in the form of hardware failure (such as router or link failures), software failure (such as bugs in router software), and misconfigurations made by network operators. These faults result in routing loops, suboptimal routing, black holes, and access control violations that make services unavailable or prone to attacks (e.g., DDoS attacks). Managing a large network in the midst of these failures, and keeping the network up and running to support the demands of network users is very challenging. This task is almost impossible to perform without the help of automated network monitoring and debugging tools.

However, diagnosing problems in networks remains a black art. Operators often rely on heuristics such as sending probes, reviewing logs, or even observing mailing lists and making phone calls that slow response to failures. To address this, some automated tools for network diagnostics [34, 83] analyze configuration files constructed by operators. While useful, these tools have two limitations stemming from their analysis of high-level configuration files. First, configuration analysis *cannot find bugs in router software*, which interprets and acts on those configuration files. Both commercial and open-source router software regularly exhibit bugs that affect network availability or security [81], and have led to multiple high-profile outages and vulnerabilities [31, 85]. Second, configuration analysis *must model complex configuration languages and dynamic protocol behavior* in order to determine the ultimate effect of a configuration. As a result, these tools generally focus on checking correctness of a single protocol such as BGP [34, 36] or firewalls [17, 83]. Such diagnosis will be unable to reason about interactions that span multiple protocols, and may have difficulty dealing with the diversity in configuration languages from different vendors making up typical networks.

In this thesis, we take a different and complementary approach. Instead of using configuration files to diagnose problems in the control plane, our goal is to *diagnose problems as close as possible to the network's actual behavior* through formal analysis of data-plane state. We call this technique *data-plane verification*. It has several benefits. First, by checking the output produced by network software rather than its inputs, we can catch bugs that are invisible at the level of configuration files. Second, it becomes easier to perform unified analysis of a network across many protocols and implementations, because data-plane analysis avoids modeling dynamic routing protocols, and operates on comparatively simple input formats that are common across many protocols and implementations.

1.1 Overview of data-plane verification

In data-plane verification, the forwarding entries maintained by the routers and switches are used as input. These entries are stored at forwarding tables called FIB (Forwarding Information Base) at the network devices, and can be fetched using CLI (Command Line Interface) commands or through SNMP (Simple Network Management Protocol). In the case of a Software-Defined Network (SDN) such as one running OpenFlow [63], this information can be maintained at the centralized controller, and made readily available to any network debugging tool. These forwarding entries are the final results of all the computations that are performed by the control-plane protocols such as OSPF (Open Shortest Path First), IS-IS (Intermediate System to Intermediate System), BGP (Border Gateway Protocol), etc. This helps us to model the network behavior easily.

Use of data-plane state allows us to accurately model the behavior of different packets that may traverse the network at any given moment. It allows us to compute the set of packets that the network will never carry. We can also monitor the traversal path of different packets to verify whether they can reach their intended destinations or not. Moreover, we can check for violations of access control policies, and look for routing loops and black holes. All these checks are difficult to perform using configuration verification alone.

Data-plane verification is not limited to performing queries and checks that deal with reachability only. By extracting information about link usage, packet counts, and other statistics maintained by the network devices, we can run queries that will be able to provide useful information about the performance of the network. These types of queries will be useful in multi-tenant cloud data centers where multiple users will be sharing a single physical network. These users may have strict performance requirements that the cloud operator needs to support depending on service level agreements (SLAs).

1.2 Proposed contributions of this thesis

In this section, I enumerate a comprehensive list of problems that we solve in this dissertation work. We develop a set of techniques that will help network operators to find problems in their networks easily. Our techniques are based on data-plane verification that allows us to catch errors that may affect packet traversal behavior. Therefore, we are able to catch errors that are visible in the data plane. Moreover, our techniques are useful during network provisioning to assess the effect of potential changes to the network, and for performing "what if" analysis. Below is a list of our proposed contributions.

- **Static analysis of data-plane state:** We propose a SAT-based technique to perform static analysis of network behavior using data-plane state (Chapter 2). We named it *Anteater*. Here, a network's data-plane state and policies or invariants are represented as instances of boolean satisfiability problem (SAT), and a SAT solver is used to perform analysis.

  In this work, I also perform a detailed analysis of a number of known bugs found in an open-source router software. Our focus is to categorize bugs according to their visibility in the data plane, and figure out whether they can be detected by Anteater.

- **Real-time verification of network-wide invariants:** In this work, we study 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. With this goal in mind, we present a design, VeriFlow, which demonstrates that the goal of real-time verification of network-wide invariants is achievable (Chapter 3). 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.

- **A list of invariants to detect network misconfigurations:** I provide a list of invariants that will help network operators to detect and fix common network misconfigurations. Based on our interactions with network operators running enterprise networks I found that these invariants are extremely helpful to ensure a secure and dependable network.

The rest of the thesis is organized as follows. In Chapter 2 I present our first data-plane verification tool called Anteater. I present our second tool called VeriFlow, which is the primary focus of this thesis, in Chapter 3. Chapter 4 presents a list of invariants or policy checks that will benefit network operators to ensure correct behavior of network operations. Finally, I present concluding remarks and direction for future works in Chapter 5.

CHAPTER 2

STATIC ANALYSIS OF DATA-PLANE STATE

In this chapter, I present an overview of our data-plane debugging tool called Anteater [59]. Anteater is a tool for general analysis of the data-plane state of network devices. Anteater collects the network topology and devices' forwarding information bases (FIBs), and represents them as boolean functions. The network operator specifies an invariant to be checked against the network, such as reachability, loop-free forwarding, or consistency of forwarding rules between routers. Anteater combines the invariant and the data-plane state into instances of boolean satisfiability problem (SAT), and uses a SAT solver to perform analysis. If the network state violates an invariant, Anteater provides a specific counterexample — such as a packet header, FIB entries, and path — that triggers the potential bug.

We applied Anteater to a large university campus network, analyzing the FIBs of 178 routers that support over 70,000 end-user machines and servers, with FIB entries inserted by a combination of BGP, OSPF, and static ACLs and routes. Anteater revealed 23 confirmed bugs in the campus network, including forwarding loops and stale ACL rules. 9 of these faults were fixed by the campus network operators. For example, Anteater detected a forwarding loop between a pair of routers that was unintentionally introduced after a network upgrade, and had been present in the network for over a month. These results demonstrate the utility of the approach of data-plane analysis.

Our contributions in this chapter are as follows:

- Anteater comprised the first design and implementation of a data-plane analysis system used to find real bugs in real networks. We used Anteater to find 23 bugs in an operational network.
- We show how to express three key invariants as SAT problems, and propose a novel algorithm for handling packet transformations.

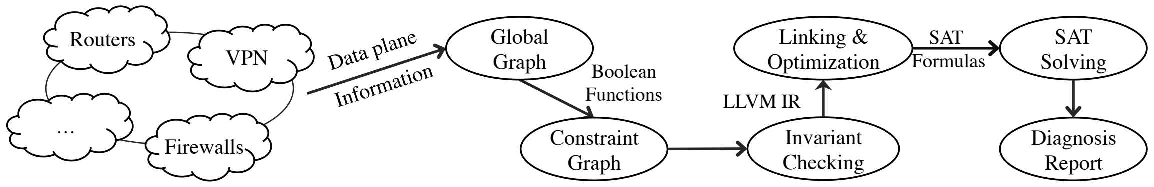


Figure 2.1: The workflow of Anteater. Clouds are network devices. Ovals are stages in the workflow. Text on the edges shows the type of data flowing between stages.

- We develop optimizations to our algorithms and implementation to enable Anteater to check invariants efficiently using a SAT solver, and demonstrate experimentally that Anteater is sufficiently scalable to be a practical tool.

2.1 Overview of architecture

Anteater's primary goal is to detect and diagnose a broad, general class of network problems. The system detects problems by analyzing the contents of forwarding tables contained in routers, switches, firewalls, and other networking equipment (Figure 2.1). Operators use Anteater to check whether the network conforms to a set of *invariants* (i.e., correctness conditions regarding the network's forwarding behavior). Violations of these invariants usually indicate a bug in the network. Here are a few examples of invariants:

- Loop-free forwarding: There should not exist any packet that could be injected into the network that would cause a forwarding loop.
- Connectivity: All computers in the campus network are able to access both the intranet and the Internet, while respecting network policies such as access control lists.
- Consistency: The policies of two replicated routers should have the same forwarding behavior. More concretely, the possible set of packets that can reach the external network through them is the same.

Anteater checks invariants through several steps. First, it collects the contents of FIBs from networking equipment through vtys (terminals), SNMP, or control sessions maintained to routers [33, 55]. These FIBs may be simple IP longest prefix match rules, or more complex actions like access control lists or modifications of the packet header [13, 14, 63]. Second, the operator creates new invariants or selects from a menu of standard invariants to be checked against the network. This is done via bindings in Ruby or in a declarative language that we designed to streamline the expression of invariants. Third, Anteater translates both the FIBs and invariants into instances of SAT, which are resolved by an off-the-shelf SAT solver. Finally, if the results from the SAT solver indicate that the supplied invariants are violated, Anteater will derive a counterexample to help diagnosis.

The next section describes the design and implementation in more detail, including writing invariants, translating the invariants and the network into instances of SAT, and solving them efficiently.

2.2 Modeling network behavior

A SAT problem evaluates a set of boolean formulas to determine if there exists at least one variable assignment such that all formulas evaluate to true. If such an assignment exists, then the set of formulas are satisfiable; otherwise they are unsatisfiable.

SAT is an NP-complete problem. Specialized tools called SAT solvers, however, use heuristics to solve SAT efficiently in some cases [24]. Engineers use SAT solvers in a number of different problem domains, including model checking, hardware verification, and program analysis. I give more details later in this chapter.

Network reachability can, in the general case, also be NP-complete. We cast network reachability and other network invariants as SAT problems. In this section I discuss our model for network policies, and our algorithms for detecting bugs using sets of boolean formulas and a SAT solver.

Anteater uses an existing theoretical algorithm for checking reachability [79], and we use this reachability algorithm to design our own algorithms for detecting forwarding loops, detecting packet loss (i.e., “black holes”), and checking forwarding consistency between routers. Also, we present a novel algorithm for handling arbitrary packet transformations.

Figure 2.2 shows our notation. A network G is a 3-tuple $G = (V, E, \mathcal{P})$, where V is the set of networking devices and possible destinations, E is the set of directed edges representing connections between vertices, and $\mathcal{P}$ is a function defined on E to represent general policies.

Symbol	Description
G	Network graph $(V, E, \mathcal{P})$
V	Vertices (e.g., devices) in G
E	Directed edges in G
$\mathcal{P}$	Policy function for edges

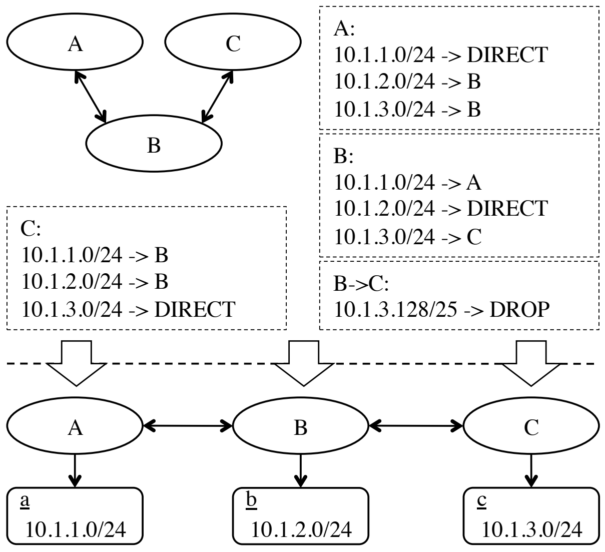
Figure 2.2: Notation used in describing network model.

Since many of the formulas we discuss deal with IP prefix matching, we introduce the notation $var =_{width} prefix$ to simplify our discussion. This notation is a convenient way of writing a boolean formula saying that the first *width* bits of the variable *var* are the same as those of $prefix$. For example, $dst\_ip =_{24}$ 10.1.3.0 is a boolean formula testing the equality between the first 24 bits of *dst_ip* and 10.1.3.0. The notion $var \neq_{width} prefix$ is the negation of $var =_{width} prefix$.

For each edge (u, v), we define $\mathcal{P}(u, v)$ as the policy for packets traveling from u to v, represented as a boolean formula over a symbolic packet. A *symbolic packet* is a set of variables representing the values of fields in packets, like the MAC address, IP address, and port number. A packet can flow over an edge if and only if it satisfies the corresponding boolean formulas. We use this function to represent general policies including forwarding, packet filtering, and transformations of the packet. $\mathcal{P}(u, v)$ is the conjunction (logical *and*) over all policies' constraints on symbolic packets from node u to node v.

$\mathcal{P}(u, v)$ can be used to represent a filter. For example, in Figure 2.3 the filtering rule on edge (B, C) blocks all packets destined to 10.1.3.128/25; thus, $\mathcal{P}(B, C)$ has $dst\_ip \neq_{25}$ 10.1.3.128 as a part of it. Forwarding is represented as a constraint as well: $\mathcal{P}(u, v)$ will be constrained to include only those symbolic packets that router u would forward to router v. The sub-formula $dst\_ip =_{24}$ 10.1.3.0 in $\mathcal{P}(B, C)$ in Figure 2.3 is an example.

Packet transformations — for example, setting a quality of service bit, or tunneling the packet by adding a new header — might appear different since they intuitively modify the symbolic packet rather than just constraining it. Somewhat surprisingly, we can represent transformations as constraints, too, through a technique that we present later.



$$\begin{aligned}
\mathcal{P}(A,a) &= \ dst\_ip =_{24} 10.1.1.0 \\
\mathcal{P}(A,B) &= \ dst\_ip =_{24} 10.1.2.0 \lor dst\_ip =_{24} 10.1.3.0 \\
\mathcal{P}(B,A) &= \ dst\_ip =_{24} 10.1.1.0 \\
\mathcal{P}(B,b) &= \ dst\_ip =_{24} 10.1.2.0 \\
\mathcal{P}(B,C) &= \ dst\_ip =_{24} 10.1.3.0 \land dst\_ip \neq_{25} 10.1.3.128 \\
\mathcal{P}(C,B) &= \ dst\_ip =_{24} 10.1.1.0 \lor dst\_ip =_{24} 10.1.2.0 \\
\mathcal{P}(C,c) &= \ dst\_ip =_{24} 10.1.3.0
\end{aligned}$$

Figure 2.3: An example of a 3-node IP network. *Top:* Network topology, with FIBs in dashed boxes. *Bottom:* graph used to model network behavior. Ovals represent networking equipment; rounded rectangles represent special vertices such as destinations, labeled by lower case letters. The lower half of the bottom figure shows the value of $\mathcal{P}$ for each edge in the graph.

2.2.1 Running reachability queries

Next, we describe how Anteater checks the most basic invariant: reachability. We later use this algorithm to check higher-level invariants.

Recall that vertices V correspond to devices or destinations in the network. Given two vertices $s, t \in V$, we define the s-t reachability problem as deciding whether there exists a packet that can be forwarded from s to t. More formally, the problem is to decide if there exists a symbolic packet p and an $s \rightsquigarrow t$ path such that p satisfies all constraints $\mathcal{P}$ along the edges of the path. Figure 2.4 shows a dynamic programming algorithm to calculate a boolean formula f representing reachability from s to t. The boolean formula

```
function reach$(s, t, k, G)$
$r[t][0] \leftarrow$ true
$r[v][0] \leftarrow$ false for all $v \in V(G) \setminus t$
for $i = 1$ to $k$ do
   for all $v \in V(G) \setminus t$ do
      $r[v][i] \leftarrow \bigvee_{(v,u) \in E(G)} (\mathcal{P}(v,u) \wedge r[u][i-1])$
   end for
end for
return $\bigvee_{1 \leq i \leq k} r[s][i]$
```

Figure 2.4: Algorithm to compute a boolean formula representing reachability from s to t in at most k hops in network graph G.

f has a satisfying assignment if and only if there exists a packet that can be routed from s to t in at most k hops. This part of Anteater is similar to an algorithm proposed by Xie *et al.* [79], expressed as constraints rather than sets of packets. To guarantee that all reachability is discovered, one would pick in the worst case $k = n1$ where n is the number of network devices modeled in G. A much smaller k may suffice in practice because path lengths are expected to be smaller than $n1$. We give an example run of the algorithm for the network of Figure 2.3. Suppose we want to check reachability from A to C. Here, $k = 2$ suffices since there are only 3 devices. Anteater initializes P as shown in Figure 2.3, and the algorithm initializes $s \leftarrow A, t \leftarrow C, k \leftarrow 3, r[C][0] \leftarrow true, r[A][0] \leftarrow false$, and $r[B][0] \leftarrow false$. After the first iteration of the outer loop we have:

$r[A][1] = false$

$r[B][0] = \mathcal{P}(B,C) = dst\_ip =_{24} 10.1.3.0 \wedge dst\_ip =_{25} 10.1.3.128$

$r[B][2] = false$

The algorithm then returns the formula $r[A][1] \vee r[A][2]$.

2.2.2 Checking forwarding loops, packet loss, and consistency

The reachability algorithm can be used as a building block to check other invariants.

```
function loop($v, G$)
$v' \leftarrow$ a new vertex in $V(G)$
for all $(u, v) \in E(G)$ do
    $E(G) \leftarrow E(G) \cup \{(u, v')\}$
    $\mathcal{P}(u, v') \leftarrow \mathcal{P}(u, v)$
end for
Test satisfiability of reach($v$, $v'$, $|V(G)|$, $G$)
```

Figure 2.5: Algorithm to detect forwarding loops involving vertex v in network G.

```
function packet_loss($v, D, G$)
$n \leftarrow$ the number of network devices in $G$
$d \leftarrow$ a new vertex in $V(G)$
for all $u \in D$ do
    $(u, d) \leftarrow$ a new edge in $E(G)$
    $\mathcal{P}(u, d) \leftarrow$ true
end for
$c \leftarrow$ reach($v$, $d$, $n$, $G$)
Test satisfiability of $\neg c$
```

Figure 2.6: Algorithm to check whether packets starting at v are dropped without reaching any of the destinations D in network G.

Loops: Figure 2.5 shows Anteater's algorithm for detecting forwarding loops involving vertex v. The basic idea of the algorithm is to modify the network graph by creating a dummy vertex v' that can receive the same set of packets as v (i.e., v and v' have the same set of incoming edges and edge policies). Thus, $v - v'$ reachability corresponds to a forwarding loop. The algorithm can be run for each vertex v. Anteater thus either verifies that the network is loop-free, or returns an example of a loop.

Packet loss: Another property of interest is whether "black holes" exist: i.e., whether packets may be lost without reaching any destination. Figure 2.6 shows Anteater's algorithm for checking whether packets from a vertex v could be lost before reaching a given set of destinations D, which can be picked as (for example) the set of all local destination prefixes plus external routers. The idea is to add a "sink" vertex d that is reachable from all of D, and then (in the algorithm's last line) test the absence of $v - d$ reachability.

This will produce an example of a packet that is dropped or confirm that none exists. [1] Of course, in some cases packet loss is the correct behavior. For example, in the campus network we tested, some destinations are filtered due to security concerns. Our implementation allows operators to specify lists of IP addresses or other conditions that are intentionally not reachable; Anteater will then look for packets that are unintentionally black-holed. We omit this extension from Figure 2.6 for simplicity.

Consistency: Networks commonly have devices that are expected to have identical forwarding policy, so any differing behavior may indicate a bug. Suppose, for example, that the operator wishes to test if two vertices $v1$ and $v2$ will drop the same set of packets. This can be done by running $packet_loss$ to construct two formulas $c1 = packet_loss(v1, D, G)$ and $c2 = packet_loss(v2, D, G)$, and testing satisfiability of $(c1xorc2)$. This offers the operator a convenient way to find potential bugs without specifically listing the set of packets that are intentionally dropped. Other notions of consistency (e.g., based on reachability to specific destinations) can be computed analogously.

2.2.3 Packet transformation

The discussion in earlier subsections assumed that packets traversing the network remain unchanged. Numerous protocols, however, employ mechanisms that transform packets while they are in flight. For example, MPLS swaps labels, border routers can mark packets to provide QoS services, and packets can be tunneled through virtual links which involves prepending a header. In this subsection, I present a technique that flexibly handles packet transformations.

Basic technique: Rather than working with a single symbolic packet, we use a symbolic packet history. Specifically, we replace each symbolic packet s with an array $(s_0, ..., s_k)$ where s_i represents the state of the packet at the ith hop. Now, rather than transforming a packet, we can express a transformation as a constraint on its history: a packet transformation $f(\cdot)$ at hop i induces the constraint $s_i + 1 = f(si)$. For example, an edge traversed by two MPLS label switched paths with incoming labels l_1^{in},l_2^{in} and corresponding outgoing labels l_1^{out},l_2^{out} would have the transformation constraint

[1] This loss could be due either to black holes or loops. If black holes specifically are desired, then either the loops can be fixed first, or the algorithm can be rerun with instructions to filter the previous results. I omit the details.

$\bigvee_{j\in\{1,2\}} \left(s_i.label = l_j^{in} \wedge s_{i+1}.label = l_j^{out}\right)$

Another transformation could represent a network address translation (NAT) rule, setting an internal source IP address to an external one:

$s_{i+1}.source_ip = 12.34.56.78$

A NAT rule could be non-deterministic, if a snapshot of the NAT's internal state is not available and it may choose from multiple external IP addresses in a certain prefix. This can be represented by a looser constraint:

$s_{+1}.source_ip =_{24} 12.34.56.0$

And of course, a link with no transformation simply induces the identity constraint:

$s_{i+1} = s_i$

We let $T_i(v, w)$ refer to the transformation constraints for packets arriving at v after i hops and continuing to w.

Application to invariant algorithm: Implementing this technique in our earlier reachability algorithm involves two principal changes. First, we must include the transformation constraints T in addition to the policy constraints P. Second, the edge policy function $P(u, v)$, rather than referring to variables in a single symbolic packet s, will be applied to various entries of the symbolic packet array (s_i). Therefore, it is parameterized with the relevant entry index, which we write as $P_i(u, v)$; and when computing reachability we must check the appropriate positions of the array. Incorporating those changes, Line 5 of our reachability algorithm (Figure 2.4) becomes

$r[v][i] \leftarrow \bigvee_{(v,u)\in E(G)} (T_{i-1}(v, u) \wedge P(v, u) \wedge r[u][i-1])$

The loop detection algorithm, as it simply calls reachability as a subroutine, requires no further changes.

The packet loss and consistency algorithms have a complication: as written, they test satisfiability of the *negation of* a reachability formula. The negation can be satisfied either with a symbolic packet that would be lost in the network, or a symbolic packet history that could not have existed because it violates the transformation constraints. We need to differentiate between these, and find only true packet loss.

To do this, we avoid negating the formula. Specifically, we modify the network by adding a node l acting as a sink for lost packets. For each non-destination node u, we add an edge $u \to l$ annotated with the constraint that the packet is dropped by u (i.e., the packet violates the policy constraints on all of u's outgoing edges). We also add an edge $w \to l$ with no constraint, for each destination node $w \notin D$. We can now check for

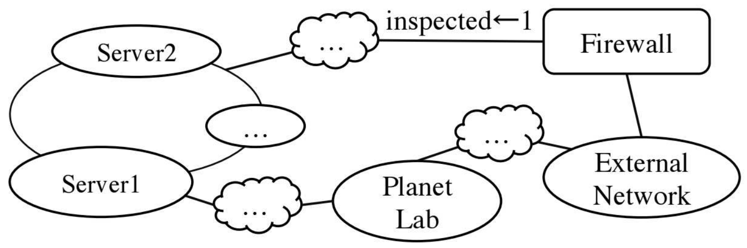


Figure 2.7: An example where packet transformations allow convenient checking of firewall policy. Solid lines are network links; text on the links represents a transformation constraint to express the invariant. Clouds represent omitted components in the network.

packet loss starting at v by testing satisfiability of the formula $reach(v, l, n1, G)$ where n is the number of nodes and G is the network modified as described here.

The consistency algorithm encounters a similar problem due to the XOR operation, and has a similar solution. We note two effects which are not true in the simpler transformation-free case. First, the above packet loss algorithm does not find packets which loop (since they never transit to l); but of course, they can be found separately through our loop-detection algorithm. Second, computing up to $k = n1$ hops does not guarantee that all reachability or loops will be discovered. In the transformation-free case, $k = n1$ was sufficient because after $n1$ hops the packet must either have been delivered or revisited a node, in which case it will loop indefinitely. But transformations allow the state of a packet to change, so revisiting a node does not imply that the packet will loop indefinitely. In theory, packets might travel an arbitrarily large number of hops before being delivered or dropped. However, we expect $k \leq n1$ to be sufficient in practice.

Application to other invariants: Packet transformations enable us to express certain other invariants succinctly. Figure 2.7 shows a simplified version of a real-world example from a campus network. Most servers are connected to the external network via a firewall, but the PlanetLab servers connect to the external network directly. For security purposes, all traffic between campus servers and PlanetLab nodes is routed through the external network, except for administrative links between the PlanetLab nodes and a few trusted servers. One interesting invariant is to check whether all traffic from the external network to protected servers indeed goes through the firewall as intended. This invariant can be expressed conveniently as follows. We introduce a new field inspected in the symbolic packet, and for each edge (f, v) going from the firewall f towards the internal network of servers, we add a transformation constraint:

$T_i(f, v) = s_{i+1}.inspected \leftarrow 1$

Then for each internal server S, we check whether

$(s_k.inspected = 0) \wedge R(ext, S)$

where ext is the node representing the external network, and $R(S, ext)$ is the boolean formula representing reachability from ext to S computed by the reach algorithm. If this formula is true, Anteater will give an example of a packet which circumvents the firewall.

2.3 Complexity

In this section, we discuss the complexity of the basic problem of determining reachability in a network given its data-plane state. The difficulty of determining reachability depends strongly on what functions we allow the data plane to perform. If network devices implement only IP-style longest prefix match forwarding on a destination address, it is fairly easy to show that reachability can be decided in polynomial time. However, if we augment the data plane with richer functions, the problem quickly becomes difficult. As we show below, packet filters make reachability NP-Complete; and of course, reachability is undecidable in the case of allowing arbitrary programs in the data plane.

It is useful to mention how this complexity relates to the approach of Xie *et al.* [79], whose reachability algorithm is essentially the same as ours, but written in terms of set union/intersection operations rather than SAT. As pointed out in [79], even with packet filters, the reachability algorithm terminates within $O(V^3)$ operations. However, this algorithm only calculates a formula representing reachability, and does not evaluate whether that formula is satisfiable. In [79], it was assumed that evaluating the formula (via set operations in the formulation of [79]) would be fast. This may be true in many instances, but in the general case deciding whether one vertex can reach another in the presence of packet filters is *not* in $O(V^3)$, unless $P = NP$. Thus, to handle the general case, the use of SAT or similar techniques is required since the problem is NP-complete. We choose to use an existing SAT solver to leverage optimizations for determining satisfiability.

We now describe in more detail how packet filters make reachability NP-Complete. The input to the reachability problem consists of a directed graph $G = (V, E)$, the boolean policy function $Q(e, p)$ which returns true when packet p can pass along edge e, and two vertices $s, t \in V$. The problem is to decide whether there exists a packet p and an

$s \rightsquigarrow t$ path in G, such that $Q(e, p) = \mathit{true}$ for all edges e along the path. (Note this problem definition does not allow packet transformations.) To complete the definition of the problem, we must specify what sort of packet filters the policy function Q can represent. We could allow the filter to be any boolean expression whose variables are the packet's fields. In this case, the problem can trivially encode arbitrary SAT instances by using a given SAT formula as the policy function along a single edge $s \to t$, with no other nodes or edges in the graph, with the SAT formula's variables being the packet's fields. Thus, that formulation of the reachability problem is NP-Complete. One might wonder whether a simpler, more restricted definition of packet filters makes the problem easy. We now show that even when Q for each edge is a function of *a single bit* in the packet header, the problem is still NP-complete because the complexity can be encoded into the network topology.

Proposition 1. Deciding reachability in a network with single-bit packet filters is NP-Complete.

Proof: Given a packet and a path through the network, since the length of the path must be $< |V|$, we can easily verify in polynomial time whether the packet will be delivered. Therefore, the problem is in NP.

To show NP-hardness, suppose we are given an instance of a 3-SAT problem with n binary variables $x_1, ..., x_n$ and k clauses $C_1, ..., C_k$. Construct an instance of the reachability problem as follows. The packet will have n one-bit fields corresponding to the n variables x_i. We create $k + 1$ nodes $v_0, v_1, ..., v_k$, and we let $s = v_0$ and $t = v_k$. For each clause C_i, we add three parallel edges e_{i1}, e_{i2}, e_{i3} all spanning $v_{i1} \to v_i$. If the first literal in clause C_i is some variable x_i, then the policy function $Q(e_{i1}, p) = \mathit{true}$ if and only if the ith bit of p is 1; otherwise the first literal in C_i is the negated variable x_i, and we let $Q(e_{i1}, p) = \mathit{true}$ if and only if the ith bit of p is 0. The policy functions for e_{i2} and e_i3 are constructed similarly based on the second and third literals in C_i. With the above construction a packet p can flow from v_{i1} to v_i if and only if C_i evaluates to true under the assignment corresponding to p. Therefore, p can flow from s to t if and only if all 3-SAT clauses are satisfied. Thus, since 3-SAT is NP-complete, reachability with single-bit packet filters is NP-complete.

2.4 Implementation

We implemented Anteater on Linux with about 3,500 lines of C++ and Ruby code, along with roughly 300 lines of auxiliary scripts to canonicalize data-plane information from Foundry, Juniper and Cisco routers into a comma-separated value format.

Our Anteater implementation represents boolean functions and formulas in the intermediate representation format of LLVM [56]. LLVM is not essential to Anteater; our invariant algorithms could output SAT formulas directly. But LLVM provides a convenient way to represent SAT formulas as functions, inline these functions, and simplify the resulting formulas.

In particular, Anteater checks an invariant as follows. First, Anteater translates the policy constraints P and the transformation constraints T into LLVM functions, whose arguments are the symbolic packets they are constraining. Then Anteater runs the desired invariant algorithm (reachability, loop detection, etc.), outputting the formula using calls to the P and T functions. The resulting formula is stored in the @*main* function. Next, LLVM links together the P, T , and @*main* functions and optimizes when necessary. The result is translated into SAT formulas, which are passed into a SAT solver. Finally, Anteater invokes the SAT solver and reports the results to the operator. Recall the example presented earlier. We want to check reachability from A to C in Figure 2.3. Anteater translates the policy function $P(B, C)$ into function @$p_b c()$, and puts the result of dynamic programming algorithm into @$main()$:

```
define @p_bc(%si, %si+1) {
%0 = load %si.dst_ip
%1 = and %0, 0xffffff00
%2 = icmp eq 0xa010300, %1
%3 = and %0, 0xffffff80
%4 = icmp eq 0xa010380, %3
%5 = xor %4, true
%6 = and %2, %5
ret %6 }
@pkt = external global
define void @main() {
%0 = call @p_bc(@pkt, @pkt)
%1 = call @p_ab(@pkt, @pkt)
%2 = and %0, %1
call void @assert(%2)
ret void
}
```

The function @p_bc represents the function $P(B, C) = dst\_ip =_{24} 10.1.3.0 \wedge dstip \neq_{25}$ 10.1.3.128 The function takes two parameters $\%si$ and $\%si + 1$ to support packet transformations as described previously. The @$main$ function is shown at the right side of the snippet. @p_ab is the LLVM function representing $P(A, B)$. @pkt is a global variable representing a symbolic packet. Since there is no transformation involved, the main function calls the policy functions @p_bc and @p_ab with the same symbolic packet. The call to @$assert$ indicates the final boolean formula to be checked by the SAT solver. Next, LLVM performs standard compiler optimization, including inlining and simplifying expressions, whose results are shown on the left:

```
define void @main() {
%0 = load @pkt.dst_ip
%1 = and %0, 0xffffff00
%2 = icmp eq %1, 0xa010300
%3 = and %0, 0xffffff80
%4 = icmp ne %3, 0xa010380
%5 = and %2, %4
%6 = and %0, 0xfffffe00
%7 = icmp eq %6, 0xa010200
%8 = and %5, %7
call void @assert(i1 %8)
ret void }
```

```
:formula
(let (?t1 (bvand p0 0xffffff00))
(let (?t2 (= ?t1 0x0a010300))
(let (?t3 (bvand p0 0xffffff80))
(let (?t4 (not (= ?t3 0x0a010380)))
(let (?t5 (and ?t2 ?t4))
(let (?t6 (bvand p0 0xfffffe00))
(let (?t7 (= ?t6 0x0a010200))
(let (?t8 (and ?t5 ?t7))
(?t8)))))))))
```

Then the result is directly translated into the input format of the SAT solver, which is shown in the right. In this example, it is a one-to-one translation except that @*pkt.dst_ip* is renamed to $p0$. After that, Anteater passes the formula into the SAT solver to determine its satisfiability. If the formula is satisfiable, the SAT solver will output an assignment to $pkt.0/p0$, which is a concrete example (the destination IP in this case) of the packet which satisfies the desired constraint. The work flow of checking invariants is similar to that of compiling a large C/C++ project. Thus Anteater uses off-the-shelf solutions (i.e., $make - j16$) to parallelize the checking. Anteater can generate @*main* functions for each instance of the invariant, and check them independently (e.g., for each starting vertex when checking loop-freeness). Parallelism can therefore yield a dramatic speedup. Anteater implements language bindings for both Ruby and SLang, a declarative, Prolog-like domain-specific language that we designed for writing customized invariants, and implemented on top of Ruby-Prolog [11]. Operators can express invariants via either Ruby scripts or SLang queries; we found that both of them are able to express the three invariants efficiently.

2.5 Evaluation

Our evaluation of Anteater has three parts. First, we applied Anteater to a large university campus network. Our tests uncovered multiple faults, including forwarding loops, traffic-blocking ACL rules that were no longer needed, and redundant statically-configured FIB entries.

Second, we evaluate how applicable Anteater is to detecting router software bugs by classifying the reported effects of a random sample of bugs from the Quagga Bugzilla database. We find that the majority of these bugs have the potential to produce effects detectable by Anteater.

Third, we conduct a performance and scalability evaluation of Anteater. While far from

Invariants	Loops	Packet loss	Consistency
Alerts	*9*	*17*	*2*
Being fixed	9	0	0
Stale config.	0	13	1
False pos.	0	4	1
No. of runs	7	6	6

Figure 2.8: Summary of evaluation results of Anteater on a campus network.

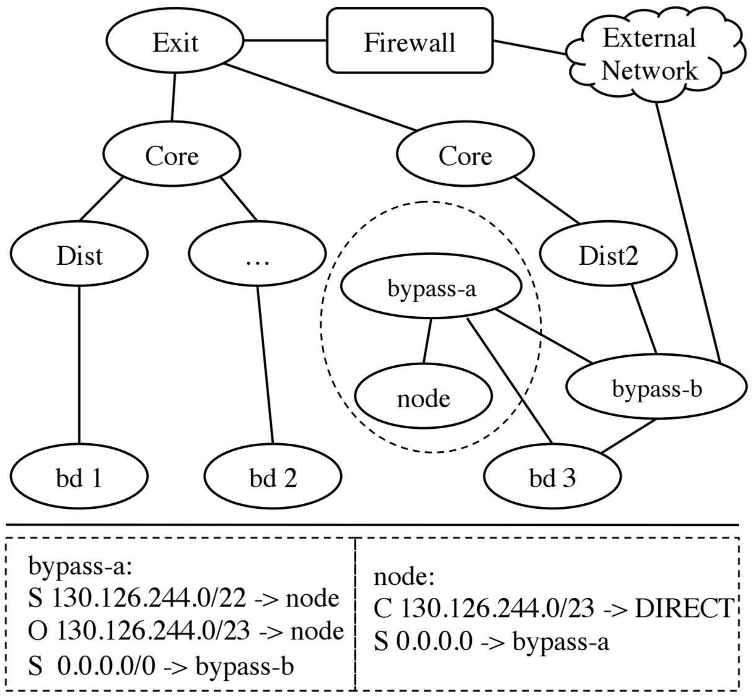
ideal, Anteater takes moderate time (about half an hour) to check for static properties in networks of up to 384 nodes.

We ran all experiments on a Dell Precision T5500 Workstation running 64-bit CentOS 5. The machine had two 2.4 GHz quad-core Intel Xeon X5530 CPUs, and 48 GB of DDR3 RAM. It connected to the campus network via a Gigabit Ethernet channel. Anteater ran on a NFS volume mounted on the machine. The implementation used LLVM 2.9 and JRuby 1.6.2. All SAT queries were resolved by Boolector 1.4.1 with PicoSAT 936 and PrecoSAT 570[24]. All experiments were conducted under 16-way parallelism.

2.5.1 Bugs found in a deployed network

We applied Anteater to a campus network. We collected the IP forwarding tables and access control rules from 178 routers in the campus. The maximal length of loop-free paths in the network is 9. The mean FIB size was 1,627 entries per router, which were inserted by a combination of BGP, OSPF, and static routing. We also used a network-wide map of the campus topology as an additional input.

We implemented the invariants described above, and report their evaluation results on the campus network. Figure 2.8 reports the number of invariant violations we found with Anteater. The row *Alert* shows the number of distinct violations detected by an invariant, as a bug might violate multiple invariants at the same time. For example, a forwarding loop creating a black hole would be detected by both the invariant for detecting forwarding loops and the invariant for detecting packet loss. We classified these alerts into three categories. First, the row *Being fixed* means the alerts are confirmed as bugs and currently being fixed by the campus network operators. Second, the row *Stale configuration* means that these alerts result from explicit and intentional configuration



```
bypass-a:
S 130.126.244.0/22 -> node
O 130.126.244.0/23 -> node
S  0.0.0.0/0 -> bypass-b
```

```
node:
C 130.126.244.0/23 -> DIRECT
S 0.0.0.0 -> bypass-a
```

Figure 2.9: Top: Part of the topology of the campus network. Ovals and solid lines are routers and links respectively. The oval with dashed lines circles the location where a forwarding loop was detected. Bottom: Fragments of data plane information in the network. S stands for static, O stands for OSPF, and C stands for connected.

rules, but rules that are outdated and no longer needed. The campus network operators decided to not fix these stale configurations immediately, but plan to revisit them during the next major network upgrade. Third, *False positive* means that these alerts flag a configuration that correctly reflected the operator's intent and these alerts are not bugs. Finally, *No. of runs* reports the total number of runs required to issue all alerts; the SAT solver reports only one example violation per run. For each run, we filtered the violations found by previous runs and rechecked the invariants until no violations were reported.

Forwarding loops

Anteater detected 9 potential forwarding loops in the network. One of them is shown in Figure 2.9 highlighted by a dashed circle. The loop involved two routers: *node* and $bypass - a$. Router $bypass - a$ had a static route for prefix 130.126.244.0/22 towards router *node*. At the same time, router *node* had a default route towards router $bypass-a$. As shown in the FIBs, according to longest prefix match rules, packets destined to 130.126.244.0/23 from router $bypass-a$ could reach the destination. Packets destined to the prefix 130.126.244.0/22 but not in 130.126.244.0/23 would fall into the forwarding loop.

Incidentally, all 9 loops happened between these two routers. According to the network operator, router $bd3$ used to connect with router *node* directly, and *node* used to connect with the external network. It was a single choke point to aggregate traffic so that the operator could deploy Intrusion Detection and Prevention (IDP) devices at one single point. The IDP device, however, was unable to keep up after the upgrade, so router $bypass-a$ was introduced to offload the traffic. As a side effect, the forwarding loops were also introduced when the operator configured forwarding for that router incorrectly.

These loops are reachable from 64 of 178 routers in the network. All loops were confirmed by the network operator and were fixed.

Packet loss

Anteater issued 17 packet loss alerts, scattered at routers at different levels of hierarchy. One is due to the lack of default routes in the router; three are due to blocking traffic towards unused IP spaces; and the other 13 alerts are because the network blocks traffic towards certain end-hosts. We recognized that four alerts are legitimate operational practice and classified them as false positives. Further investigation of the other 13 alerts shows that they are stale configuration entries: 7 out of 13 are internal IP addresses that were used in the previous generation of the network. The other 6 blocked IP addresses are external, and they are related to security issues. For example, an external IP was blocked in April 2009 because the host made phishing attempts to the campus e-mail system. The block was placed to defend against the attack without increasing the load on the campus firewalls. The operator confirmed that these 13 instances can be dated back as early as September 2008 and they are unnecessary, and probably will be removed during next major network upgrade.

Consistency

Based on conversations with the campus network operators, we know that campus routers in the same level of hierarchy should have identical policies. Hence, we picked one representative router in the hierarchy and checked the consistency between this router and all others at the same level of hierarchy. Anteater issued two new alerts: (1) The two core routers had different policies on IP prefix 10.0.3.0/24; (2) Some building routers had different policies on the private IP address ranges 169.254.0.0/16 and 192.168.0.0/16. Upon investigating the alert we found that one router exposed its web-based management interface through 10.0.3.0/24. The other alert was due to a legacy issue that could be dated back to the early 1990's: according to the design documents of the campus, 169.254.0.0/16 and 192.168.0.0/16 were intended to be only used within one building. Usually each department had only one building and these IP spaces were used in the whole department. As some departments spanned their offices across more than one building, network operators had to maintain compatibility by allowing this traffic to go one level higher in the hierarchy, and let the router at higher level connect them together by creating a virtual LAN for these buildings.

2.5.2 Applicability to router bugs

To evaluate the effectiveness of Anteater's data-plane analysis approach for catching router software bugs, we studied 78 bugs randomly sampled from the Bugzilla repository of Quagga [9]. Quagga is an open-source software router, which is used in both research and production [69]. We studied the same set of bugs presented in [81]. For each bug, we studied whether it could affect the data plane, as well as what invariants are required to detect it. We found 86% (67 out of 78) of the bugs might have visible effects on data plane, and potentially can be detected by Anteater.

60 bugs could be detected by the packet loss detection algorithm, and 46 bugs could be detected by the loop detection algorithm. For example, when under heavy load, Quagga 0.96.5 fails to update the Linux kernel's routing tables after receiving BGP updates (Bug 122). This can result in either black holes or forwarding loops in the data plane, which could be detected by Anteater.

7 bugs can be detected by other network invariants. For example, in Quagga 0.99.5, a BGP session could remain active after it has been shut down in the control plane (Bug

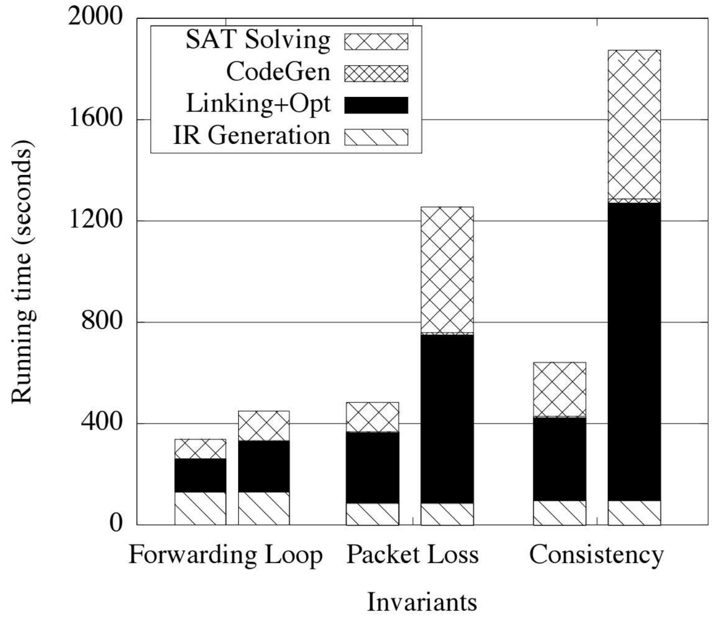


Figure 2.10: Performance of Anteater when checking three invariants. Time is measured by wall-clock seconds. The left and the right column represent the time of the first run and the total running time for each invariant.

416). Therefore, packets would continue to follow the path in the data plane, violating the operator's intent. It is possible to detect this bug via a customized query: checking that there is no data flow across the given link.

11 bugs lack visible effects on the data plane. For example, the terminal hangs in Quagga 0.96.4 during the execution of `show ip bgp` when the data plane has a large number of entries (Bug 87). Anteater is unable to detect this type of bug.

2.5.3 Performance and scalability

Performance on the campus network

Figure 2.10 shows the total running time of Anteater when checking invariants on the campus network. I present both the time spent on the first run and the total time to issue all alerts.

Anteater's running time can be broken into three parts: (a) compiling and executing the invariant checkers to generate an intermediate representation (IR); (b) optimizing the IR with LLVM and generating SAT formulas; (c) running the SAT solver to resolve the SAT queries.

The characteristics of the total running time differ for the three invariants. The reason is that a bug has different impact on each invariant; thus the number of routers needed to be checked during the next run varies greatly. For example, if there exists a forwarding loop in the network for some subnet S, the loop-free forwarding invariant only reports routers which are involved in the forward loop. Routers that remain unreported are proved to loop-free with respect to the snapshot of data plane, provided that the corresponding SAT queries are unsatisfiable. Therefore, in the next run, Anteater only needs to check those routers which are reported to have a loop. The connectivity and consistency invariants, however, could potentially report that packets destined for the loopy subnet S from all routers are lost, due to the loop. That means potentially all routers must be checked during the next run, resulting in longer run time.

Scalability

Scalability on the campus network: To evaluate Anteater's scalability, we scaled down the campus network while honoring its hierarchical structure by removing routers at the lowest layer of the hierarchy first, and continuing upwards until a desired number of nodes remain. Figure 2.11 presents the time spent on the first run when running the forwarding loop invariant on different subsets of the campus network.

Figure 2.12 breaks down the running time for IR generation, linking and optimization, and SAT solving. I omit the time of code generation since we found that it is negligible. Figure 2.12 shows that the running time of these three components are roughly proportional to the square of the number of routers. Interestingly, the running time for SAT solver also roughly fits a quadratic curve, implying that it is able to find heuristics to resolve our queries efficiently for this particular network.

Scalability on synthesized autonomous system (AS) networks: We synthesized FIBs for 6 AS networks (ASes 1221, 1755, 3257, 3967, 4755, 6461) based on topologies from the Rocketfuel project [10], and evaluated the performance of the forwarding loop invariant. We picked $k = 64$ in this experiment. To evaluate how sensitive the invariant is to the complexity of FIB entries, we defined L as a parameter to control the number of "levels"

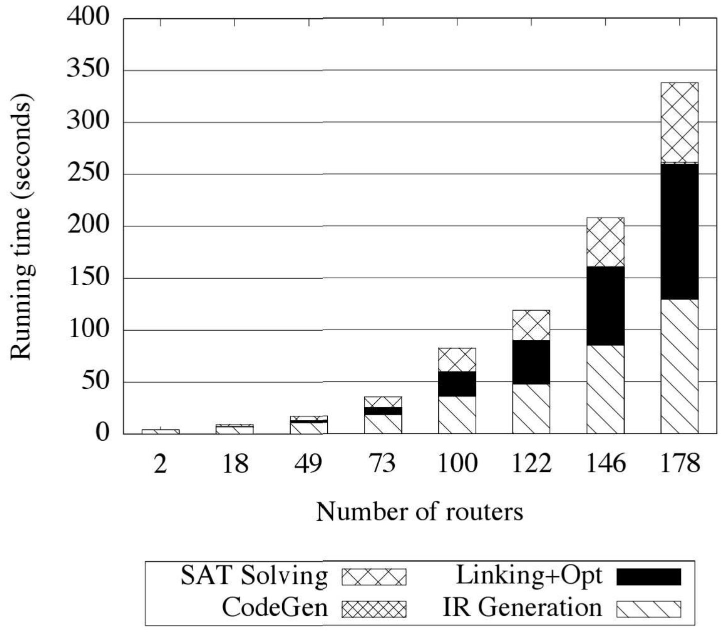


Figure 2.11: Scalability results of the loop-free forwarding invariant on different subsets of the campus network. The parameter k was set to $n1$ for each instance.

of prefixes in the FIBs. When $L = 1$, all prefixes are non-overlapping /16s. When $L = 2$, half of the prefixes (chosen uniform-randomly) are non-overlapping /16s, and each of the remaining prefixes is a sub-prefix of one random prefix from the first half — thus exercising the longest-prefix match functionality. For example, with $L = 2$ and 2 prefixes, we might have $p1$ = 10.1.0.0/16 and $p2$ = 10.1.1.0/24. Figure 2.13 shows Anteater's running time on these generated networks; the $L = 2$ case is only slightly slower than $L = 1$.

It takes about half an hour for Anteater to check the largest network (AS 1221 with 384 vertices). These results have a large degree of freedom: they depend on the complexity of network topology and FIB information, and the running time of SAT solvers depends on both heuristics and random number seeds. These results, though inconclusive, indicate that Anteater might be capable of handling larger production networks.

Scalability on networks with packet transformations: We evaluated the case of the campus network with network address translation (NAT) devices deployed. We manually injected NAT rules into the data in three steps. First, we picked a set of edge routers. For each router R in the set, we created a phantom router R' which only had a bidirectional link to R. Second, we attached a private subnet for each phantom router R', and updated

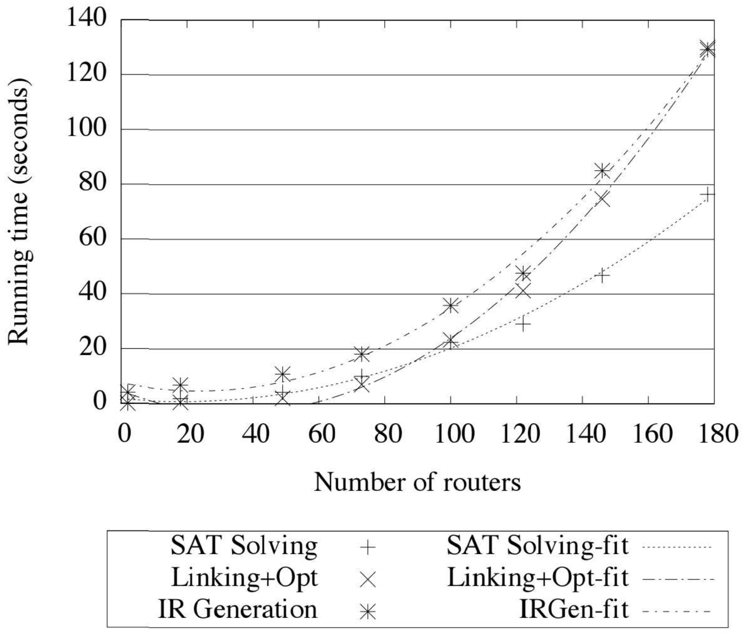


Figure 2.12: Scatter plots for individual components of the data of Figure 2.11. Solid lines are quadratic curves fitted for each category of data points.

the FIBs of both R and R' accordingly for the private subnet. Finally, we added NAT rules as described earlier on the links between R' and R.

Figure 2.14 presents the running time of the first run of the loop-free forwarding invariant as a function of the number of routers involved in NAT. We picked the maximum hops k to be 20 since the maximum length of loop-free paths is 9 in the campus network.

The portion of time spent in IR generation and code generation is consistent among the different number of NAT-enabled routers. The time spent on linking, optimization and SAT solving, however, increases slowly with the number of NAT-enabled routers.

2.6 Discussion

Collecting FIB snapshots in a dynamic network: If FIBs change while they are being collected, then Anteater could receive an inconsistent or incomplete view of the network. This could result in false negatives, false positives, or reports of problems that are only temporary (such as black holes and transient loops during network convergence).

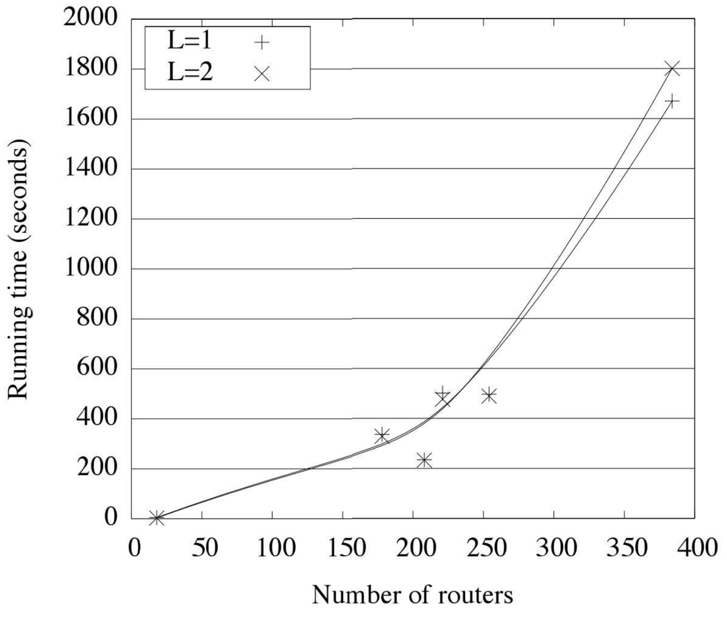


Figure 2.13: Scalability results of the loop-free forwarding invariant on 6 AS networks from [79]. L is the parameter to control the complexity of FIBs. Dots show the running time of the invariant for each network. Solid lines are fitted curves generated from the dots.

There are several ways to deal with this problem. First, one could use a consistent snapshot algorithm [40, 57]. Second, if the network uses a software-defined networking approach [64], forwarding tables can be directly acquired from centralized controllers.

However, our experience shows that the problem of consistent snapshots may not be critical in many networks, as the time required to take a snapshot is small compared to the average time between changes of the FIBs in the campus network. To study the severity of this problem over a longer timespan, we measured the frequency of FIB changes on the Abilene Internet2 IP backbone, by replaying Internet2's BGP and IS-IS update traces to reconstruct the contents of router FIBs over time. BGP was responsible for the majority (93%) of FIB changes. Internal network (IS-IS) changes occurred at an average frequency of just 1.2 events per hour across the network.

We also note that if changes do occur while downloading FIBs, we can avoid a silent failure. In particular, Cisco routers can be configured to send an SNMP trap on a FIB change; if such a trap is registered with the FIB collection device, and received during the FIB collection process, the process may be aborted and restarted.

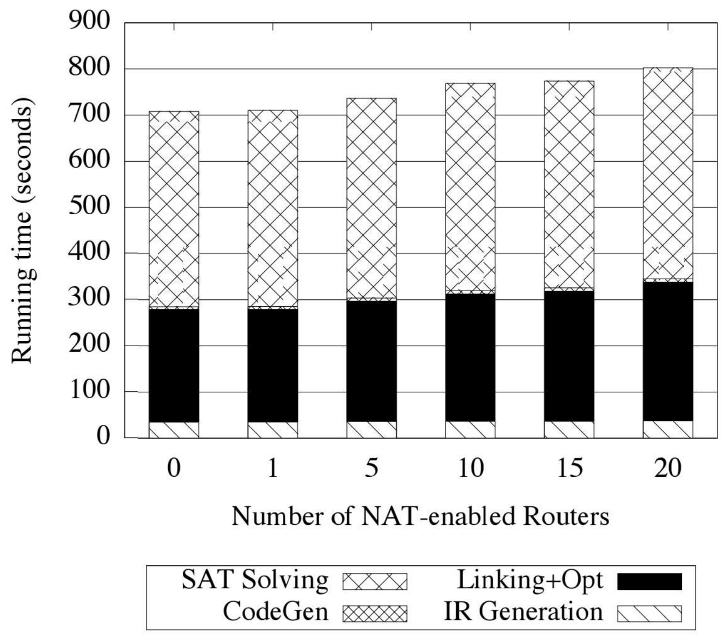


Figure 2.14: Running time of the loop-free forwarding invariant as a function of the number of routers that have NAT rules.

Collecting FIB snapshots in the presence of network failures: Network reachability problems might make acquiring FIBs difficult. Fortunately, Anteater can make use of solutions available today, including maintaining separately tunneled networks at the forwarding plane [55, 33] or operating through out-of-band control circuits [18], in order to gather data-plane state. (More philosophically, we note that if parts of the network are unreachable, then one problem has already been discovered.)

Would using control-plane analysis reduce overhead? Anteater's runtime leaves room for improvement. However, using control-plane analysis in place of Anteater does not address this problem, as the invariants of interest are computationally difficult regardless of whether the information is represented at the control or data plane. It is unclear whether one approach can be fundamentally faster; differences may come down to the choice of which invariants to test, and implementation details. However, we note that the data-plane analysis approach may be easier because unlike control-plane analysis, it need not predict future system inputs or dynamic protocol convergence.

Extending Anteater to handle more general properties: The generality of boolean satisfiability enables Anteater to handle other types of network properties beyond those presented in this paper. For example, Anteater could model network latency by introducing a new field in the symbolic packet to record the packet's total latency, and increasing it at each hop according to the link's latency using our packet transformation algorithms. (The SAT solver we used supports arithmetic operations such as $+$, , $\leq$ that would be useful for representing network behavior and constraints involving latency.)

Of course, some bugs are beyond Anteater's reach, such as those that have no effect on the contents of forwarding state. That includes some hardware failures (e.g., corrupting the contents of the packet during forwarding), and configuration issues that do not affect the FIB.

2.7 Related work

Static analysis of the data plane: The research most closely related to Anteater performs static analysis of data-plane protocols. Xie *et al.* [79] introduced algorithms to check reachability in IP networks with support for ACL policies. Their design was a theoretical proposal without an implementation or evaluation. Anteater uses this algorithm, but we show how to make it practical by designing and implementing our own algorithms to use reachability to check meaningful network invariants, developing a system to make these algorithmically complex operations tractable, and using Anteater on a real network to find 23 real bugs. Xie *et al.* also propose an algorithm for handling packet transformations. However, their proposal did not handle fully general transformations, requiring knowledge of an inverse transform function, and only handling non-loopy paths. Our novel algorithm handles arbitrary packet transformations (without needing the inverse transform). This distinction becomes important for practical protocols that can cause packets to revisit the same node more than once (e.g., MPLS Fast Reroute).

Roscoe *et al.* [72] proposed predicate routing to unify the notions of both routing and firewalling into boolean expressions, Bush and Griffin [25] gave a formal model of integrity (including connectivity and isolation) of virtual private routed networks, and Hamed *et al.* [44] designed algorithms and a system to identify policy conflicts in IPSec, demonstrating bug-finding efficacy in a user study. In contrast, Anteater is a general framework that can be used to check many protocols, and we have demonstrated that it can find bugs in real deployed networks.

Static analysis of control-plane configuration: Analyzing configurations of the control plane, including routers [22, 35] and firewalls [17, 21, 83], can serve as a sanity check prior to deployment. As discussed in the introduction, configuration analysis has two disadvantages. First, it must simulate the behavior of the control plane for the given configuration, making these tools protocol-specific; indeed, the task of parsing configurations is non-trivial and error-prone [60, 81]. Second, configuration analysis will miss non-configuration errors (e.g., errors in router software and inconsistencies between the control plane and data plane [42, 61, 81]; see our study of such errors above).

However, configuration analysis has the potential to detect bugs before a new configuration is deployed. Anteater can detect bugs only once they have affected the data plane — though, as we have shown, there are subtle bugs that fall into this category (e.g., router implementation bugs, copying wrong configurations to routers) that only a data-plane analysis approach like Anteater can detect. Control-plane analysis and Anteater are thus complementary.

Intercepting control-plane dynamics: Monitoring the dynamics of the control plane can detect a broad class of failures [38, 45] with little overhead, but may miss bugs that only affect the data plane. As above, the approach is complementary to ours.

Traffic monitoring: Traffic monitoring is widely used to detect network anomalies as they occur [19, 65, 75, 76]. Anteater's approach is complementary: it can provably detect or rule out certain classes of bugs, and it can detect problems that are not being triggered by currently active flows or that do not cause a statistical anomaly in aggregate traffic flow.

SAT solving in other settings: Work on model checking, hardware verification and program analysis [23, 80, 82] often encounter problems that are NP-Complete. They are often reduced into SAT problems so that SAT solvers can solve them effectively in practice. This work inspired our approach of using SAT solving to model and analyze data-plane behavior.

2.8 Conclusion and future work

We published our work on Anteater in SIGCOMM 2011 [59]. This work is coauthored with Haohui Mai, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey and Samuel T.

King. It is the first design and implementation of a data-plane analysis system that was used to find real bugs in a real network. In future, we plan to improve the running time of Anteater using incremental SAT solving. Currently, Anteater is quite slow is responding to user queries. Depending on the size of the network, Anteater may take a few minutes (sometimes a few hours) to check a certain invariant. By incrementally updating the SAT formulas, and by reusing previous computations, we can improve the running time of Anteater significantly. However, due to the development and strong results obtained from our second tool *VeriFlow*, we decided to fully focus on making VeriFlow a better product that can be used by network operators to detect and diagnose network security and performance vulnerabilities.

CHAPTER 3

REAL-TIME VERIFICATION OF NETWORK-WIDE INVARIANTS

Existing network debugging tools (including Anteater) [59, 49, 16, 15] suffer from slow running time (in the order of seconds to hours) that prevents them from reacting to new network events immediately. Therefore, they are unable to verify network properties in real time as the network state evolves. These previous approaches operate offline, and thus only find bugs after they have already caused damage.

To address this challenge, we develop VeriFlow, which demonstrates that the goal of real-time verification of network-wide invariants is achievable. VeriFlow provides an efficient technique for reasoning about network-wide properties using a low-level view of the network (the data plane), as close as possible to the network's actual behavior.

VeriFlow performs real-time data-plane verification in the context of regular networks and software-defined networks (SDNs). An SDN comprises, at a high level, (1) a standardized and open interface to read and write the data-plane state of network devices; (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 [7] 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.

3.1 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 [59]. 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 [59, 16] (which are generally too slow), we use a custom algorithm. I 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 [63], 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, in an SDN VeriFlow is implemented as a shim layer between the controller and the network, and monitors all communication in either direction. In non-SDN networks, VeriFlow can receive data-plane updates through SNMP or by periodically polling the devices. However, in this case, VeriFlow will not be able to act as a proactive verifier, but still fast enough to detect problems before they can be exploited by outside attackers.

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 [15, 59, 49], 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 (§ 3.1.1). 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 § 3.3.1). 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 (§ 3.1.2). Third, VeriFlow traverses these graphs (or runs custom user-defined code) to determine the status of one or more invariants (§ 3.1.4). The following subsections describe these steps in detail. Figure 3.1 shows the placement and operations of VeriFlow in an SDN.

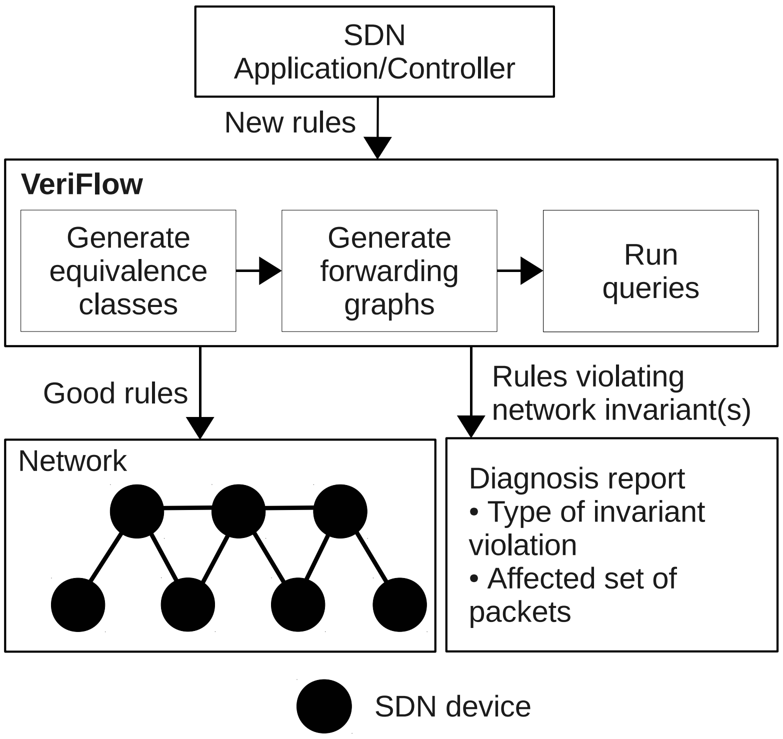

SDN Application/Controller
New rules
VeriFlow
Generate equivalence classes
Generate forwarding graphs
Run queries
Good rules
Rules violating network invariant(s)
Network
Diagnosis report
• Type of invariant violation
• Affected set of packets
SDN device

Figure 3.1: VeriFlow sits between the SDN applications and devices to intercept and check every rule entering the network.

3.1.1 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 [15, 59]. 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 [7] (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
- 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 [77].

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.[1] (Note that an optimization in our implementation uses a condensed set of fields in the trie; see § 3.2.2.)

[1](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).

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 3.2). 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 3.2).

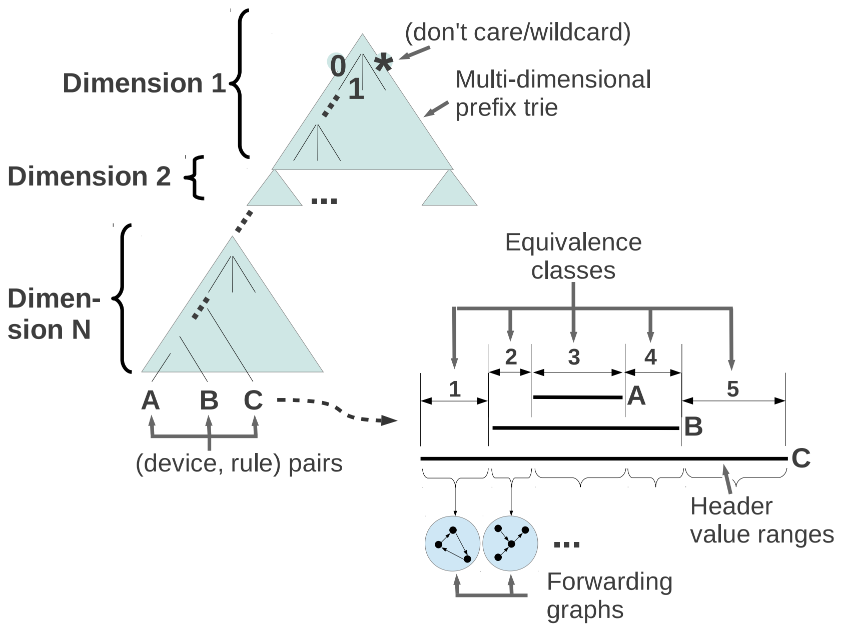


Figure 3.2: VeriFlow's core algorithmic process.

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 3.2. 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 3.2 could have been combined into a single EC. However, this method performs well in practice. More details are shown in Figure 3.3, Algorithm 1, and Algorithm 2.

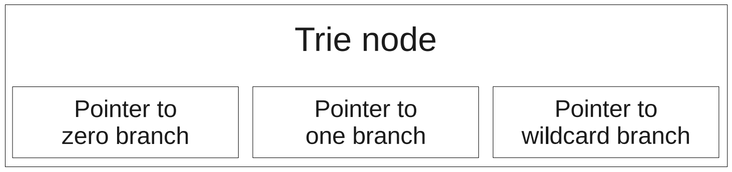


Figure 3.3: Structure of a trie node.

Algorithm 1 Add a rule in the trie.

```
Require: Rule to add
  $node \leftarrow root$
  Check all rule-bits starting from left
  while There are more bits do
    if $bit = 0$ then
      $node \leftarrow node.zero\_branch$
    else if $bit = 1$ then
      $node \leftarrow node.one\_branch$
    else
      $node \leftarrow node.wildcard\_branch$
    end if
  end while
  Put the rule in $node$
```

We consider overlapping rules at all the devices because inserting a rule at a switch can alter the entire path taken by packets that match the new rule. In addition to this, there can be other non-overlapping rules at the local switch that need to be considered – for example if the instruction changes the destination IP address of the packet and sends it to another flow table at the local switch. To handle this, Algorithm 2 needs to be used multiple times to process all the rules that can influence the forwarding behavior of a packet. This process is repeated until no additional affected rules are found. Since OpenFlow instructions are very simple as compared to general code, this approach does not run into state explosion problems that can come up with symbolically executing general software [26, 28, 67]. More details on VeriFlow's performance are given in Section 3.3.

Algorithm 2 Find overlapping rules.

```
Require: New rule
  $level = \varnothing$
  $level += root$
  Check all rule-bits starting from left
  while There are more bits do
    while $level$ is not empty do
      $node \leftarrow level.next\_element$
      if $bit = 0$ then
        $next\_level$ += $node.zero\_branch$
        $next\_level$ += $node.wildcard\_branch$
      else if $bit = 1$ then
        $next\_level$ += $node.one\_branch$
        $next\_level$ += $node.wildcard\_branch$
      else
        $next\_level$ += $node.zero\_branch$
        $next\_level$ += $node.one\_branch$
        $next\_level$ += $node.wildcard\_branch$
      end if
    end while
    $level \leftarrow next\_level$
    $next\_level.clear$
  end while
  return All the rules stored by the nodes present in $level$
```

3.1.2 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.

3.1.3 Handling packet transformations

Our first version of VeriFlow did not handle packet transformations. It assumed that a packet (or, packet set) does not change its signature as it traverses from the source to the destination. This is not true in real networks, because the packet header can get transformed in different ways. For example, as the packet moves from one device to the next, the source and destination MAC addresses of the packet change. This change happens at every hop. Also, devices like NAT (Network Address Translation) boxes and MPLS switches update the packet header as the packet passes through them. We updated the VeriFlow algorithm to handle packet transformations. In this section, we present a brief overview of our packet transformation algorithm.

As packets can experience header change at every hop, we can no longer compute network-wide equivalence classes by traversing our trie structure just once. We need to traverse the trie multiple times on a device-by-device basis. Also, we now build the forwarding graph incrementally as we compute packet sets at each device. Below are the steps we follow to compute the forwarding graph that get affected when a new rule is inserted into the network.

- We start with the device where the new rule is inserted, and find all the local equivalence classes in that device affected by the new rule.
- For each affected equivalence class, we get the highest priority rule, and check the action list associated with that rule. If the action list contains transformation actions (e.g., SET_DL_DST or SET_DL_VLAN in case of OpenFlow), we apply those transformations to update the corresponding header field values of the equivalence class.
- After applying all the transformation actions, we check the next hop device information in the output action (if there is one), and use it to traverse the trie a second time. This time we use the current transformed equivalence class to get all the overlapping local equivalence classes at the next hop device.
- We apply this technique until we reach a device that consumes the packet set, or that we have already visited for a previously encountered packet set (which also indicates presence of a routing loop).

Although our new algorithm has to traverse the trie structure multiple times, our experiments show that the running time of VeriFlow is not affected much given this change. This is due to the fast lookup performance of our trie structure.

3.1.4 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 § 3.2.3, so that new invariants can be written and plugged in. We provide a list of invariants in

Chapter 4 that we found useful for network operators to ensure correct network behavior, and can be easily implemented in VeriFlow.

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. However, 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.

3.1.5 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 (§ 3.3) 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).

3.2 Implementation

We describe three key aspects of our implementation: our shim layer to intercept network events (§ 3.2.1), an optimization to accelerate verification (§ 3.2.2), and our API for custom invariants (§ 3.2.3).

3.2.1 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* [74] 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 [43] to improve performance; our performance evaluation is of this version. We expect one could similarly integrate VeriFlow with other controllers, such as Floodlight [3], Beacon [1] and Maestro [27], 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. VeriFlow 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.

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.

3.2.2 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.[2] 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

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

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.

3.2.3 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 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 (§ 3.2.2). 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.

3.3 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 (§ 3.3.1), 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 (§ 3.3.2) is to assess the impact of VeriFlow on TCP connection setup latency and throughput as perceived by end users of an SDN.

Please note that most of the results presented in this section have been obtained through the earlier version of VeriFlow that does not support packet transformation. However, we also re-ran one of our main experiments using the new version of VeriFlow, and found that VeriFlow's performance does not degrade much due to the changes in the algorithm. Result from this new experiment is presented in § 3.3.1.

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.

3.3.1 Per-update processing time

In this experiment, we simulated a network consisting of 172 routers following a Rocketfuel [10] topology (AS 1755), and replayed BGP (Border Gateway Protocol) RIB (Routing Information Base) and update traces collected from the Route Views Project [12]. 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 § 3.1. 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 3.4(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 observed similar results when we applied VeriFlow in two real world operational networks.

We ran the same experiment using the new version of VeriFlow that supports packet transformation. Results from this experiment are shown in Figure 3.5. From this figure we observe that the total verification time of VeriFlow is still below 1ms for most of the updates, with mean verification time of 0.59ms (a 55% increase compared to the previous algorithm).

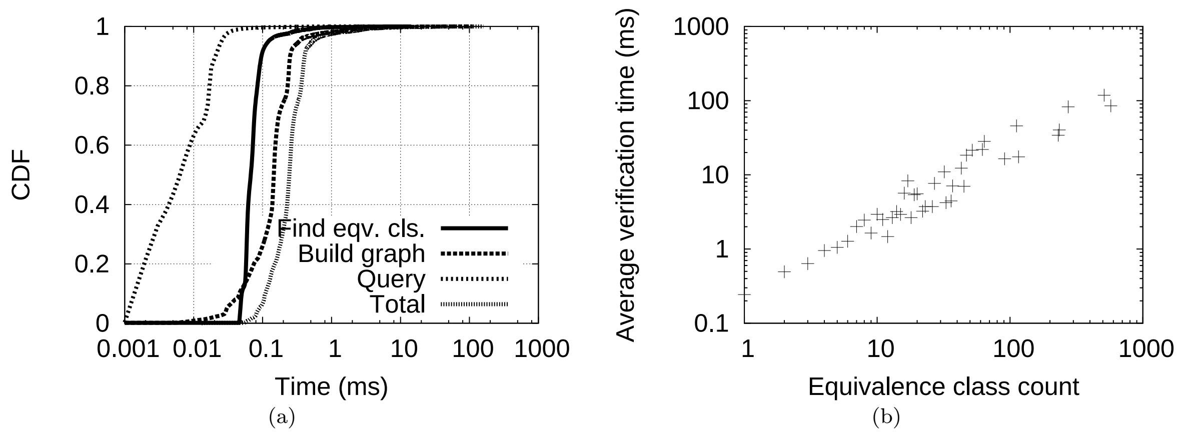


Figure 3.4: Per-update processing times: (a) Microbenchmark results, using the Route Views trace. Total verification time of VeriFlow remained below 1ms for 97.8% of the updates. (b) Scatter plot showing the influence of number of equivalence classes on verification time.

We found that the number of ECs that are affected by a new rule strongly influences verification time. The scatter plot of Figure 3.4(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 looking at the long tail of Figure 3.4(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

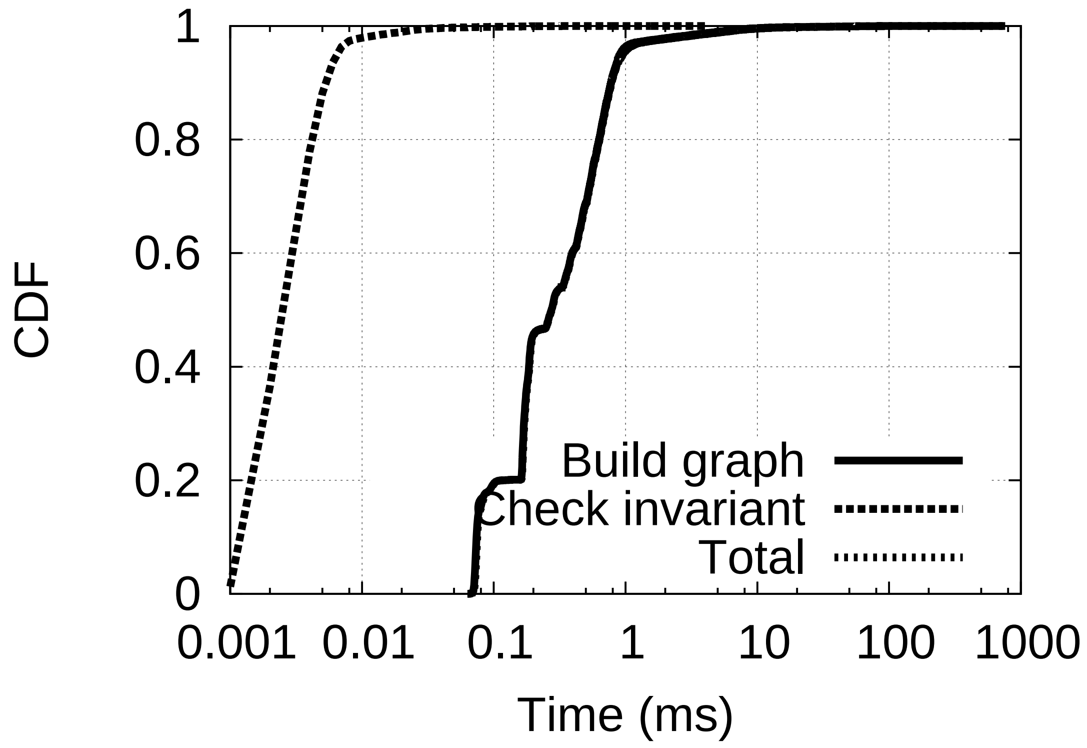


Figure 3.5: Microbenchmark results, using the Route Views trace and the updated algorithm that supports packet transformation. Total verification time of VeriFlow still remains below 1ms for 96% the updates.

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 3.6 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 3.6 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 3.1 shows the results from this experiment.

From Table 3.1, 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).

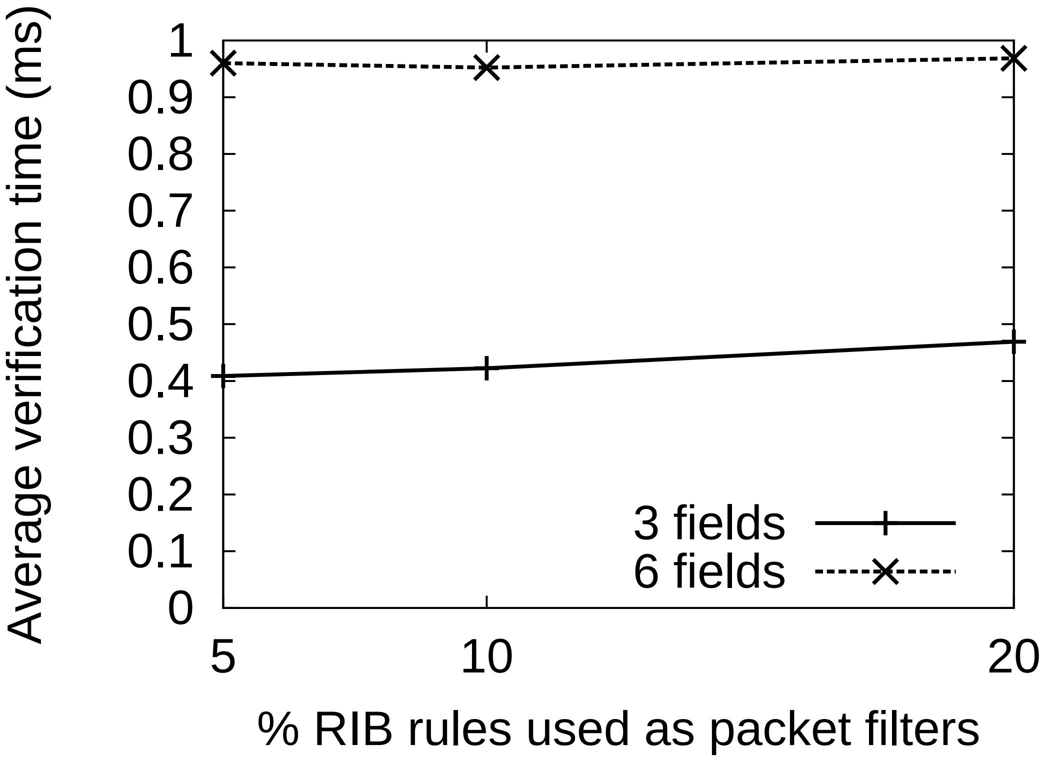


Figure 3.6: Results from multi-field packet filter experiment using the Route Views trace. As more fields are used in forwarding rules, the running time of VeriFlow increases. The average verification latency is not significantly influenced as we increase the number of filters present in the network.

Table 3.1: 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

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 3.7.

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 [49]. 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.

3.3.2 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 [4].

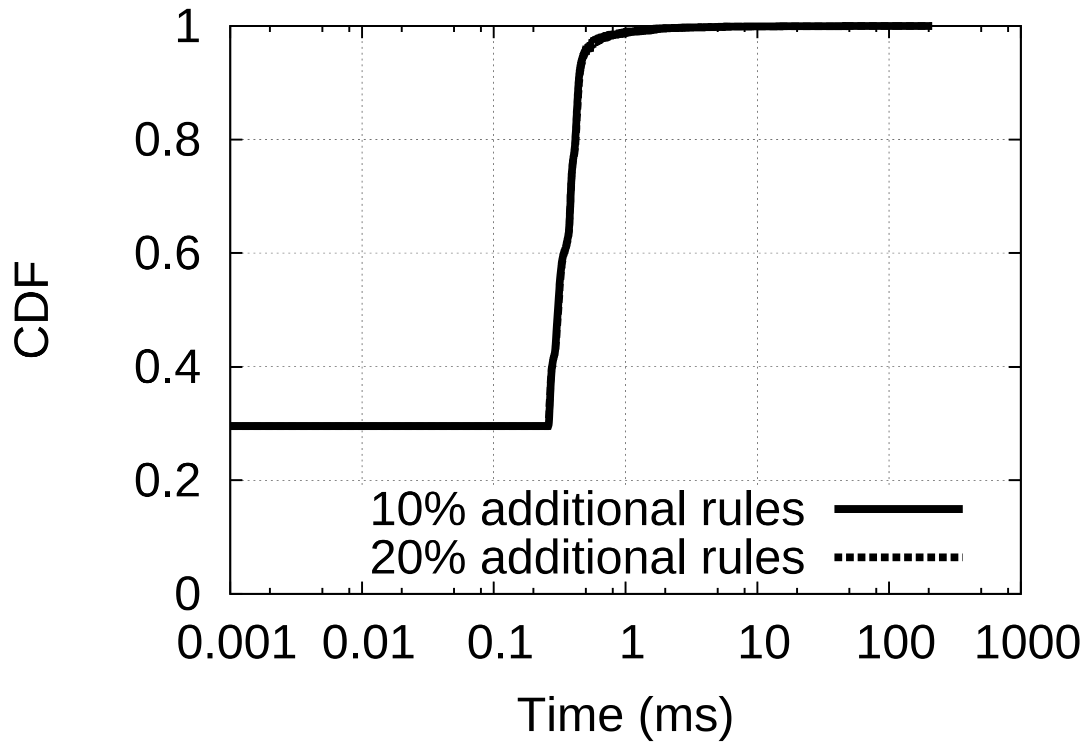
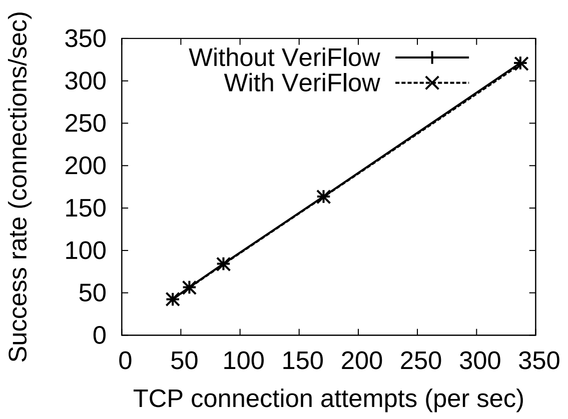
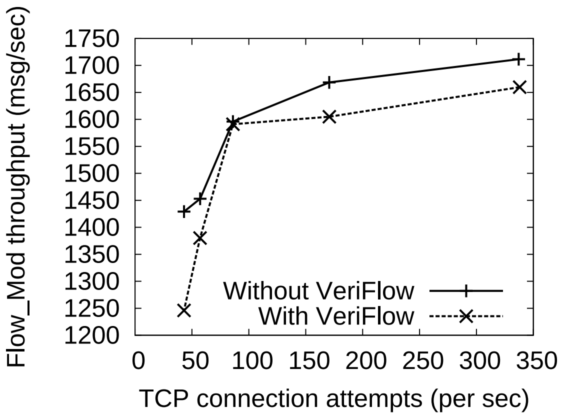


Figure 3.7: Results from the conflict detection test. VeriFlow is fast enough to compute all the conflicting rules within hundreds of microseconds for 99% of the updates.

Mininet creates 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)



(b)

Figure 3.8: 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.

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 [6]) 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. Moreover, in this experiment, we allocated one core of the CPU to NOX and the remaining 7 cores to Mininet using the *taskset* command.

Figure 3.8(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 3.8(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 3.9 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.

3.3.3 Comparison with related work

Finally, we compared performance of our technique with two pieces of related work: the Hassel tool presented in [49] (provided to us by the authors), and a BDD-based analysis tool that we implemented from scratch following the strategy presented in [16]

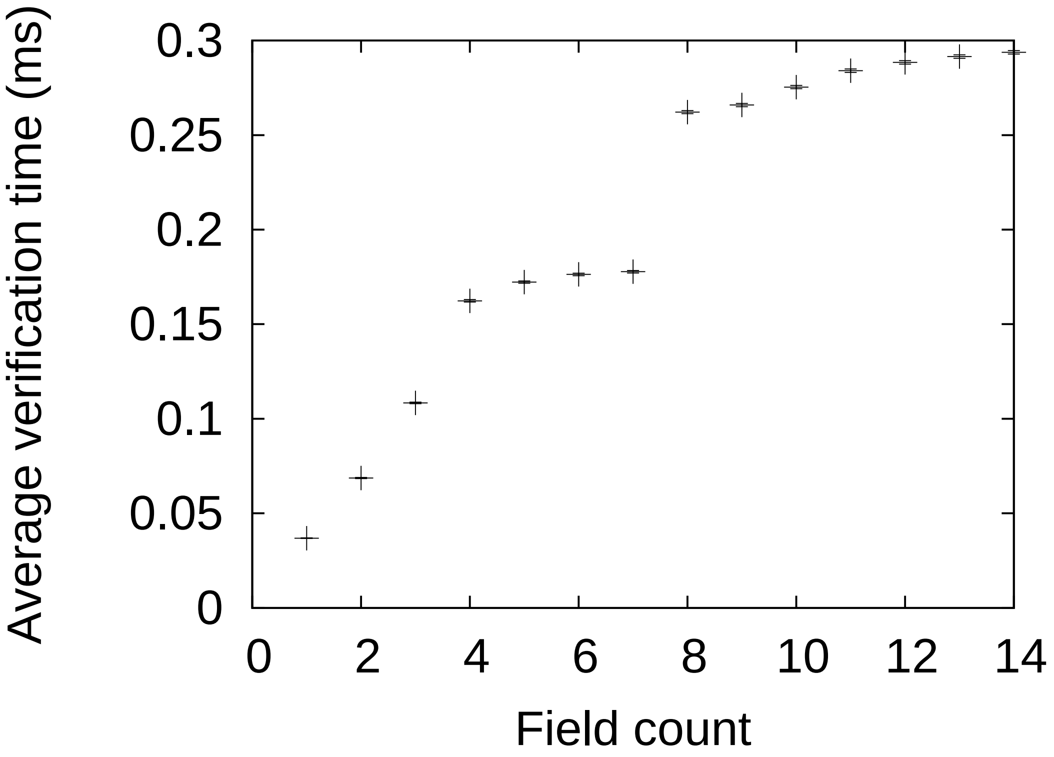


Figure 3.9: Effect of the number of packet header fields on VeriFlow's verification speed. As we increase the number of fields, overhead of VeriFlow increases gradually.

(the original code was not available to us). The authors of [49] 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 data-plane state that was used in [49]. 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 [5] 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 [16]. 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.

3.3.4 Bugs found in a real network

We applied VeriFlow to check reachability issues in an operational network. This network consists of around 250 routers with Cisco, HP, Brocade and Juniper devices. We ran a bunch of *show* commands in all these devices to collect the routing table and interface information in order to build the data-plane model inside VeriFlow. Then we used our loop invariant checker to check for infinite loops in the network.

To our surprise, VeriFlow reported that there are 17 ranges of IP destination addresses that may cause loop in the network. We verified this through the *traceroute* [46] tool. All the addresses were found to be looping between two core devices in the network. We tested this both from withing the network and from outside the network. This error proved to be a serious one because this makes the network vulnerable to Distributed Denial of Service (DDoS) attacks. By sending traffic to these addresses with a high TTL value, an attacker can overload a critical link causing disruption to Internet access. We shared this information with the network operators managing this network, and they informed us that this error was due to a change in their router configuration that was made a couple of months ago. That means this network was vulnerable to attacks for more than 2 months. VeriFlow pinpointed the pair of devices involved and the rules causing this loop, and helped the operator to quickly remove the faulty rule from the configuration.

3.4 Discussion

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, the number of flows on a link always remains below a threshold, two VLANs do not traverse the same set of devices ensuring performance isolation, or bandwidth usage of a set of links always remains below a certain threshold. These checks can be performed by annotating the new rules and the network devices with additional attributes representing their resource requirements/constraints, and putting these attributes in the forwarding graphs of the affected equivalence classes. We also need to supply real-time traffic information to VeriFlow to perform these queries. Moreover, as mentioned in Section 3.1.4, network operators can check other properties using the VeriFlow API; performance may, however, depend on the property being checked and its implementation.

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 [70, 62], where there are well-defined stages during which the network state is consistent and can be checked.

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.

3.5 Related work

Programming OpenFlow networks: NOX [43] 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 SDN applications to register for network-triggered events, and to send OpenFlow commands to the switches. Frenetic [39] 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, Frenetic 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. Therefore, VeriFlow is orthogonal to these works as it can detect violations of network invariants while applications are running on a real network.

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 [28] 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 [74] 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 [66], the authors presented two algorithms to detect conflicting rules in a virtualized OpenFlow network. In another work [68], 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.* [70] proposed a technique that uses an idea similar to the one proposed in [47]. 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 [62] 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) [34] 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 [59] for examples).

Concurrent with our work on VeriFlow, NetPlumber [48] is a tool based on Header Space Analysis (HSA) [49] 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 [16] and FlowChecker [15] 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.

Monitoring network traffic: The traditional approach to network traffic monitoring focuses on statistical monitoring at a per-flow or per-interface granularity. Sampled NetFlow [2] with sampling parameter N (one out of every N packets is sampled) is used to detect and monitor large flows. SNMP is used to collect link-based statistics, and sometimes used in conjunction with NetFlow to detect traffic volume anomalies [54]. cSAMP [73] improves upon NetFlow with hash-based coordination and network-wide optimization to eliminate redundant reports without explicit communication between routers. While these approaches are useful for accounting [30], traffic engineering [37], and volume anomaly detection, they cannot be used to proactively detect faults in network behavior in real time. Similarly, penetration testing [8], while useful in finding security weaknesses in network systems, is not rigorous and not proactive.

Finally, a number of more recent works have identified and solved important related problems in the field of network verification, some of which build upon our approaches. Some examples of recent work include the following. VeriCon [20] performs verification on network controller programs, ensuring that an SDN program is correct on all admissible topologies and all possible sequences of network events. VeriCon uses first-order logic to specify admissible network topologies, and was shown to be effective on a variety of simple controller programs. Libra [84] provides a mechanism to verify forwarding tables in networks using a MapReduce-style divide and conquer approach. RINC [41] performs real-time diagnosis of network through a series of increasingly more fine-grained measurements. RINC provides a simple query interface to easily write diagnosis applications.

3.6 Current status and future work

We first published an initial version of this work in HotSDN 2012 [50]. This paper was selected as one of the two top papers presented at that workshop, and was republished in ACM SIGCOMM Computer Communication Review [51]. Later, we published an extended version of this work in NSDI 2013 [52]. A summarized version of this work was also presented at ONS 2013 [53]. This work is coauthored with Xuan Zou, Wenxuan Zhou, Matthew Caesar and P. Brighten Godfrey.

We are currently working on some interesting directions to extend VeriFlow. We plan to build a distributed version of VeriFlow to deal with SDNs with multiple distributed controllers. The idea is to share minimal amount of information among different instances of VeriFlow so that we can check global invariants quickly. We also plan to improve VeriFlow's intelligence by providing the network operators a quantitative measurement on the severity of each reported error event. This will allow network operators to judiciously decide which errors need immediate attention, and which errors can be safely ignored (such as transient black holes).

We believe the real-time nature of our system, coupled with its ability to provide strong guarantees on the ability to detect and localize complex faults, may enable it to become a standard best practice for future network operations, and an essential tool for highly dependable networks. Our system also provides a general platform for reasoning about the behavior of network protocols, which we believe will make it a valuable tool for building and testing network monitoring and management applications.

CHAPTER 4

LIST OF USEFUL INVARIANTS

Networks operators want their network to be properly configured to ensure required reachability between pairs of devices. However, in some cases non-reachability is something that is needed. For example, in a multi-tenant cloud environment, virtual machines belonging to two customers should not be able to communicate with each other; guest devices in an enterprise network should not be able to communicate with internal servers. VeriFlow can help operators to ensure that certain properties in their network always hold true. This chapter presents a list of invariants or policy checkers that can be easily implemented using VeriFlow. The device independent graph model generated by VeriFlow makes invariant development very easy. In the following sections, I provide short description of a few invariants that I found quite useful for network operators. I discussed these with a few operators, and all of them expressed the need of such high-level policy checkers to find problems in their networks.

4.1 Basic reachability

The purpose of the *basic reachability* invariant is to check reachability of a packet (or packet set) from a source device to its ultimate destination. In VeriFlow, the verification function traverses the directed edges in the forwarding graph (using depth-first search in our implementation) and checks at every hop whether the current device consumes the packet or not.

4.2 Loop-freeness

Routing loops are detrimental to network performance. It not only consumes bandwidth unnecessarily but also opens doors for Distributed Denial of Service (DDoS) attacks. Using VeriFlow we can check for existence of both finite and infinite loops. As we show in Section 3.3.4, this invariant was able to detect multiple IP destination ranges that would have fallen into infinite loops (limited by their TTL values) between two important devices in an operational network.

4.3 Black hole detection

Black holes are caused by routers dropping packets. A router can drop packets for multiple reasons. First, there can be an explicit Access Control List (ACL) rule dropping a certain set of packets. Second, if the network operator forgets to put a "permit all" statement at the end of an ACL, then all packets not matching the earlier entries in that ACL will get dropped. Third, if a router does not have its default gateway configured, then any packet for which it does not have a routing table entry will get dropped. The job of the *black hole detection* invariant is to report all these cases. It reports the device where the drop will occur along with the rule (or absence of rule) causing the drop.

4.4 Consistency

Sometimes two or more devices are deployed in a network for sharing load or to improve fault tolerance. A packet (or packet set) crossing one of these devices should end up at the same destination irrespective of which of these devices it crossed. The *consistency* invariant checks this policy. Given two (or more) routers R_1, R_2 that are intended to have identical forwarding operations, this verification function traverses the forwarding graph starting at both R_1 and R_2 to test whether the fate of packets is the same in both cases. Any difference may indicate a bug.

4.5 VLAN isolation

Packets from one VLAN should not leak into another VLAN unless there is a layer-3 entity involved to perform the routing. The *VLAN isolation* invariant checks whether inter-VLAN communication is possible or not. The verification function keeps track of all the VLANs traversed by a packet set, and reports a violation if a path from the *from* VLAN to the *to* VLAN is detected.

4.6 Loose path routing

The *loose path routing* invariant checks whether a packet set starting from a source device passes through a set of devices or not. Device ordering is not enforced here. It can be used to ensure that certain sets of packets always go though a set of monitoring points.

4.7 Strict path routing

The *strict path routing* invariant also checks whether a packet set starting from a source device passes through a set of devices or not. However, device ordering is enforced here. It can be used to ensure that certain sets of packets strictly follows a path from source to destination. For example, network operators can use this invariant to make sure that packets from the Internet always visit the firewall before entering the local network.

4.8 Path length constraint

The *path length constraint* invariant checks whether a packet set reaches its destination within a configured number of hops or not. By destination, we mean a device that consumes the packet. It can be used to ensure that packet are not unnecessarily bouncing around in the network before reaching their intended destinations.

4.9 Detect overlapping rules

When two or more rules cover the same packet header range, then their priority decides which one will be used to forward the matching packets. In regular networks, the priority is determined by longest prefix matching. In SDNs, a priority field is used for this purpose. However, sometimes it is useful to check beforehand whether a new rule will overlap with an existing rule or not. The trie structure maintained by VeriFlow to store all the rules makes it fast and easy to check for such overlaps.

4.10 Change of next hop

Two or more overlapping rules can have different next hop values. Therefore, depending on the priority of these rules, the matching packets can change their path when a new overlapping rule with higher priority is added in the device. The *change of next hop* invariant checks for such cases whenever the data-plane state of the network changes.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, my primary goal is to develop a robust network monitoring and debugging framework that will help network operators to run their networks efficiently and securely. In order to provide accurate information about network events and possible errors that may affect packet traversal, we utilize data-plane state to remain as close as possible to the network's actual behavior. Our previous and on-going works in this area have shown promising results both in terms of error detection capability and efficiency. Our tool, Anteater, was the first one to be applied in a real network, and detected multiple real bugs. Our next tool, VeriFlow, was the first tool capable of verifying network-wide invariants in real time. VeriFlow leverages a set of efficient algorithms to check rule modification events in real time before they are sent to the live network. 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.

I am currently focusing on extending VeriFlow to make it more general and efficient as described in Section 3.6. Eventually, I plan to develop a network monitoring and debugging software that can work seamlessly with both general networks and SDNs. I will utilize the techniques developed in this thesis to build the different components of this software suite.

REFERENCES

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

[2] Cisco NetFlow. http://www.cisco.com/web/go/netflow.

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

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

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

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

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

[8] Penetration testing overview. http://www.coresecurity.com/penetration-testing-overview.

[9] Quagga software routing suite. http://www.quagga.net.

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

[11] Ruby-Prolog. https://rubyforge.org/projects/ruby-prolog.

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

[13] JUNOS: MPLS fast reroute solutions, network operations guide.

[14] The all new 2010 intel core vpro processor family: Intelligence that adapts to your needs (whitepaper). ftp://download.intel.com/products/vpro/whitepaper/crossclient.pdf.

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

[16] 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).

[17] Al-Shaer, E. S., and Hamed, H. H. Discovery of policy anomalies in distributed firewalls. In *INFOCOM* (2004).

[18] Apple. What is lights out management? `https://support.apple.com/kb/TA24506`.

[19] Baccelli, F., Machiraju, S., Veitch, D., and Bolot, J. The role of PASTA in network measurement. *ACM SIGCOMM* (September 2006).

[20] Ball, T., Bjorner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Shapira, M., and Valadarsky, A. VeriCon: towards verifying controller programs in software-defined networks. *PLDI* (June 2014).

[21] Bartal, Y., Mayer, A., Nissim, K., and Wool, A. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy* (1999).

[22] Benson, T., Akella, A., and Maltz, D. Unraveling the complexity of network management. In *NSDI* (2009).

[23] Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y. Symbolic model checking without BDDs.

[24] Brummayer, R., and Biere, A. Boolector: An efficient SMT solver for bit-vectors and arrays. *TACACS* (1999).

[25] Bush, R., and Griffin, T. Integrity for virtual private routed networks. *IEEE INFOCOM* (2003).

[26] Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *OSDI* (December 2008).

[27] 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`.

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

[29] Cisco Systems Inc. Spanning tree protocol problems and related design considerations, August 2005. `http://www.cisco.com/en/US/tech/tk389/tk621/technologies_tech_note09186a00800951ac.shtml`.

[30] Duffield, M., Lund, C., and Thorup, M. Charging from sampled network usage. In *IMC* (2001).

[31] Duffy, J. BGP bug bites Juniper software. *Network World* (December 2007).

[32] Evers, J. Trio of Cisco flaws may threaten networks. *CNET News* (January 2007).

[33] Farinacci, D., Li, T., Hanks, S., Meyer, D., and Traina, P. Generic routing encapsulation (gre). RFC 2784, March 2000.

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

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

[36] Feamster, N., and Rexford, J. Network-wide prediction of BGP routes. *IEEE/ACM Transactions on Networking 15* (2007), 253–266.

[37] Feldmann, A., Greenberg, A., Lund, C., Reingold, N., Rexford, J., and True, F. Deriving traffic demands for operational IP networks: Methodology and experience. In *IEEE/ACM Transactions on Networking* (2001).

[38] Feldmann, A., Maennel, O., Mao, Z., Berger, A., and Maggs, B. Locating Internet routing instabilities. *ACM SIGCOMM* (August 2004).

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

[40] Geels, D., Altekar, G., Maniatis, P., Roscoe, T., and Stoica, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007).

[41] Ghasemi, M., Benson, T., and Rexford, J. RINC: Real-time inference-based network diagnosis in the cloud. *Under Submission* (2015).

[42] Goodell, G., Aiello, W., Griffin, T., Ioannidis, J., McDaniel, P., and Rubin, A. Working around BGP: An incremental approach to improving security and accuracy of interdomain routing. In *Proc. NDSS* (2003).

[43] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. NOX: Towards an operating system for networks. *SIGCOMM CCR* (2008).

[44] Hamed, H., Al-Shaer, E., and Marrero, W. Modeling and verification of IPSec and VPN security policies. *International Conference on Network Protocols (ICNP)* (2005).

[45] Hu, X., and Mao, Z. Accurate real-time idenfication of ip prefix hijacking. *IEEE Symposium on Security and Privacy* (2007).

[46] Jacobson, V. Traceroute. `ftp://ftp.ee.lbl.gov/traceroute.tar.gz`.

[47] John, J. P., Katz-Bassett, E., Krishnamurthy, A., Anderson, T., and Venkataramani, A. Consensus routing: The Internet as a distributed system. In *NSDI* (2008).

[48] 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).

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

[50] Khurshid, A., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying network-wide invariants in real time. In *HotSDN* (2012).

[51] Khurshid, A., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying network-wide invariants in real time. *SIGCOMM CCR 42*, 4 (October 2012).

[52] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying network-wide invariants in real time. In *NSDI* (2013).

[53] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying network-wide invariants in real time. In *ONS* (2013).

[54] Lakhina, A., Crovella, M., and Diot, C. Diagnosing network-wide traffic anomalies. In *SIGCOMM* (2004).

[55] Lasserre, M., and Kompella, V. Virtual private lan service (vpls) using label distribution protocol (ldp) signaling. RFC 4762, January 2007.

[56] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis and transformation. *CGO* (2004).

[57] Liu, X., Guo, Z., Wang, X., Chen, F., Lian, X., Tang, J., Wu, M., Kaashoek, M., and Zhang, Z. D3S: Debugging deployed distributed systems. *NSDI* (2008).

[58] Mahajan, R., Wetherall, D., and Anderson, T. Understanding BGP misconfiguration. In *SIGCOMM* (2002).

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

[60] Mandelbaum, Y., Lee, S., and Caldwell, D. Adaptive parsing of router configuration languages. In *Workshop INM* (2008).

[61] Mao, Z. M., Johnson, D., Rexford, J., Wang, J., and Katz, R. Scalable and accurate identification of AS-level forwarding paths. In *Proc. IEEE INFOCOM* (2004).

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

[63] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., and Shenker, S. OpenFlow: Enabling innovation in campus networks. *SIGCOMM CCR* (2008).

[64] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., and Shenker, S. OpenFlow: Enabling innovation in campus networks. *ACM Computer Communication Review* (April 2008).

[65] Nagios. `http://www.nagios.org`.

[66] Natarajan, S., Huang, X., and Wolf, T. Efficient conflict detection in flow-based virtualized networks. In *ICNC* (2012).

[67] Peresini, P., and Canini, M. Is your OpenFlow application correct? In *CoNEXT Student Workshop* (2011).

[68] Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., and Gu, G. A security enforcement kernel for OpenFlow networks. In *HotSDN* (2012).

[69] Quagga Routing Suite. Commercial Resources. `http://www.quagga.net/commercial.php`.

[70] Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., and Walker, D. Abstractions for network update. In *SIGCOMM* (2012).

[71] Renesys. Longer is not always better. `http://www.renesys.com/blog/2009/02/longer-is-not-better.shtml`.

[72] Roscoe, T., Hand, S., Isaacs, R., Mortier, R., and Jardetzky, P. Predicate routing: Enabling controlled networking. *ACM Computer Communication Review* (2003).

[73] Sekar, V., Reiter, M. K., Willinger, W., Zhang, H., Kompella, R. R., and Andersen, D. G. cSAMP: A system for network-wide flow monitoring. In *NSDI* (2008).

[74] 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).

[75] Silveira, F., Diot, C., Taft, N., and Govindan, R. ASTUTE: Detecting a different class of traffic anomalies. In *SIGCOMM* (2010).

[76] Tune, P., and Veitch, D. Towards optimal sampling for flow size estimation. *Internet Measurement Conference (IMC)* (October 2008).

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

[78] Wu, J., Mao, Z. M., Rexford, J., and Wang, J. Finding a needle in a haystack: Pinpointing significant BGP routing changes in an IP network. In *NSDI* (2005).

[79] 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).

[80] Xie, Y., and Aiken, A. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst. 29*, 3 (2007).

[81] Yin, Z., Caesar, M., and Zhou, Y. Towards understanding bugs in open source router software. *SIGCOMM CCR* (June 2010).

[82] Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., and Pasupathy, S. SherLog: Error diagnosis by connecting clues from run-time logs. In *ASPLOS* (2010).

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

[84] Zeng, H., Zhang, S., Ye, F., Jeyakumar, V., Ju, M., Liu, J., McKeown, N., and Vahdat, A. Libra: Divide and conquer to verify forwarding tables in huge networks. *NSDI* (2014).

[85] Zmijewski, E. Reckless driving on the Internet, February 2009. `http://www.renesys.com/blog/2009/02/the-flap-heard-around-the-worl.shtml`.