\bigodot 2010 Haohui Mai

DIAGNOSE NETWORK FAILURES VIA DATA-PLANE ANALYSIS

BY

HAOHUI MAI

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Assistant Professor Samuel T. King

ABSTRACT

Diagnosing problems in networks is a time-consuming and error-prone process. Previous tools to assist operators 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 paper studies an alternate approach: diagnosing problems through static analysis of the data plane. 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. We present Anteater, a tool for checking invariants in the data plane. Anteater translates high-level network invariants into boolean satisfiability problems, checks them against network state using a SAT solver, and reports counterexamples if violations have been found. Applied to a large campus network, Anteater revealed 23 bugs, including forwarding loops and stale ACL rules, with only five false positives. Nine of these faults are being fixed by campus network operators. To my parents, for their love and support.

ACKNOWLEDGMENTS

This thesis is derived from an earlier project, which I work with Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey and Samuel T. King. I would like to thank them for their work devoted in this project.

TABLE OF CONTENTS

LIST OF TABLES
LIST OF FIGURES
CHAPTER 1 INTRODUCTION 1
CHAPTER 2 ARCHITECTURE
CHAPTER 3 ANTEATER DESIGN63.1 Modeling network policies73.2 Calculating reachability83.3 Loop detection, full reachability, and consistent routing93.4 Handling packet transformations103.5 Making Anteater practical12
CHAPTER 4 EXPRESSING INVARIANTS IN SLANG 17
CHAPTER 5 IMPLEMENTATION
CHAPTER 6EVALUATION216.1Bug finding in a deployed network216.2Applicability to router bugs256.3Performance evaluation26
CHAPTER 7 LIMITATIONS
CHAPTER 8 RELATED WORK
CHAPTER 9 CONCLUSIONS 34
APPENDIX A PROOF OF NP-COMPLETENESS
REFERENCES

LIST OF TABLES

3.1	Notation used in Chapter 3	7
6.1	Summary of evaluation results of Anteater on our campus	<u> </u>
	network	2

LIST OF FIGURES

1.1	Overview of the Anteater architecture	2
3.1	An example of a 3-node IP network. a) Network topology and FIBs for routers A , B and C . FIBs are shown in dashed boxes. b) The network graph modeling all policies in a). Ovals represent networking equipment, Rounded rectangles represent special vertices such as destinations, labeled by lower case letters. The lower half of the graphs	
3.2	shows the value of \mathcal{P} for each edge in the graph An algorithm to calculate a boolean formula f representing reachability from s to t for a network graph G . l is a parameter controlling the maximum length of paths that	13
	the user is interested in	1/
3.3	An algorithm to detect forwarding loops involving vertex	14
	v in the network G	14
3.4	An algorithm to check the connectivity invariants for ver-	
05	$\operatorname{tex} v \dots $	14
3.5	An algorithm to check whether the forwarding policy of u and v is consistent	15
3.6	An 3-node IP network augmented from Figure 3.1. The	10
	new policy over (A, B) and (B, C) are shown in dashed boxes.	15
3.7	The execution process of algorithm when determining the formula representing reachability from node A to node C	
	on the graph shown in Figure 3.6	16
4.1	An example of SLang program and its evaluation process. a) The topology of a 3-node network, being checked by invariants of the absence of forwarding loop. b) SLang Pro- gram to implement loop detection algorithm in Figure 3.3. c) Evaluating Rule 3 in Figure 4.1 when running over the network shown in Figure 4.1a). The annotation over the	
	edges shows the bindings introduced during the evaluation	18

6.1	a) The topology of the campus network. Ovals represent	
	routers. Solid lines represent connections. An oval with	
	dashed lines circles the location where a forwarding loop	
	is detected. b) Fragments of data-plane information in the	
	network.	23
6.2	Performance of Anteater when checking three invariants.	
	Run time is measured by wall-clock seconds. All experi-	
	ments are run under 16-way parallelism.	27
6.3	Scalability results of the forwarding loop invariant on UIUC	
	campus network.	28
6.4	Scalability results of the forwarding loop invariant on nine	
	generated AS-level networks	29

CHAPTER 1 INTRODUCTION

Modern enterprise networks are complex, incorporating hundreds or thousands of network devices from multiple vendors performing diverse codependent functions such as routing, switching, and access control across physical and virtual networks (VPNs and VLANs). As in any complex computer system, enterprise networks are prone to a wide range of errors [1, 2, 3, 4, 5, 6, 7, 8], such as misconfigurations, software bugs, or unexpected interactions across protocols. These errors can lead to oscillations, black holes, faulty advertisements, or route leaks that ultimately cause disconnectivity and security vulnerabilities.

Diagnosing problems in networks, however, remains a black art. Operators often rely on heuristics — sending probes, reviewing logs, even observing mailing lists and making phone calls — that slow response to failures.¹ To address this, automated tools for network diagnostics [1, 9] analyze configuration files (i.e. control planes) 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 [6], which have led to multiple high-profile outages and vulnerabilities [4, 10, 11]. 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

¹As one example, a Cisco design technote advises that "Unfortunately, there is no systematic procedure to troubleshoot an STP issue. ... Administrators generally do not have time to look for the cause of the loop and prefer to restore connectivity as soon as possible. The easy way out in this case is to manually disable every port that provides redundancy in the network. ... Each time you disable a port, check to see if you have restored connectivity in the network." [8]



Figure 1.1: Overview of the Anteater architecture.

BGP [1] or firewalls [12, 9]. It's difficult for such diagnosis to reason about interactions that span multiple protocols, as well as dealing with the diversity in configuration languages from different vendors making up typical networks.

We take an alternative approach: diagnosing problems as close as possible to the network's actual behavior through formal analysis of *data-plane states*. Data-plane analysis has two benefits.:

- it opens the opportunities to catch bugs that are invisible at the level of configuration files, by checking the *results* of routing software rather than its inputs.
- It becomes easier to perform unified analysis of a network across protocols and implementations. For example, it's sufficient to model IP forwarding tables, which is relatively simple, in order to model the effects of IP routing protocols such as BGP, OSPF, and RIP.

This paper describes the design and evaluation of Anteater, a tool that performs formal analysis of the data-plane state of network devices. Anteater collects the network topology and the Forwarding Information Bases (FIBs) of network devices, then performs analysis over the resulting network-wide state to check invariants specified in a high-level query language, which we call SLang. We provide a set of common invariants (including reachability and loop detection), but SLang can encode a wide variety of checks. Anteater compiles the query and data-plane state into an instance of a *boolean satisfiability problem* (SAT), and uses a SAT solver [13] to perform analysis. If an invariant is violated, Anteater provides a specific counterexample — such as a packet header, FIB entries, and path that would cause a loop.

Anteater builds on the work of Xie, *et al.* [14], which introduced algorithms to check reachability in IP networks with support for ACL policies. Anteater

differs in that it embodies a complete algorithmic design, and supports arbitrary packet transformations (see discussions in Chapter 3). Moreover, we demonstrate a working implementation and evaluate it on a real network. To the best of our knowledge, Anteater is the first deployed data-plane analysis tool.

We applied Anteater to a large 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 multiple faults in the network, including forwarding loops and stale ACL rules. Several of these faults are being fixed by campus network operators. For example, one detected forwarding loop between a pair of routers was accidentally 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.

The main contributions of this paper are:

- A general, unified framework to systematically detect network problems across multiple protocols via data-plane analysis.
- An algorithm computes network reachability with arbitrary packet transformation, which requires no knowledge of the inverse function of the transforamtion, and is capable of handling for looping paths opposed to previous work [14].
- A proof showing that computing reachability with packet filters is actually NP-complete. It contrasts to the conjecture in [14] saying that the problem can be solved in polynomial time.
- The evaluation with a production network conneting over 70,000 enduser machines and servers.

The rest of the paper organizes as follows. Chapter 3 describes the design of Anteater. Chapter 4 discusses SLang, the high-level declarative language to check network properties. We brief our implementation in Chapter 5, and present our evaluation on Anteater in Chapter 6. Then we discuss the limitation of Anteater in Chapter 7, and the related work in Chapter 8. Finally we conclude in Chapter 9.

CHAPTER 2 ARCHITECTURE

This paper presents the design and implementation of Anteater. Our primary goal is to detect and diagnose a broad, general class of network failures. Anteater detects network problems by applying techniques of formal analysis on the content of forwarding tables contained in routers, switches, firewalls, and other networking equipment (Figure 1.1). 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:

- Absence of forwarding loops. There exists no forwarding loops in the network.
- Connectivity. All computers in the campus network are able to access both the Intranet and the Internet, while respecting network policies and service level agreements (SLAs).
- 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 are the same in general cases.

Not all bugs can be detected with these three invariants. For example, a network operator may want to check if any local computers can access the external network without going through the firewall. To support more sophisticated invariants, as well as the increasingly advanced nature of packet forwarding (with technologies like OpenFlow, programmable NICs, header rewriting, and transformation rules [15, 16, 17]), Anteater should support a large and extensible class of queries.

To support more sophisticated invariants, Anteater allows expression of invariants in SLang, a declarative language we developed that allows operators to tailor checks with network-specific knowledge.

Anteater checks these invariants through several steps. First, Anteater collects the contents of FIBs from networking equipment through vtys (terminals), SNMP, or control sessions maintained to routers [18, 19]. Second, the operator creates new or selects from existing SLang invariants to be checked against the network. Third, Anteater translates both the FIBs and invariants into SAT problems, 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 counter example based on the results to help diagnosis.

The rest of the this paper describes the design and implementation in more detail, including writing invariants, translating the invariants and the network into SAT problems.

CHAPTER 3

ANTEATER DESIGN

We cast network reachability and other network invariants as boolean satisfiability (SAT) problems. In this chapter we discuss our model for network policies, our algorithms for detecting bugs using sets of boolean formulas and a SAT solver, and our approaches for reducing Anteater's runtime.

A SAT problem is a problem that evaluates a set of boolean formulas to determine if there exists at least one variable assignment so that all formulas evaluate to true. If such an assignment exists, then the set of formulas are *satisfiable*, otherwise they are *unsatisfiable*. Although SAT is NP-complete, specialized tools called SAT solvers use heuristics to solve SAT efficiently in some cases [20]. Engineers use SAT solvers in a number of different problem domains, including model checking, hardware verification, and program analysis.

We preface our discussion by noting that Anteater's algorithm for constructing boolean formulas for reachability builds on that of Xie et al. [14]. Our most important contribution is our complete implementation applied to a real network (§6). In addition, Anteater makes two algorithmic contributions:

- The algorithm presented in [14] computed reachability formulas over sets of packets (in $O(|V|^3)$) time, including in networks with packet filters, it did not deal with *evaluating* these formulas, instead stating an assumption that set operations take O(1) time. We show, however, that computing reachability with packet filters is NP-complete (see Appendix). This motivates our design of computing reachability in terms of boolean formulas and testing them with a SAT solver.
- Our approach handles arbitrary packet transformations. It requires no knowledge of the inverse function of the transformation, and it also handles loopy paths compared to the algorithms in [14].

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				
r[v][d][k]	Boolean formula for reachability				
	from node v to node d in exactly				
	k hops				

Table 3.1: Notation used in Chapter 3.

3.1 Modeling network policies

We use the notation shown in Figure 3.1 in the following discussion. 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. \mathcal{P} is a function defined on E to represent general policies.

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 is exactly the same as the ones in 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, like the MAC address, IP address, and port number in packet headers. A packet can flow over an edge if 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 uto node v.

It is easy to see how $\mathcal{P}((u, v))$ can be used to represent a filter. For example, in Figure 3.1 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 $dst_i p =_{24} 10.1.3.0$ in $\mathcal{P}((B, C))$ in Figure 3.1 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, after a trick that we defer to Section 3.4.

3.2 Calculating reachability

This section describes how Anteater calculates reachability, which we will use as a building block for checking higher-level invariants.

We define the reachability problem from s to t as deciding whether there exists a symbolic packet p and an $s \sim t$ path such that p satisfies all constraints \mathcal{P} of the edges along the path. Similar to the algorithm presented in [14], Figure 3.2 shows a dynamic programming algorithm to calculate a boolean formula f representing reachability from s to t for a network graph G, such that f is satisfiable if and only if the reachability problem from s to t has a solution.

In Figure 3.2, r[v][d][k] is the boolean formula which represents the reachability from node v to node d in exactly k hops. The disjunction for calculating r[v][d][i+1] is to evaluate all paths leaving node v.

Depends on the nature of the problem, users might be interested in different numbers of l. For the reachability problem, considering paths containing at most n-1 hops, where n is the number of network devices modeled in G, is sufficient to cover all loop-free paths in G.

Consider how this algorithm works for determining the formula representing reachability from node A to node C in the graph shown in Figure 3.1. It's sufficient to set n equals to 3 as there are only 3 routers in Figure 3.1. When the algorithm starts, Anteater initializes \mathcal{P} as shown in Figure 3.1. Also, Anteater initializes s to node A, t to Node C, n to 3, and r[u][v][0] to true for all nodes u and v in the graph. When the algorithm runs, Anteater sets r as follows:

$$r[A][C][1] = \text{ false}$$

$$r[B][C][1]$$

$$= dst_ip =_{24} 10.1.3.0 \land dst_ip \neq_{25} 10.1.3.128$$

$$r[A][C][2]$$

$$= r[B][C][1] \land \mathcal{P}(A, B)$$

$$= dst_ip =_{24} 10.1.3.0 \land dst_ip \neq_{25} 10.1.3.128 \land$$

$$(dst_ip =_{24} 10.1.2.0 \lor dst_ip =_{24} 10.1.3.0)$$

$$r[B][C][2] = \text{ false}$$

The algorithm returns the formulas $r[A][C][1] \lor r[A][C][2]$ as the result. We will use the formula as a building block for checking higher-level invariants described in the next section.

3.3 Loop detection, full reachability, and consistent routing

Using our basic reachability algorithm as a building block, we show how to check three invariants: absence of forwarding loops, full reachability, and consistency of routing policies between different routers.

Figure 3.3 shows Anteater's algorithm for detecting forwarding loops which involves vertex v in the network. The intuition behind the algorithm is to add a shadow vertex v' that is separate from v but has all of the exact same incoming edges with the same edge policies. That way, v' is reachable from vif and only if the network has a forwarding loop involving v, thus loop(v,G)is satisfiable when there exists a loop involving vertex v. The user can run the algorithm over all vertices in the graph to detect forwarding loops in the network.

Figure 3.4 shows Anteater's algorithm for determining connectivity for vertex v, i.e., whether v is able to access all possible destinations. The idea here is to add a vertex d serving as a sink of all other vertices representing possible destinations, and understanding the reachability between v and d would get some insights on the connectivity for vertex v. In addition, our algorithm also allows operators to specify lists of IP addresses that are intentionally not reachable, but we omitted this from the algorithm we present in Figure 3.4 for simplicity.

Such a process is done through checking satisfiability of $\neg c$ in Figure 3.4. For each vertex u which represents external routers or destinations, the algorithm puts no restrictions on edge (u, d). Hence if a packet can go from vto u, then it must be able to go from v to d. c only evaluates to false when it can't reach d within n hops, where n is the number of network devices in G. i.e., encountering a black holes or forwarding loops. Therefore, finding an assignment such such that $\neg c$ evaluates to true (i.e., testing satisfiability of $\neg c$) would give an example that is unable to reach its destination.

Derived from the algorithm in Figure 3.4, Figure 3.5 shows Anteater's algorithm for checking whether the forwarding policy of two different vertices, u and v, are consistent. More concretely, the set of destinations that u and vcan reach are the same, even though they could reach the same destination through different path. The idea is to test whether their results on connectivity checking are equivalent. This is done by testing satisfiability of $(r_u \text{ xor } r_v)$. The formula evaluates to true when r_u and r_v evaluate to different values, implying an inconsistent forwarding policy. Network engineer could extract the information for further diagnosis.

3.4 Handling packet transformations

The discussion in earlier sections assumed that packets traversing the network remain unchanged. However, numerous protocols employ mechanisms that *transform* packets while they are in flight. For example, MPLS can add or remove 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 section, we give extensions to the algorithms discussed above in order to handle deterministic packet transformations. For brevity, we discuss the extension to the algorithm presented in Figure 3.2 (extensions to the other algorithms follow trivially).

The basic idea is to model a packet transformation as a constraint between the old and new packet. We replace each symbolic packet s with an array (s_i) , where s_i represents the state of the packet at the *i*th hop. Intuitively, a packet transformation $f(\cdot)$ at hop *i* induces the constraint $s_{i+1} = f(s_i)$; a hop at which there is no transformation induces the constraint $s_{i+1} = s_i$.

We describe the algorithm in the context of an simplified example. Figure 3.6 augments the network in Figure 3.1 with two new policies over the edges (A, B) and (B, C): the policy on edge (A, B) toggles t when t equals to 0 or 1, otherwise t stays identical. The policy on edge (B, C) filters all packets except t = 1.

The extension modifies the algorithm as follows. First, the policy function \mathcal{P} now takes two additional parameters, s and s', which are the symbolic packets representing the states of the packets before and after flowing through the edge. We define the net policy function as $\mathcal{P}_{|s,s'}$. We also define s(var) as the variable var in the symbolic packet s. The transformation is expanded into boolean formulas over s and s'. For example, the sub formula in Figure 3.6

$$(s(t) = 1 \land s'(t) = 0) \lor (s(t) = 0 \land s'(t) = 1)$$
$$\lor (s(t) = s'(t) \land s(t) \neq 0 \land s(t) \neq 1)$$

models the transformation over t described above. Anteater adds identity transformations (e.g. $s(dst_ip) = s'(dst_ip)$) for fields that are not involved in the transformation.

Next, the extension uses an array of symbolic packets to model the state of the packet at each hop. The extension needs to initialize n symbolic packets, namely s_0, s_1, \dots, s_{n-1} at the beginning of the algorithm, and changes the line that calculates the boolean formula (Line 5) into the following:

$$r[v][t][i+1] \leftarrow \bigvee_{(v,u) \in E(G)} \left(r[u][t][i] \land \mathcal{P}(v,u)_{|s_i,s_{i+1}} \right)$$

As a reference, we present a run of the resulting algorithm to determine the formula representing reachability from node A to node C in Figure 3.7. For brevity, we simplified the results and only left the constraint on variables in s_2 , which modeled the initial header of the packet exiting A. The result reads only the packet satisfying the constraint $dst_ip = 25$ 10.1.3.0 $\wedge t = 0$ which can go from node A to node C.

3.5 Making Anteater practical

The main challenge to making Anteater practical is solving SAT problems efficiently. Anteater addresses the challenge using two approaches. First, Anteater reduces the number of SAT queries required to check invariants. Second, Anteater exploits parallelism when solving the problems.

Anteater tests the satisfiability by posing a query to the SAT solver. To reduce the number of SAT queries for checking invariants, we optimize the algorithms for the common case – invariant violations are uncommon – to reduce the complexity of checking invariants. To determine whether a vertex is able to reach the rest of the network, the connectivity algorithm poses only one SAT query to check this invariant. If Anteater detects an invariant violation, the network operator can use more detailed, but expensive, queries to gather additional information.

To exploit the increasing parallelism of modern commodity processors, we check invariants in parallel. The first source of parallelism is that each of the invariants we check are independent and can be checked in parallel. The second source of parallelism is that invariants on individual devices are independent, allowing us to calculate invariants for each device in parallel, even for the same invariant. These two sources of parallelism supply ample opportunities to exploit parallelism to speed up processing.



$$\mathcal{P}((B, A)) = dst_{-i}p =_{24} 10.1.1.0
\mathcal{P}((B, b)) = dst_{-i}p =_{24} 10.1.2.0
\mathcal{P}((B, C)) = dst_{-i}p =_{24} 10.1.3.0 \land dst_{-i}p \neq_{25} 10.1.3.12
\mathcal{P}((C, B)) = dst_{-i}p =_{24} 10.1.1.0 \lor dst_{-i}p =_{24} 10.1.2.0
\mathcal{P}((C, c)) = dst_{-i}p =_{24} 10.1.3.0$$

Figure 3.1: An example of a 3-node IP network. a) Network topology and FIBs for routers A, B and C. FIBs are shown in dashed boxes. b) The network graph modeling all policies in a). Ovals represent networking equipment, Rounded rectangles represent special vertices such as destinations, labeled by lower case letters. The lower half of the graphs shows the value of \mathcal{P} for each edge in the graph.

```
\begin{aligned} & \textbf{function reach}(s,t,l,G) \\ & r[u][v][0] \leftarrow \text{true}, \, \forall u, v \in V(G). \\ & \textbf{for } i = 0 \text{ to } n - 2 \text{ do} \\ & \textbf{for all } v \in V(G) \text{ do} \\ & r[v][t][i+1] \leftarrow \bigvee_{(v,u) \in E(G)} (r[u][t][i] \wedge \mathcal{P}(v,u)) \\ & \textbf{end for} \\ & \textbf{end for} \\ & \textbf{return} \quad \bigvee_{1 \leq i \leq n-1} r[s][t][i] \end{aligned}
```

Figure 3.2: An algorithm to calculate a boolean formula f representing reachability from s to t for a network graph G. l is a parameter controlling the maximum length of paths that the user is interested in.

```
function loop(G)

n \leftarrow the number of network devices in G

m \leftarrow an empty hash map

for all v that represents a network device do

v' \leftarrow a new vertex in V(G), m[v] \leftarrow v'

for all (u, v) \in E(G) do

(u, v') \leftarrow a new edge in E(G)

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', n, G).

end for
```

Figure 3.3: An algorithm to detect forwarding loops involving vertex v in the network G.

```
function connectivity(v, G)

n \leftarrow the number of network devices in G

d \leftarrow a new vertex in V(G)

for all u \in V(G) - \{d\} do

if u is a vertex either representing external routers or destinations then

(u, d) \leftarrow a new edge in E(G)

\mathcal{P}((u, d)) \leftarrow true.

end if

end for

c \leftarrow reach(v, d, n, G)

Test satisfiability of \neg c.
```

Figure 3.4: An algorithm to check the connectivity invariants for vertex v.

function consistent_forwarding(u, v, G) $n \leftarrow$ the number of network devices in G $d \leftarrow$ a new vertex in V(G)for all $w \in V(G) - \{d\}$ do if w is a vertex either representing external routers or destinations then $(w, d) \leftarrow$ a new edge in E(G) $\mathcal{P}((w, d)) \leftarrow$ true. end if end for $r_u \leftarrow \operatorname{reach}(u, d, n, G)$ $r_v \leftarrow \operatorname{reach}(v, d, n, G)$ Test satisfiability of $(r_u \operatorname{xor} r_v)$.

Figure 3.5: An algorithm to check whether the forwarding policy of u and v is consistent.



Figure 3.6: An 3-node IP network augmented from Figure 3.1. The new policy over (A, B) and (B, C) are shown in dashed boxes.

$$\begin{split} r[A][C][2] \\ &= r[B][C][1] \land \mathcal{P}(A,B)_{|s_1,s_2} \\ &= s_0(dst_ip) =_{24} 10.1.3.0 \land s_0(t) = 1 \\ \land s_1(dst_ip) \neq_{25} 10.1.3.128 \land s_0(t) = s_1(t) \\ \land s_0(dst_ip) = s_1(dst_ip) \land s_1(dst_ip) = s_2(dst_ip) \\ \land(s_1(dst_ip) =_{24} 10.1.2.0 \lor s_1(dst_ip) =_{24} 10.1.3.0) \\ \land((s_1(t) = 1 \land s_2(t) = 0) \lor (s_1(t) = 0 \land s_2(t) = 1) \\ \lor (s_1(t) = s_2(t) \land s_1(t) \neq 0 \land s_1(t) \neq 1)) \\ &= s_2(dst_ip) =_{25} 10.1.3.0 \land s_2(t) = 0 \\ r[B][C][2] = \text{ false} \end{split}$$

Figure 3.7: The execution process of algorithm when determining the formula representing reachability from node A to node C on the graph shown in Figure 3.6.

CHAPTER 4

EXPRESSING INVARIANTS IN SLANG

Anteater introduces SLang, a declarative, Prolog-like language, to simplify the task of writing invariants in Anteater.

Figure 6.1c) is an example of a SLang program (which implements the loop detection algorithm presented in Figure 3.3). A SLang program is constructed by a list of rules. A rule has the format *head* :- *body*, which reads "*head* is true if *body* evaluates to true". Both head and body contain a list of *atoms*, separated by commas. An atom can take in a number of variables to describe a particular fact. For example, the atom edge(u, v, g, p) in Rule 3 evaluates to true if and only if there is an edge between vertex u and v in graph g, and the policy(i.e., $\mathcal{P}((u, v))$ in Section 3) over the edge is p. We define the list of atoms to evaluate to true if and only if each atom evaluates to true in sequential order. A rule is called a *fact rule* when the head of a rule is omitted; the first two rules in Figure 6.1c) are fact rules. As implied by their names, fact rules are a collection of basic facts.

The user initiates the execution of a SLang program by posting a query. A query has the format ?- body, meaning that the user is interested in the facts described in body. SLang evaluates the query through a depth-first search over all possibilities that body evaluates to true. The SLang engine tries to evaluate the first element in body, if it's not a known fact, it recursively evaluates the atom until all atoms are reduced to facts, resulting in a solution of the SLang program. When the engine finishes the recursions, it reports all solutions to the user.

Figure 4.1b) implements the algorithm shown in Figure 3.3, which works as follows: Rule 1 initializes Anteater, reads the FIB information, and stores the resulting graph into g. Rule 2 implements the hash map m in Figure 3.3. For each vertex v which represents a network device, it adds a corresponding shadow vertex v' into the graph and stores the relationship into the map m. Rule 3 copies all information of incoming edges for v, by saying that for all



Figure 4.1: An example of SLang program and its evaluation process. a) The topology of a 3-node network, being checked by invariants of the absence of forwarding loop. b) SLang Program to implement loop detection algorithm in Figure 3.3. c) Evaluating Rule 3 in Figure 4.1 when running over the network shown in Figure 4.1a). The annotation over the edges shows the bindings introduced during the evaluation.

possible vertex u, if there is an edge from a vertex u to v with the policy p, then add an edge from u to v' with the same policy. The plus operator at the beginning of the rule instructs SLang to evaluate the rules aggressively, i.e., copying the information of incoming edges before to continue. Rule 4 posts a query to the SLang engine checking reachability between v and v' for all possible pairs of v and v' in m. More concretely, it says for all possible pairs of v and v' in m, the result of $\operatorname{reach}(v,v',g)$ (in Figure 3.2) is the boolean formula c, and test satisfiability of c through evaluating the atom $\operatorname{sat}(c)$. For more complex properties, the user can define atoms in C/C++ and invoke them in SLang through C/C++ binding of SLang.

Figure 4.1c) shows how SLang evaluates Rule 3 when running on the network shown in Figure 4.1a). Recall Rule 3 tries to copy the information of incoming edges of v into its shadow vertex v'. We assume v' is shadow vertex of v in the following discussion. The goal of evaluation is to infer new facts about edge(u,v',g,p) (i.e., reducing free variables u,v',g,p into constants) through the rule. At a high level, this is done through a depth-first search over the search space: First, it replaces the head of the rule (edge(u,v',g,p))of the rule with its body, and recursively evaluates the body (Node 1). At each stage of evaluation, it tries to bind an atom with free variables to atoms with constant parameters that evaluate to true. For example, the engine discovers m(A, A') is true in the left sub tree, thus the engine binds v, v' with A, A', then it continues the search from Node 2 with the above bindings. Different ways of unification forms a new sub search tree in the space. The engine goes deep into the search tree until it successfully reduces all variables into constants, which is called a solution. For example, the engine discovers that $edge(B,A',g,\mathcal{P}((B,A)))$ evaluates to true at Node 4, effectively adding a new edge from B to A' with the policy $\mathcal{P}((B,A))$ in graph g.

The evaluation process terminates when the SLang engine cannot discover any new facts. In this example, the process terminates after it discovers all of the following atoms are true: $edge(B,A',g,\mathcal{P}((B,A)))$, $edge(A,B',g,\mathcal{P}((A,B)))$, $edge(C,B',g,\mathcal{P}((C,B)))$, and $edge(B,C',g,\mathcal{P}((B,C)))$.

That way, the rule copies the information of incoming edges of v into its shadow vertex for each v representing a network device.

CHAPTER 5

IMPLEMENTATION

We have implemented Anteater on Linux with about 4,000 lines of codes, along with roughly 400 lines of auxiliary scripts to canonicalize data-plane information from Foundry, Juniper and Cisco routers into comma-separated values format.

Anteater modifies YAP Prolog to compile and execute SLang programs. SLang programs interact with Anteater through C/C++ bindings of YAP Prolog.

Anteater prepares a global graph for checking the invariants: it parses the FIB information for each network device, creates a local graph. Then it translates packet forwarding, filtering and transformation policies into intermediate representations, and merges them back into a global graph.

Anteater uses LLVM Intermediate Representation (IR) [21] as its intermediate representation. Anteater translates network policies into LLVM IR. Anteater implements the policy function \mathcal{P} as a collection of functions in LLVM IR. A SLang program invokes Anteater to generate the IR, after the execution of the program, Anteater uses LLVM to perform a number of standard compiler optimizations, including inlining, const propagation, global sub-expression elimination, and strength reduction at IR level.

After the optimization, Anteater translates the IR into the native format of the Yices [13] SMT solver, then it invokes Yices to solves the SAT problem. We let Yices to output the assignment when the SAT problem is satisfiable. Anteater parses these results and reports them back to the user.

CHAPTER 6 EVALUATION

Our evaluation of Anteater consists of three components. First (§ 6.1), to evaluate Anteater's ability to find bugs in deployed systems, we apply Anteater to the University of Illinois at Urbana-Champaign (UIUC) 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 (§6.2), 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; we note, of course, that a particular bug's effects will not always be triggered in a live deployment.

Third (§6.3), we conduct a performance and scalability evaluation of Anteater. While far from ideal, Anteater is sufficiently fast to check static properties in networks of up to 1,000 nodes.

6.1 Bug finding in a deployed network

To evaluate Anteater, we applied it to the University of Illinois at Urbana-Champaign (UIUC) campus network. We collected the IP forwarding tables and access control rules from 178 routers in the campus. The mean FIB size was 1,627 entries, which were inserted by a combination of BGP, OSPF, and static routing. We also inferred the network topology based on the data.¹

We have implemented all invariants described in chapter 3, and report their evaluation results on our campus network. We ran all experiments on

¹In general, we expect topology to be a separate input. In this case, a network-wide map was not available to us, and it was inferrable from the FIBs.

Invariants	Loops	Reacha- bility	Consistency
New alerts	9	17	2
Being fixed	9	0	0
Stale config.	0	13	1
False pos.	0	4	1

Table 6.1: Summary of evaluation results of Anteater on our campus network.

a machine which has 48 GB of RAM and 2 2.4 GHz 4-core Intel Xeon X5530 CPUs. The machine ran the 64-bit version of CentOS 5. We used the 64-bit version of Yices 1.0.28 [13] to solve SAT queries.

Figure 6.1 reports the number of violations we found with Anteater. The row *New 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 violate both the invariants of forwarding loop and full reachability. We classified these alerts into three categories. First, the row *Being fixed* means the alerts are confirmed as bugs and currently being fixed by our campus network operators. Second, the row *Stale configuration* means that these alerts result from explicit and intentional configuration rules, but rules that are outdated and no longer needed. Our 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.

6.1.1 Forwarding loops

Anteater detects nine potential forwarding loops in the network. One of them is shown in Figure 6.1 highlighted by a dashed circle. The loop involves two routers: router node and router bypass-a. Router bypass-a sets up a static route for 130.126.244.0/22 towards router node. At the same time, Router node has a default route towards router bypass-a.

As shown in the FIBs, only parts of the prefix are going into the forwarding loop. According to longest prefix match rules, packets destined to



Figure 6.1: a) The topology of the campus network. Ovals represent routers. Solid lines represent connections. An oval with dashed lines circles the location where a forwarding loop is detected. b) Fragments of data-plane information in the network.

130.126.244.0/23 from router bypass-a could reach the destination. Packets destined to the prefix 130.126.244.0/22 except 130.126.244.0/23 would fall into the forwarding loop.

Incidentally, all nine loops happen between these two routers. According to the network operator, router bd 3 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, and router bypass-a was introduced to offload the traffic. As a side effect, the forwarding loops were also introduced when the operator was migrating configuration.

These loops have been confirmed by the network operator and they are currently being fixed.

6.1.2 Reachability

Anteater issues 17 unreachability alerts, scattered at routers at different levels of hierarchy: one of them is due to the lack of default routes in the router; three of them are due to blocking traffic towards unused IP spaces, the others 13 alerts are because the network blocks traffic towards end hosts.

We recognized that the first four alerts are legitimate operational practice and classified them as false positives. Further investigation on the other 13 alerts shows that they are stale configuration entries: seven out of 13 are internal-used IP addresses that were used in the previous generation of the network. The other six 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.

6.1.3 Consistency

Based on our conversations with our campus network operators, we know that campus routers in the same level of hierarchy should have identical policies, hence we pick one representative router in the hierarchy and check the consistency between this router and all other routers in the same level of hierarchy. Anteater issues two new alerts: (1) The two core routers have different policies on IP prefix 10.0.3.0/24; (2) Some building routers has different policies on the private IP address ranges 169.254.0.0/16 and 192.168.0.0/16.

Upon investigating the bug we found that one router exposes its webbased management interface through 10.0.3.0/24. The other alert is due to a legacy issue that could be dated back to early 90's: according to the design documents of the campus, 169.254.0.0/16 and 192.168.0.0/16 are intended to be only used within one building. Usually each department has only one building and these IP spaces were used in the whole department. As some departments have 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 to connect them together by creating a virtual LAN for these buildings.

6.2 Applicability to router bugs

In addition to detecting bugs in router configurations, Anteater also detects bugs in the router software itself when these bugs affect the data plane. To test how effective Anteater would be at catching router software bugs, we manually characterized 78 bugs randomly sampled from the Bugzilla repository [22] of Quagga [23], an open source software router (the same set of bugs presented in [6]). These defects cannot be detected by configuration checking tools, since they appear even with entirely correct configurations; but Anteater can detect those bugs which affect the data plane.

We classified the bugs with the goal of determining how many could affect the data plane, and what invariants Anteater would require to catch them. Overall, 67 of the 78 bugs (86%) can produce data plane effects detectable by Anteater. Below, we describe the categories and one representative bug from each. (Note the categories, after the first, are not mutually exclusive.)

No data-plane effects: 11 bugs (14%). For example, in Bug 87 in the Bugzilla database, the interactive terminal hangs when the show ip bgp command is executed and the FIB has a large number of entries.

Detectable with unreachability / loops: 60 bugs (77%) / 46 bugs (59%), respectively. For example, in Bug 122, the Zebra daemon of Quagga fails to install routes received from the bgpd daemon into the kernel during heavy load. But bgpd keeps announcing these routes, and hence the router announces routes that the kernel does not know about. If the kernel did not previously have FIB entries for these particular prefixes, it would act as a black hole. If it did previously have entries, these would be inconsistent with the control plane announcements and thus can result in forwarding loops. Thus, Bug 122 may be detected by either unreachability or loops.

Detectable with other predicates 64 bugs (82%): For example, Bug 416 causes a BGP session to remain active after it has been shut down at the control plane. As a result, the router might have paths to IP prefixes that operators intend to avoid, and packets might follow the path, which violates the desired network policies. We reproduced this bug on a local Quagga testbed and successfully detected it with Anteater, via a query to detect a violation of the operator's intent (that data should not flow across the given link). This type of bug would not be detected by our standard unreachability or loop invariants, yet it has a data plane effect that Anteater could discover

with an appropriate invariant. There are several ways Anteater could detect this kind of bug in a practical deployment. First, if the affected router's reachability was intended to mirror that of another router, it could be detected by a consistency check. Second, operators can code targeted queries in SLang. In addition, an interesting direction for future work would be for router command line tools to use Anteater as a way to assert that the operator's commands actually take effect in the data plane, so that operators would not have to individually code queries.

Note that 25 bugs produced data plane effects due to a process crash or deadlock, and therefore may be fairly easily detectable through simpler tests. 54% of the bugs produced data plane effects detectable by Anteater without a process crash.

We note that a particular bug's effects will not always be triggered in a live deployment. However, the majority of these bugs have the potential to produce effects detectable by Anteater, indicating that our approach may be a promising method to detect software router bugs.

6.3 Performance evaluation

In this section we discuss Anteater's runtime performance when we check for loops, reachability, and consistent routing behavior for the 178 routers in the UIUC campus network.

Figure 6.2 shows the running time of Anteater when checking our invariants on the campus network. The running time of Anteater can be broken into three parts: (a) compiling and executing the invariant checkers to generate IR; (b) optimizing the IR with LLVM and generating SAT formulas; (c) running Yices to resolve the SAT queries.

The characteristics of run-time behavior differ for the three invariants. The invariant of forwarding loop spent most of the running time on solving the SAT problems, yet performing optimization in the other two invariants dominates the running time. The optimizer performs standard compiler optimization to simplify the boolean formulas. It may not recognize all patterns used in the forwarding loop invariants, leaving extra complexity for Yices.



Figure 6.2: Performance of Anteater when checking three invariants. Run time is measured by wall-clock seconds. All experiments are run under 16-way parallelism.

6.3.1 Scalability

Our performance data on the UIUC campus network shows that Anteater is a practical tool for large networks. In this section we show that Anteater is likely to be able to handle larger networks as well. First, we show Anteater's scalability on our campus network by scaling down the network. Second, we run Anteater on large networks by using data from the Rocketfuel [24] project.

Figure 6.3 shows the scalability of Anteater when running the forwarding loop invariant. We scaled down the number of vertices, and honored the hierarchy of the network during the scaling because we always remove routers at the lowest-level of hierarchy first during scaling. We also keep the graph connected to model a realistic network.

Figure 6.4 shows our results for experiments on data from the Rocketfuel [24] project. We generated the topology of nine AS-level networks (1221, 1239, 1755, 2914, 3257, 3967, 4755, 6461, 7018) based on the data, and evaluated the performance of the forwarding loop invariant.

To evaluate how sensitive the invariant is to the complexity of FIB entries, we define L as a parameter to control the complexity of FIBs and to test the invariant with different values of L. Let l(r) to be the maximum number of layers of nested prefixes for router r, and L is the average number over all l



Figure 6.3: Scalability results of the forwarding loop invariant on UIUC campus network.

for all routers.

For the largest network (AS-2914 with 1013 vertices), it takes six hours to check the invariant. These results have a large degree of freedom: they depend on the complexity of network topology and FIB information, and the running time of Yices depends on both heuristics and random number seeds. However, these results do indicate that Anteater might be capable of handling large production networks.



Figure 6.4: Scalability results of the forwarding loop invariant on nine generated AS-level networks.

CHAPTER 7 LIMITATIONS

There are several limitations that restrict applicability of our work. First, Anteater assumes that faults have a visible effect on the contents of forwarding state. This assumption may not always hold. Some hardware failures (e.g., corrupting the contents of the packet during forwarding), and benign configuration issues may not be visible from forwarding state of conventional routers. However, we believe our design can detect an important and common set of faults in networks that affect forwarding plane state.

The current implementation of Anteater does not understand policies at the application level nor latencies in the network. Policies at the application level are a special case of handling multiple layers of protocols in Anteater, when the mapping application and network layer is made available [25]. It is possible to model network latency as a special case of packet transformation by introducing a new variable to record the maximum latency along the paths, and modifying it at each hop. Other work shows that SAT is capable of model the arithmetic operations required by such a transformation [14].

When collecting snapshots of data plane information Anteater could miss some bugs when information is incomplete. It is also possible that Anteater would report problems that are only temporary, such as black holes and transit loops during network convergence. Moreover, it could report false positives when given an inconsistent snapshot. Collecting a consistent snapshot of the whole network could be an application of consistent snapshot algorithms [26], or benefit from other next-generation infrastructures [27]. As Anteater primarily serves as a bug detection tool, our experience shows that the requirement of consistent snapshots could be relaxed, as the time required to take a snapshot is small compared to the average time between changes of the FIBs in our campus network.

Anteater might need to acquire FIBs when network reachability is affected. Fortunately, Anteater can make use of solutions available today, including maintaining separately tunneled networks at the forwarding plane [18, 19] or operating through out-of-band control circuits [28], in order to gather data plane state.

CHAPTER 8 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. [14] introduced algorithms to check reachability in IP networks with support for ACL policies. Our core algorithm builds on [14], but that work assumed constant-time set operations, which we have shown hides an NP-complete problem, and did not fully handle protocols that modify packets; Anteater contributes solutions to these problems (see discussion in §3). Moreover, we demonstrate a working implementation and evaluate it on a real network, showing usefulness of the data-plane analysis approach.

Bush and Griffin [29] gave a formal model of integrity (including connectivity and isolation) of virtual private routed networks, and Hamed et al. [30] 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 [1] and firewalls [12, 31, 9], 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 [32, 6]. Second, configuration analysis will miss non-configuration errors (e.g., errors in router software and inconsistencies between the control plane and data plane [33, 34, 6]).

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 that 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 [35, 36] 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 [37, 25, 38, 39, 40, 41]. 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.

Programming languages for networks. Research on declarative networking [42], language support for building distributed systems [43, 44], and debugging distributed systems [45, 46] have shown the importance of language support during construction and debugging of modern networks. These works inspired us to design and implement SLang, a declarative, Prolog-like language to enables operators to express network properties in a succinct way.

SAT solving in other settings. Work on model checking [47], hardware verification [48] and program analysis [49, 50] often encounter problems that are NP-Complete. In practice, they are often reduced into SAT problems so that SAT solvers can solve them effectively in practice. Some research has been proposed to interpret the unsatisfiability core of the reduced SAT problem for diagnosis and problem localization [51]. These works inspired our approach of using SAT solving to model and analyze data-plane behavior.

CHAPTER 9 CONCLUSIONS

We presented Anteater, an efficient, expressive and extensible framework to diagnose network failures via data-plane analysis. Anteater collects dataplane information from network devices, models forwarding-plane behavior as instances of satisfiability problems, and leverages various formal analysis techniques to systematically analyze the network. To our best knowledge, Anteater is the first framework that is able to formally analyze cross-layer, inter-component properties of a computer network.

We found Anteater to be useful for finding real bugs in large networks. We ran Anteater on the UIUC campus network and found 23 bugs. Anteater completed the analysis quickly and gave us concrete counterexamples that we used to help our network operators repair the campus networks. Although there were five false positives, in all of these cases the network operator was able to determine quickly that these alerts were in fact benign. Overall, our study suggests that debugging networks based on data-plane information and using boolean satisfiability is a practical and effective way to debug today's complex network.

APPENDIX A

PROOF OF NP-COMPLETENESS

As pointed out in [14], the reachability algorithm terminates within $O(V^3)$ operations. According to Proposition 1 below, however, determining the reachability between two vertices in the network is *not* in $O(V^3)$, unless P = NP:

Proposition 1. Determining reachability in a network with packet filters is NP-Complete.

Proof. Given an instance of a 3-SAT problem with n variables x_1, x_2, \dots, x_n and k clauses $C_i = \hat{x_a} \vee \hat{x_b} \vee \hat{x_c}$, in which $\hat{x_j} = x_j$ or $\bar{x_j}$, $\forall 1 \leq i \leq k, 1 \leq a, b, c, j \leq n$, construct a network G as the following:

- 1. A packet p being forwarded in G is a vector of n variables $p = (x_1, x_2, \cdots, x_n)$.
- 2. Add k+1 vertices $v_1, v_2, \cdots, v_{k+1}$ in G.
- 3. Add an edge $e_i = (v_i, v_{i+1}), \forall 1 \le i \le k$.
- 4. Let the policy over edge e_i to be C_i .

With the above construction a packet p can flow through e_i if C_i evaluates to true under assignment p. Therefore, determining the reachability between v_1 and v_{k+1} corresponds to the answer to the 3-SAT problem.

Therefore, the problem is NP-Complete because 3-SAT is NP-Complete.

The key is that calculating the formulas representing the reachability and determining the reachability are two different problems. To calculate general reachability in a network, the use of SAT *does not* increase the complexity. Furthermore, the use of SAT or similar techniques is required since the problem is NP-Complete.

REFERENCES

- [1] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *Proc. USENIX NSDI*, 2005.
- [2] J. Wu, Z. M. Mao, J. Rexford, and J. Wang, "Finding a needle in a haystack: Pinpointing significant BGP routing changes in an IP network," in *Proc. USENIX NSDI*, 2005.
- [3] Renesys, "Longer is not always better," http://www.renesys.com/blog/ 2009/02/longer-is-not-better.shtml.
- [4] J. Duffy, "BGP bug bites Juniper software," Network World, December 2007.
- [5] J. Evers, "Trio of Cisco flaws may threaten networks," *CNET News*, January 2007.
- [6] Z. Yin, M. Caesar, and Y. Zhou, "Towards understanding bugs in open source router software," in ACM Computer Communication Review, June 2010.
- [7] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding BGP misconfiguration," in *Proc. ACM SIGCOMM*, 2002.
- [8] Cisco Systems Inc., "Spanning tree protocol problems and related design considerations," http://www.cisco.com/en/US/tech/tk389/tk621/ technologies_tech_note09186a00800951ac.shtml, August 2005, document ID 10556.
- [9] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra, "FIREMAN: A toolkit for FIREwall Modeling and ANalysis," in *Proc. IEEE S&P*, 2006.
- [10] E. Zmijewski, "Reckless driving on the Internet," http://www.renesys. com/blog/2009/02/the-flap-heard-around-the-worl.shtml, February 2009.
- [11] P. B. Godfrey, "What happened to the Internet on Friday," http://youinfinitesnake.blogspot.com/2010/08/ what-happened-to-internet-on-friday.html, August 2010.

- [12] E. S. Al-Shaer and H. H. Hamed, "Discovery of policy anomalies in distributed firewalls," in *Proc. IEEE INFOCOM*, 2004.
- [13] B. Dutertre and L. de Moura, "The Yices SMT solver," http://yices.csl.sri.com/tool-paper.pdf, SRI International, August 2006.
- [14] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, "On static reachability analysis of IP networks," in *Proc. IEEE INFOCOM*, 2005.
- [15] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, and S. Shenker, "OpenFlow: Enabling innovation in campus networks," ACM Computer Communication Review, April 2008.
- [16] "The all new 2010 Intel Core vPro processor family: Intelligence that adapts to your needs (whitepaper)," 2010, ftp://download.intel.com/ products/vpro/whitepaper/crossclient.pdf.
- [17] "JUNOS: MPLS fast reroute solutions, network operations guide," 2007.
- [18] M. Lasserre and V. Kompella, "Virtual private lan service (VPLS) using label distribution protocol (LDP) signaling," RFC 4762, January 2007.
- [19] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina, "Generic routing encapsulation (GRE)," RFC 2784, March 2000.
- [20] "SAT-Race 2010," http://baldur.iti.uka.de/sat-race-2010.
- [21] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. CGO*, 2004.
- [22] Quagga Routing Suite, "Bugzilla main page," https://bugzilla.quagga. net/.
- [23] Quagga Routing Suite, http://www.quagga.net.
- [24] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with rocketfuel," in *Proc. ACM SIGCOMM*, 2002.
- [25] Cisco Systems Inc., "Cisco IOS netflow," http://www.cisco.com/en/US/ products/ps6601/products_ios_protocol_group_home.html.
- [26] A. D. Kshemkalyani and M. Singhal, Distributed Computing: Principles, Algorithms, and Systems, 1st ed. Cambridge University Press, May 2008. [Online]. Available: http://www.worldcat.org/isbn/0521876346
- [27] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani, "Consensus routing: The Internet as a distributed system," in *Proc. USENIX NSDI*, 2008.

- [28] "Out-of-band management," September 2010, http://en.wikipedia.org/ wiki/Out-of-band_management.
- [29] R. Bush and T. G. Griffin, "Integrity for virtual private routed networks," in *Proc. IEEE INFOCOM*, 2003.
- [30] H. Hamed, E. Al-Shaer, and W. Marrero, "Modeling and verification of IPSec and VPN security policies," in *Proc. ICNP*, 2005.
- [31] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: A novel firewall management toolkit," in *Proc. IEEE S&P*, 1999.
- [32] Y. Mandelbaum, S. Lee, and D. Caldwell, "Adaptive parsing of router configuration languages," in *Workshop INM*, 2008.
- [33] G. Goodell, W. Aiello, T. Griffin, J. Ioannidis, P. McDaniel, and A. Rubin, "Working around BGP: An incremental approach to improving security and accuracy of interdomain routing," in *Proc. NDSS*, 2003.
- [34] Z. M. Mao, D. Johnson, J. Rexford, J. Wang, and R. Katz, "Scalable and accurate identification of AS-level forwarding paths," in *Proc. IEEE INFOCOM*, 2004.
- [35] A. Feldmann, O. Maennel, Z. Mao, A. Berger, and B. Maggs, "Locating Internet routing instabilities," in *Proc. ACM SIGCOMM*, 2004.
- [36] X. Hu and Z. M. Mao, "Accurate real-time identification of IP prefix hijacking," in *Proc. IEEE S&P*, 2007.
- [37] Nagios, http://www.nagios.org.
- [38] D. Mauro and K. Schmidt, *Essential SNMP*. O'Reilly, 2005.
- [39] F. Baccelli, S. Machiraju, D. Veitch, and J. Bolot, "The role of PASTA in network measurement," in *Proc. ACM SIGCOMM*, 2006.
- [40] P. Tune and D. Veitch, "Towards optimal sampling for flow size estimation," in *Proc. IMC*, 2008.
- [41] F. Silveira, C. Diot, N. Taft, and R. Govindan, "ASTUTE: Detecting a different class of traffic anomalies," in *Proc. ACM SIGCOMM*, 2010.
- [42] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays," in *Proc. ACM SOSP*, 2005.
- [43] C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat, "Mace: Language support for building distributed systems," in ACM PLDI, 2007.

- [44] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," ACM Transactions on Computer Systems, vol. 18, pp. 263–297, 2000.
- [45] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang, "D3S: Debugging deployed distributed systems," in *Proc. USENIX NSDI*, 2008.
- [46] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *Proc. USENIX NSDI*, 2006.
- [47] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, 1999.
- [48] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu, "Verifying safety properties of a Power PC microprocessor using symbolic model checking without BDDs," in CAV, 1999.
- [49] Y. Xie and A. Aiken, "Saturn: A scalable framework for error detection using boolean satisfiability," ACM Transactions on Programming Languages and Systems, vol. 29, no. 3, 2007.
- [50] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sher-Log: Error diagnosis by connecting clues from run-time logs," in ASP-LOS, 2010.
- [51] I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri, "Debugging overconstrained declarative models using unsatisfiable cores," in *IEEE/ACM ASE*, 2003.