Debugging the Data Plane with Anteater

Haohui Mai, Ahmed Khurshid, Rachit Agarwal
Matthew Caesar, P. Brighten Godfrey, Samuel T. King

University of Illinois at Urbana-Champaign
{mai4, khurshi1, agarwa16, caesar, pbg, kingst}@illinois.edu

ABSTRACT

Diagnosing problems in networks is a time-consuming and error-prone process. Existing 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 instances of boolean satisfiability problems (SAT), checks them against network state using a SAT solver, and reports counterexamples if violations have been found. Applied to a large university 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.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operation; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Reliability

Keywords

Data Plane Analysis, Network Troubleshooting, Boolean Satisfiability

SIGCOMM'11, August 15–19, 2011, Toronto, Ontario, Canada.

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 [10, 11, 12, 14, 25, 32, 38, 41], such as misconfiguration, 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.

However, diagnosing problems in networks 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 [14, 43] 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 [41] and have led to multiple high-profile outages and vulnerabilities [11, 44]. 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 [14, 15] or firewalls [2, 43]. 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.

We take a different and complementary approach. Instead of diagnosing 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. Data plane analysis has two benefits. First, by checking the results of routing software rather than its inputs, we can catch bugs that are invisible at the level of configuration

[1] As 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." [10]

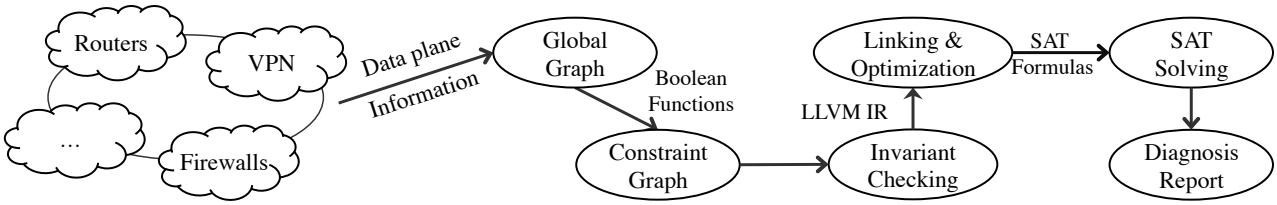


Figure 1: The work flow of Anteater. Clouds are network devices. Ovals are stages in the work flow. Text on the edges shows the type of data flowing between stages.

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.

This paper describes the design, implementation, and evaluation of Anteater, a tool that analyzes 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. Nine of these faults are being fixed by 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 are as follows:

- Anteater is 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 our campus network.
- We show how to express three key invariants as SAT problems, and propose a novel algorithm for handling packet transformations.
- 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. 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 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 are the same.

Anteater checks invariants through several steps. First, Anteater collects the contents of FIBs from networking equipment through vtys (terminals), SNMP, or control sessions maintained to routers [13, 22]. These FIBs may be simple IP longest prefix match rules, or more complex actions like access control lists or modifications of the packet header [1, 21, 28]. 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.

3. ANTEATER DESIGN

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

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: Notation used in Section 3.

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 [8]. Engineers use SAT solvers in a number of different problem domains, including model checking, hardware verification, and program analysis. Please see §7 for more details.

Network reachability can, in the general case, also be NP-complete (see Appendix). We cast network reachability and other network invariants as SAT problems. In this section we 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 [39], 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.

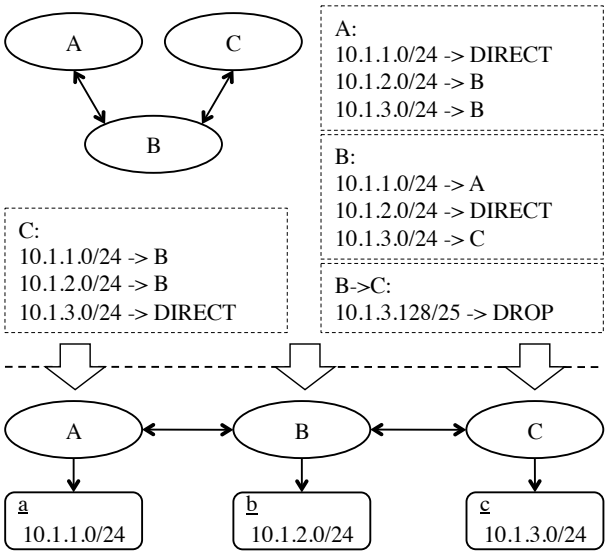
3.1 Modeling network behavior

Figure 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. $\mathcal{P}$ is a function defined on E to represent general policies.

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 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



$$\mathcal{P}(A, a) = dst\_ip =_{24} 10.1.1.0$$
$$\mathcal{P}(A, B) = dst\_ip =_{24} 10.1.2.0 \vee dst\_ip =_{24} 10.1.3.0$$
$$\mathcal{P}(B, A) = dst\_ip =_{24} 10.1.1.0$$
$$\mathcal{P}(B, b) = dst\_ip =_{24} 10.1.2.0$$
$$\mathcal{P}(B, C) = dst\_ip =_{24} 10.1.3.0 \wedge dst\_ip \neq_{25} 10.1.3.128$$
$$\mathcal{P}(C, B) = dst\_ip =_{24} 10.1.1.0 \vee dst\_ip =_{24} 10.1.2.0$$
$$\mathcal{P}(C, c) = dst\_ip =_{24} 10.1.3.0$$

Figure 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.

router v. The sub-formula $dst\_ip =_{24}$ 10.1.3.0 in $\mathcal{P}(B, C)$ in Figure 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 in §3.4.

3.2 Checking reachability

In this subsection, we describe how Anteater checks the most basic invariant: reachability. The next subsection, then, uses 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 4 shows a dynamic programming algorithm to calculate a boolean formula f representing reachability from s to t. The boolean formula 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.

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 $\bigvee_{1 \le i \le k} r[s][i]$

Figure 4: Algorithm to compute a boolean formula representing reachability from s to t in at most k hops in network graph G.

---

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

This part of Anteater is similar to an algorithm proposed by Xie et al. [39], expressed as constraints rather than sets of packets.

To guarantee that all reachability is discovered, one would pick in the worst case $k = n - 1$ 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 $n - 1$.

We give an example run of the algorithm for the network of Figure 3. Suppose we want to check reachability from A to C. Here $k = 2$ suffices since there are only 3 devices. Anteater initializes $\mathcal{P}$ as shown in Figure 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:

$$\begin{aligned} r[A][1] &= \text{false} \\ r[B][1] &= \mathcal{P}(B, C) \\ &= (dst\_ip =_{24} 10.1.3.0 \wedge dst\_ip \neq_{25} 10.1.3.128) \end{aligned}$$

After the second iteration we have:

$$\begin{aligned} r[A][2] &= r[B][1] \wedge \mathcal{P}(A, B) \\ &= dst\_ip =_{24} 10.1.3.0 \ \wedge dst\_ip \neq_{25} 10.1.3.128 \wedge \\ &\quad (dst\_ip =_{24} 10.1.2.0 \vee dst\_ip =_{24} 10.1.3.0) \\ r[B][2] &= \text{false} \end{aligned}$$

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

3.3 Checking forwarding loops, packet loss, and consistency

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

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

Loops. Figure 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 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 which 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.[2] 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 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 v_1 and v_2 will drop the same set of packets. This can be done by running packet_loss to construct two formulas $c_1 =$ packet_loss(v_1, D, G) and $c_2 =$ packet_loss(v_2, D, G), and testing satisfiability of (c_1 xor c_2). 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.

3.4 Packet transformations

The discussion in earlier subsections assumed that packets

[2]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. We omit the details.

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, we 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 $(s_0, \ldots, s_k)$ 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 $s_{i+1} = f(s_i)$. For example, an edge traversed by two MPLS label switched paths with incoming labels ℓ_1^{in}, ℓ_2^{in} and corresponding outgoing labels $\ell_1^{out}, \ell_2^{out}$ would have the transformation constraint

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

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:

$$s_{i+1}.source\_ip =_{24} 12.34.56.0$$

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

$$s_{i+1} = s_i.$$

We let $\mathcal{T}_i(v, w)$ refer to the transformation constraints for packets arriving at v after i hops and continuing to w.

Application to invariant algorithms. Implementing this technique in our earlier reachability algorithm involves two principal changes. First, we must include the transformation constraints $\mathcal{T}$ in addition to the policy constraints $\mathcal{P}$. Second, the edge policy function $\mathcal{P}(u, v)$, rather than referring to variables in a single symbolic packet s, will be applied to various entries of the symbolic packet array (s_i). So it is parameterized with the relevant entry index, which we write as $\mathcal{P}_i(u, v)$; and when computing reachability we must check the appropriate positions of the array. Incorporating those changes, Line 5 of our reachability algorithm (Fig. 4) becomes

$$r[v][i] \leftarrow \bigvee_{(v,u) \in E(G)} \left(\mathcal{T}_{i-1}(v, u) \wedge \mathcal{P}_{i-1}(v, u) \wedge r[u][i-1] \right).$$

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 couldn't 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 $u \to \ell$ 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 $w \to \ell$ with no constraint, for each destination node $w \notin D$. We can now check for packet loss starting at v by testing satisfiability of the formula $\texttt{reach}(v, \ell, n-1, G)$ 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.

Notes. 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 $k = n - 1$ hops does not guarantee that all reachability or loops will be discovered. In the transformation-free case, $k = n - 1$ was sufficient because after $n - 1$ 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 doesn't 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 $k \leq n - 1$ to be sufficient in practice.

Application to other invariants. Packet transformations enable us to express certain other invariants succinctly. Figure 7 shows a simplified version of a real-world example from our 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:

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

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.

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

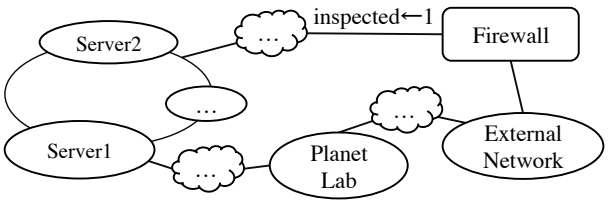


Figure 7: An example where packet transformations allow convenient checking of firewall policy. Solid lines are network links; text on the links represents a transformation constraint to express the invariant. Clouds represent omitted components in the network.

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 [23]. 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 $\mathcal{P}$ and the transformation constraints $\mathcal{T}$ into LLVM functions, whose arguments are the symbolic packets they are constraining. Then Anteater runs the desired invariant algorithm (reachability, loop detection, etc.; §3), outputting the formula using calls to the $\mathcal{P}$ and $\mathcal{T}$ functions. The resulting formula is stored in the `@main` function. Next, LLVM links together the $\mathcal{P}$, $\mathcal{T}$, 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 in §3.2. We want to check reachability from A to C in Figure 3. Anteater translates the policy function $\mathcal{P}(B, C)$ into function `@p_bc()`, and puts the result of dynamic programming algorithm into `@main()`:

```
define @p_bc(%s_i, %s_i+1) {
%0 = load %s_i.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

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

The function takes two parameters `%`s_i and `%`s_{i+1} to support packet transformations as described in §3.4.

The `@main` function is shown at the right side of the snippet. `@p_ab` is the LLVM function representing $\mathcal{P}(A, B)$. `@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 [34]. 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. The details of SLang are beyond the scope of this paper.

5. EVALUATION

Our evaluation of Anteater has three parts. First (§5.1), 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 (§5.2), 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.

Third (§5.3), we conduct a performance and scalability evaluation of Anteater. While far from 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 WorkStation T5500 machine 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 [8]. All experiments were conducted under 16-way parallelism.

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 8: Summary of evaluation results of Anteater on our campus network.

5.1 Bugs found in a deployed network

We applied Anteater to our 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 of §3, and report their evaluation results on our campus network. Figure 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 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. 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.

5.1.1 Forwarding loops

Anteater detected nine potential forwarding loops in the network. One of them is shown in Figure 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 nine loops happened 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, 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.

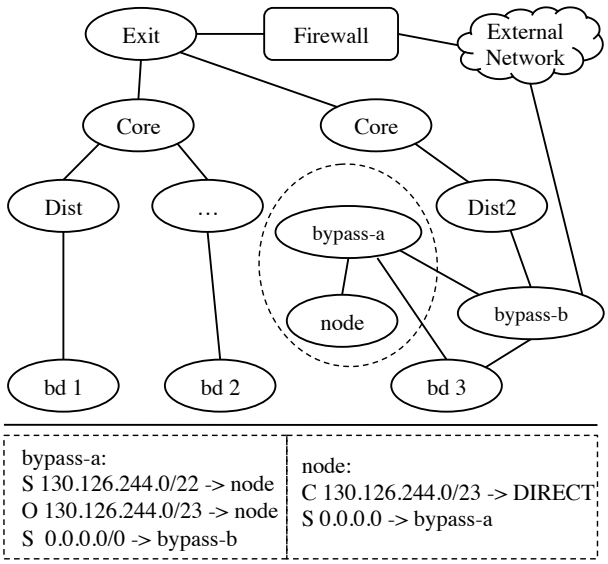


Figure 9: Top: Part of the topology of the campus network. Ovals and solid lines are routers and links respectively. The oval with dashed lines circles the location where a forwarding loop was detected. Bottom: Fragments of data plane information in the network. S stands for static, O stands for OSPF, and C stands for connected.

These loops are reachable from 64 of 178 routers in the network. All loops have been confirmed by the network operator and they are currently being fixed.

5.1.2 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: seven out of 13 are internal 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.

5.1.3 Consistency

Based on conversations with our 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.

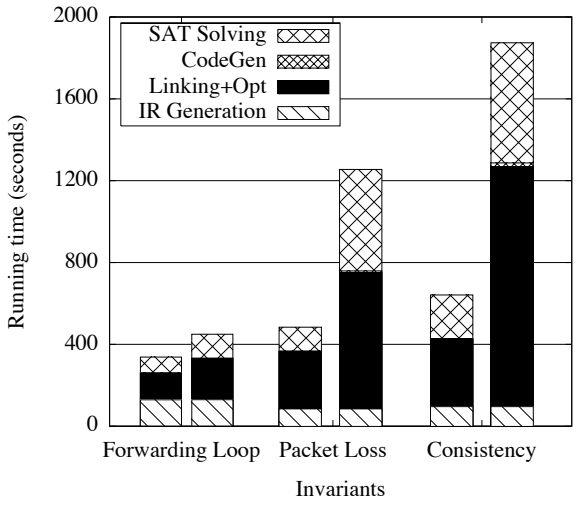


Figure 10: Performance of Anteater when checking three invariants. Time is measured by wall-clock seconds. The left and the right column represent the time of the first run and the total running time for each invariant.

5.2 Applicability to router bugs

Like configuration errors, defects in router software might affect the network. These defects tend to be out of the scope of configuration analysis, but Anteater might be able to detect the subset of such defects which manifest themselves in the data plane.

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 [30]. Quagga is an open-source software router which is used in both research and production [31]. We studied the same set of bugs presented in [41]. 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.

Detectable with `packet_loss` *and* `loop`. 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 either `packet_loss` or `loop`.

Detectable with other invariants. 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 416). Therefore, packets would continue to follow the path in the data plane, violating the operator's intent. This bug cannot be detected by either `packet_loss` or `loop`, but it is possible to detect it via a customized query: checking that there is no data flow across the given link. We reproduced this bug on a local Quagga testbed and successfully detected it with Anteater.

No visible data plane effects. 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.

5.3 Performance and scalability

5.3.1 Performance on the campus network

Figure 10 shows the total running time of Anteater when checking invariants on the campus network. We 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 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.

5.3.2 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

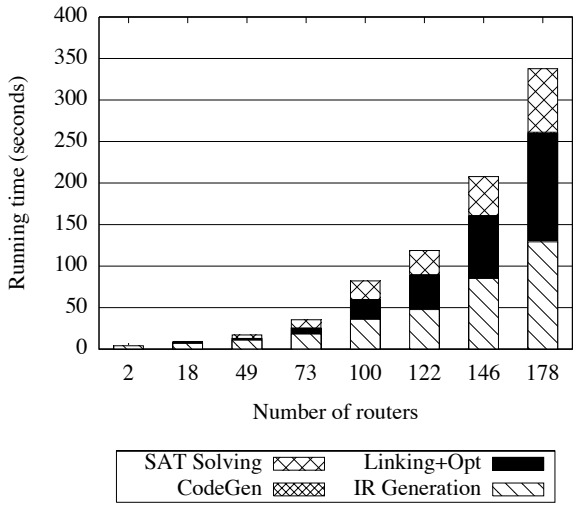


Figure 11: Scalability results of the loop-free forwarding invariant on different subsets of the campus network. The parameter k was set to $n-1$ for each instance.

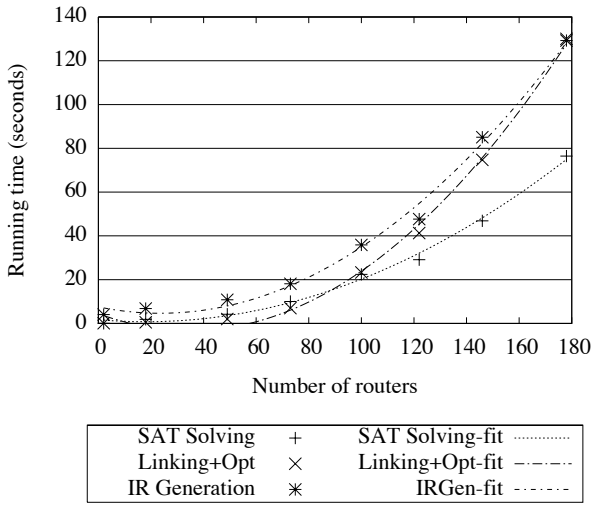


Figure 12: Scatter plots for individual components of the data of Figure 11. Solid lines are quadratic curves fitted for each category of data points.

until a desired number of nodes remain. Figure 11 presents the time spent on the first run when running the forwarding loop invariant on different subsets of the campus network.

Figure 12 breaks down the running time for IR generation, linking and optimization, and SAT solving. We omit the time of code generation since we found that it is negligible. Figure 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 six AS networks (ASes 1221, 1755, 3257, 3967, 4755, 6461) based on topologies from the Rocketfuel project [36], 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" 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 two prefixes, we might have $p_1 = 10.1.0.0/16$ and $p_2 = 10.1.1.0/24$. Figure 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 our 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 the FIBs of both R and R' accordingly for the private subnet. Finally, we added NAT rules as described in §3.4 on the links between R' and R.

Figure 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 our 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.

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).

There are several ways to deal with this problem. First, one could use a consistent snapshot algorithm [17, 24]. Second, if the network uses a software-defined networking approach [28], 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

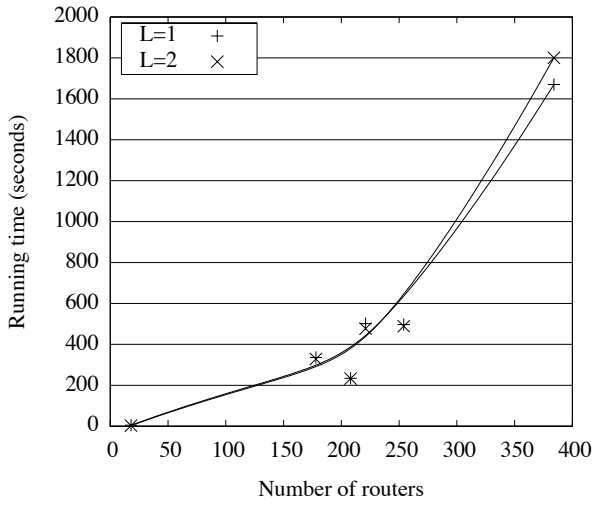


Figure 13: Scalability results of the loop-free forwarding invariant on six AS networks from [36]. L is the parameter to control the complexity of FIBs. Dots show the running time of the invariant for each network. Solid lines are fitted curves generated from the dots.

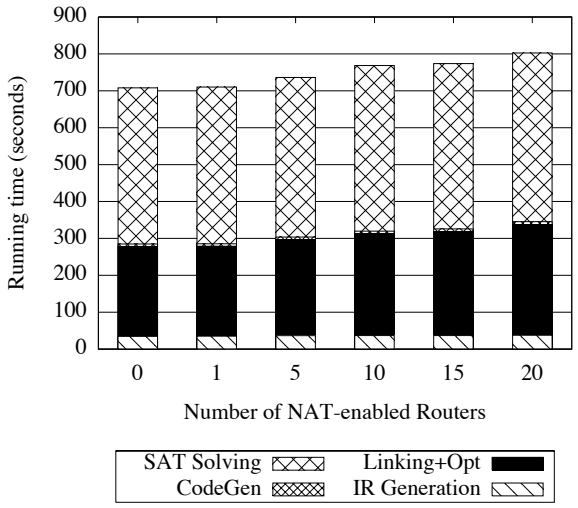


Figure 14: Running time of the loop-free forwarding invariant as a function of the number of routers that have NAT rules.

the average time between changes of the FIBs in our 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.

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 [22, 13] or operating through out-of-band control circuits [3], 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 (see Appendix) regardless of whether the information is represented at the control or data plane. It's 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.

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. [39] 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 (see the Appendix) 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. [33] proposed predicate routing to unify the notions of both routing and firewalling into boolean expressions, Bush and Griffin [9] gave a formal model of integrity (including connectivity and isolation) of virtual private routed networks, and Hamed et al. [19] 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 [6, 14] and firewalls [2, 5, 43], 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 [26, 41]. Second, configuration analysis will miss non-configuration errors (e.g., errors in router software and inconsistencies between the control plane and data plane [18, 27, 41]; see our study of such errors in §5.2).

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 [16, 20] 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 [4, 29, 35, 37]. 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 [7, 40, 42] 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.

8. CONCLUSION

We presented Anteater, a practical system for finding bugs in networks via data plane analysis. Anteater collects data plane information from network devices, models data plane behavior as instances of satisfiability problems, and uses formal analysis techniques to systematically analyze the network. To the best of our knowledge, Anteater is the first design and implementation of a data plane analysis system used to find real bugs in real networks.

We ran Anteater on our campus network and uncovered 23 bugs. Anteater helped our network operators improve the reliability of the campus network. Our study suggests that analyzing data plane information could be a feasible approach to assist debugging today's networks.

Acknowledgements

We would like to thank our shepherd, Emin Gün Sirer, and the anonymous reviewers for their valuable comments. We also thank our network operator Debbie Fligor for collecting data and sharing her operational experience. This research was funded by NSF grants CNS 0834738, CNS 0831212, CNS 1040396, and CNS 1053781, grant N0014-09-1-0743 from the Office of Naval Research, AFOSR MURI grant FA9550-09-01-0539, a grant from the Internet Services Research Center (ISRC) of Microsoft Research, and a Fulbright S&T Fellowship.

9. REFERENCES

[1] JUNOS: MPLS fast reroute solutions, network operations guide. 2007.

[2] E. S. Al-Shaer and H. H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proc. IEEE INFOCOM*, 2004.

[3] Apple. What is lights out management?, September 2010. `http://support.apple.com/kb/TA24506`.

[4] F. Baccelli, S. Machiraju, D. Veitch, and J. Bolot. The role of PASTA in network measurement. In *Proc. ACM SIGCOMM*, 2006.

[5] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. In *Proc. IEEE S&P*, 1999.

[6] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Proc. USENIX NSDI*, 2009.

[7] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS*, 1999.

[8] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Proc. TACAS*, 2009.

[9] R. Bush and T. G. Griffin. Integrity for virtual private routed networks. In *Proc. IEEE INFOCOM*, 2003.

[10] 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.

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

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

[13] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. Generic Routing Encapsulation (GRE). RFC 2784, March 2000.

[14] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proc. USENIX NSDI*, 2005.

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

[16] A. Feldmann, O. Maennel, Z. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *Proc. ACM SIGCOMM*, 2004.

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

[18] 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.

[19] H. Hamed, E. Al-Shaer, and W. Marrero. Modeling and verification of IPSec and VPN security policies. In *Proc. ICNP*, 2005.

[20] X. Hu and Z. M. Mao. Accurate real-time identification of IP prefix hijacking. In *Proc. IEEE S&P*, 2007.

[21] Intel. The all new 2010 Intel Core vPro processor family: Intelligence that adapts to your needs, 2010. http://www.intel.com/Assets/PDF/whitepaper/311710.pdf.

[22] M. Lasserre and V. Kompella. Virtual private LAN service (VPLS) using label distribution protocol (LDP) signaling. RFC 4762, January 2007.

[23] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO*, 2004.

[24] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^{3}S: Debugging deployed distributed systems. In *Proc. USENIX NSDI*, 2008.

[25] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proc. ACM SIGCOMM*, 2002.

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

[27] 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.

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

[29] Nagios. http://www.nagios.org.

[30] Quagga Routing Suite. http://www.quagga.net.

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

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

[33] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Jardetzky. Predicate routing: Enabling controlled networking. *ACM CCR*, January 2003.

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

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

[36] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with rocketfuel. In *Proc. ACM SIGCOMM*, 2002.

[37] P. Tune and D. Veitch. Towards optimal sampling for flow size estimation. In *Proc. IMC*, 2008.

[38] 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.

[39] 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.

[40] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *Proc. ACM TOPLAS*, 29(3), 2007.

[41] Z. Yin, M. Caesar, and Y. Zhou. Towards understanding bugs in open source router software. *ACM CCR*, June 2010.

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

[43] 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.

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

Appendix

In this appendix, 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. [39], 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 [39], 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 [39], it was assumed that evaluating the formula (via set operations in the formulation of [39]) 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 $\mathcal{Q}(e, p)$ 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 $\mathcal{Q}(e, p) = true$ 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 $\mathcal{Q}$ 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 $s \rightarrow t$, 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 $\mathcal{Q}$ 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 $x_1, \ldots, x_n$ and k clauses $C_1, \ldots, C_k$. 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 $v_0, v_1, \ldots, v_k$, 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 $v_{i-1} \rightarrow v_i$. If the first literal in clause C_i is some variable x_i, then the policy function $\mathcal{Q}(e_{i1}, p) = true$ if and only if the ith bit of p is 1; otherwise the first literal in C_i is the negated variable $\overline{x}_i$, and we let $\mathcal{Q}(e_{i1}, p) = true$ 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_{i-1} 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. □