Significant Scientific Work and Immediate Research Plans

The Internet, composed of a vast array of software and protocols operating over hundreds of millions of potentially-misbehaving hosts and thousands of competing ISPs, stands as one of the most complex and large-scale systems ever built by humankind. Unfortunately, like any complex system, it has proven very difficult to get its design right, leading to a vast array of failure modes, high-profile software errors, outages, misconfigurations, vulnerabilities and attacks. Fixing these problems becomes ever more critical as the Internet becomes ever more intertwined with the functioning of our modern society.

To address this challenge, I work on building architectures for ultra-reliable networks: networks that provide strong properties on the ability to detect, isolate, and automatically recover from errors in their constituent software systems. Unlike other domains that engineer for extreme resilience, such as aircraft and medical device design, formal methods are not yet used in networks, possibly because they do not address the unique challenges present in these domains. To do this, my work “bridges the gap” between more theoretical work on formal methods, Byzantine fault tolerance, and graph theory to the unique systems challenges in large-scale networks. In particular, my work follows three key dimensions:

Real-time network verification: Existing tools that check configuration files and network state operate offline at timescales of hours, and cannot detect or prevent errors as they arise. To address this, my goal is to check network-wide invariants in real time, as the network state evolves. Such a system would have a transformative effect on the availability and security of networks, enabling assurance that certain correctness properties (e.g., absence of loops, blackholes, violations of security policies) can be continually retained. To achieve this, my work constructs a formal representation of the network using equivalence-class graphs, a new kind of formal model tailored to the representation of forwarding behavior in networks for the purposes of high-speed verification. My work is able to perform correctness checking in less than a millisecond per network update, multiple orders of magnitude faster than previous approaches. This ongoing work appeared in ACM SIGCOMM, the top conference in computer networking.

Network virtualization: Recent advances in virtual machine technologies have revolutionized software development, but are limited to single-machine environments. To address this, I have created a network-layer substrate that extends the idea behind virtual machines to create entire virtual networks. My design consists of a collection of distributed algorithms to control and manipulate the execution of distributed devices while providing several novel properties not supported by existing networks. First, current networks execute nondeterministically, which significantly increases complexity for programmers. My design provides a collection of distributed algorithms that enables deterministic execution of distributed software, a problem that lies at the heart of programmability issues in networks. This substrate also allows manipulation of time (the ability to rewind, pause and fast-forward executions) and shape (the ability to clone and reposition components) of the distributed code that makes up networks, to assist the operator in debugging the code, and to “look into the future” to foresee possible faulty behavior in the future. Finally, my design enables multiple diverse instances of network components to run in parallel, to avoid faulty behavior at run time. In particular, my design executes multiple diverse instances (different code bases, memory layouts, thread orderings, update timings, etc.) within each virtual network, and uses voting to “drive” the network’s operation (e.g., to determine the output to publish to the forwarding table or to advertise to neighbors). This research was recognized with an NSF CAREER award and a substantial award from the DARPA MRC program. A major network equipment vendor is prototyping our work, targeting inclusion into their mainline router operating system release.

Self-configuring systems: Today’s ISPs hire armies of engineers to manually configure routers and debug problems, and in daily life we are surrounded by an ever-increasing array of complex embedded devices that require substantial configuration to interoperate. Forcing humans to configure and manage networks increases reaction time to faults, introduces the potential for misconfiguration, and substantially increases operating costs. What is lacking today is a principled look at how to make systems manage themselves. Toward this goal I have developed a class of “plug-and-play” protocols that bootstrap, configure, and manage themselves with minimal manual intervention. In particular, I designed RCP (a self-configuring iBGP replacement for ISP networks), SEATTLE (a self-configuring Ethernet replacement for data center networks), and Virtual Ring Routing (a self-configuring routing protocol for ad-hoc wireless networks), which resulted in a publication in NSDI (the top conference in computer systems), and two ACM SIGCOMM publica-
tions. As part of my seven-year collaboration with AT&T Labs, my work on the RCP has been deployed and is in daily use in their tier-1 ISP network.
Debugging the Data Plane with Anteater

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

This article was published in ACM SIGCOMM 2011 (14% acceptance rate). SIGCOMM is the flagship conference on computer networking.

More recent followon work to this article, extending the algorithm to perform real-time verification, was published in ACM Hot Topics in Software Defined Networking (HotSDN), August 2012.

I led this work (the author names appearing before mine are students).
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

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. 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGCOMM’11, August 15–19, 2011, Toronto, Ontario, Canada.
Copyright 2011 ACM 978-1-4503-0797-0/11/08 ...$10.00.

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]
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 vty (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
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 \( G = (V, E, P) \), where \( V \) is the set of networking devices and possible destinations, \( E \) is the set of directed edges representing connections between vertices. \( P \) 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 \( \text{var} = \text{width prefix} \) to simplify our discussion. This notation is a convenient way of writing a boolean formula saying that the first \( \text{width} \) bits of the variable \( \text{var} \) are the same as those of \( \text{prefix} \). For example, \( \text{dst}_ip =_{24} 10.1.3.0 \) is a boolean formula testing the equality between the first 24 bits of \( \text{dst}_ip \) and 10.1.3.0. The notion \( \text{var} \neq \text{width prefix} \) is the negation of \( \text{var} = \text{width prefix} \).

For each edge \((u, v)\), we define \( P(u, v) \) 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. \( P(u, v) \) is the conjunction (logical and) over all policies’ constraints on symbolic packets from node \( u \) to node \( v \).

\[ P(u, v) \] 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, \( P(B, C) \) has \( \text{dst}_ip \neq_{25} 10.1.3.128 \) as a part of it. Forwarding is represented as a constraint as well: \( P(u, v) \) will be constrained to include only those symbolic packets that router \( u \) would forward to router \( v \). The sub-formula \( \text{dst}_ip =_{24} 10.1.3.0 \) in \( P(B, C) \) 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 path that can be forwarded from \( s \) to \( t \). More formally, the problem is to decide if there exists a symbolic packet \( p \) and an \( s \leadsto t \) path such that \( p \) satisfies all constraints \( P \) 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
            \( r[v][i] \leftarrow \bigvee_{(v, u) \in E(G)} (P(v, u) \land r[u][i - 1]) \)
        end for
    end for
    return \( \bigvee_{1 \leq i \leq k} r[s][i] \)

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
        \( E(G) \leftarrow E(G) \cup \{ (u, v') \} \)
        \( P(u, v') \leftarrow P(u, v) \)
    end for
    Test satisfiability of \( \neg \)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 \( P \) 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:

\[
\begin{align*}
    r[A][1] & = false \\
r[B][1] & = P(B, C) \\
        & = (dist_{ip} =_{24} 10.1.3.0 \land dist_{ip} \neq_{25} 10.1.3.128)
\end{align*}
\]

After the second iteration we have:

\[
\begin{align*}
    r[A][2] & = r[B][1] \land P(A, B) \\
    & = dist_{ip} =_{24} 10.1.3.0 \land dist_{ip} \neq_{25} 10.1.3.128 \land \\
    & \quad (dist_{ip} =_{24} 10.1.2.0 \lor dist_{ip} =_{24} 10.1.3.0) \\
    r[B][2] & = false
\end{align*}
\]

The algorithm then returns the formula \( r[A][1] \lor r[A][2] \).

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 \( E(G) \)
        \( P(u, d) \leftarrow true \)
    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 \( \text{packet_loss} \) to construct two formulas \( c_1 = \text{packet_loss}(v_1, D, G) \) and \( c_2 = \text{packet_loss}(v_2, D, G) \), and testing satisfiability of \( c_1 \oplus c_2 \). 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 preprending 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 $i$th 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 $\ell_i^{in}$, $\ell_j^{in}$ and corresponding outgoing labels $\ell_i^{out}$, $\ell_j^{out}$ would have the transformation constraint

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

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

$$s_{i+1}, \text{source}_ip = 12.34.56.78$$

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:

$$s_{i+1}, \text{source}_ip :\geq 24 12.34.56.0$$

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

$$s_{i+1} = s_i.$$ We let $\mathcal{T}(v, w)$ 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

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

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 $\ell$ acting as a sink for lost packets. For each non-destination node $u$, we add an edge $u \rightarrow \ell$ 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 $\ell \rightarrow w$ with no constraint, for each destination node $w \not\in D$. We can now check for packet loss starting at $v$ by testing satisfiability of the formula $\text{reach}(v, \ell, n - 1, G)$ 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 $\ell$); 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 $\text{inspected}$ 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:

$$\mathcal{T}_i(f, v) = s_{i+1}, \text{inspected} \leftarrow 1.$$ Then for each internal server $S$, we check whether

$$(s_k, \text{inspected} = 0) \land R(\text{ext}, S)$$

where $\text{ext}$ is the node representing the external network, and $R(S, \text{ext})$ is the boolean formula representing reachability from $\text{ext}$ 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 \( P \) and the transformation constraints \( T \) into LLVM functions, whose arguments are the symbolic packets they are constraining. Then Anteater runs the desired invariant algorithm (reachability, loop detection, etc.; §3), outputting the formula using calls to the \( P \) and \( T \) functions. The resulting formula is stored in the @main function. Next, LLVM links together the \( P \), \( T \), and @main functions and optimizes when necessary. The result is translated into SAT formulas, which are passed into a SAT solver. Finally, Anteater invokes the SAT solver and reports the results to the operator.

Recall the example presented in §3.2. We want to check reachability from \( A \) to \( C \) in Figure 3. Anteater translates the policy function \( P(B,C) \) into function @p_bc(), and puts the result of dynamic programming algorithm into @main():

\[
\begin{align*}
define \text{void} @\text{main}() & \{ \\
\text{\%0} & = \text{load} @\text{pkt}.\text{dst}_\text{ip} : \text{formula} \\
\text{\%1} & = \text{and} \text{\%, \%0} \\text{fff00} & (\text{let} \{ t1 \text{ (bvand p0 \%fff00g)}}) \\
\text{\%2} & = \text{icmp eq} \%1, \text{\%0}010300 & (\text{let} \{ t2 \text{ (= t1 \text{0}010300)}}) \\
\text{\%3} & = \text{and} \text{\%, \%0}0fff80 & (\text{let} \{ t3 \text{ (bvand p0 \%fff80)}}) \\
\text{\%4} & = \text{icmp ne} \%3, \text{\%0}010380 & (\text{let} \{ t4 \text{ (not (= t3 \text{0}010380)}}) \\
\text{\%5} & = \text{and} \text{\%, \%4} & (\text{let} \{ t5 \text{ (and t3 t4)}}) \\
\text{\%6} & = \text{and} \text{\%, \%0}fff000 & (\text{let} \{ t6 \text{ (bvand p0 \%fff000)}}) \\
\text{\%7} & = \text{icmp eq} \%6, \text{\%0}010200 & (\text{let} \{ t7 \text{ (not (= t6 \text{0}010200)}}) \\
\text{\%8} & = \text{and} \text{\%, \%7} & (\text{let} \{ t8 \text{ (and t3 t7)}}) \\
\text{call void} @\text{assert}(@\text{t8}) & (\text{(?t8)))}) \\
\text{ret void} & \\
\end{align*}
\]

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 @\text{pkt}.\text{dst}_\text{ip} is renamed to \%0. 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 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-Log [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 Booletor 1.4.1 with PicoSAT 936 and PrecoSAT 570 [8]. All experiments were conducted under 16-way parallelism.


<table>
<thead>
<tr>
<th>Invariants</th>
<th>Loops</th>
<th>Packet loss</th>
<th>Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alerts</td>
<td>9</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>Being fixed</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Stale config.</td>
<td>0</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>False pos.</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>No. of runs</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

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.

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.

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 loopo $S$ 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
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 Rock-etfuel 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 a 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
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, Anteater 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 tunnelled 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 +, −, ≤ 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
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.

8. CONCLUSION

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 N00014-09-1-0743 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


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 matching forward 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 $Q(x, p)$ 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 $s \rightarrow t$ path in $G$, such that $Q(x, p) = true$ 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 $Q$ 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 $Q$ 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 let $s = v_0$ and $t = v_k$. For each clause $C_i$, we add the 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 $Q(e_{i1}, p) = true$ if and only if the 1st bit of $p$ is 1; otherwise the first literal in $C_i$ is the negated variable $\overline{x}_i$, and we let $Q(e_{i1}, p) = true$ if and only if the 1st 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 $s$ to $t$ 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.
Virtually Eliminating Router Bugs
Eric Keller, Minlan Yu, Matthew Caesar, Jennifer Rexford

This article was published in ACM CoNEXT (17% acceptance rate). CoNEXT is widely considered as one of the top three conferences in computer networking.

This work is now being prototyped by a major network equipment vendor, targeting release as part of their mainline router operating system.

I led this work (the author names appearing before mine are students).
Virtually Eliminating Router Bugs

Eric Keller*  Minlan Yu*  Matthew Caesar†  Jennifer Rexford*
* Princeton University, Princeton, NJ, USA  † UIUC, Urbana, IL, USA
ekeller@princeton.edu  {minlan,yu,jrex}@cs.princeton.edu  caesar@cs.uiuc.edu

ABSTRACT
Software bugs in routers lead to network outages, security vulnerabilities, and other unexpected behavior. Rather than simply crashing the router, bugs can violate protocol semantics, rendering traditional failure detection and recovery techniques ineffective. Handling router bugs is an increasingly important problem as new applications demand higher availability, and networks become better at dealing with traditional failures. In this paper, we tailor software and data diversity (SDD) to the unique properties of routing protocols, so as to avoid buggy behavior at run time. Our bug-tolerant router executes multiple diverse instances of routing software, and uses voting to determine the output to publish to the forwarding table, or to advertise to neighbors. We design and implement a router hypervisor that makes this parallelism transparent to other routers, handles fault detection and booting of new router instances, and performs voting in parallelism transparent to other routers, handles fault detection and booting of new router instances, and performs voting in the presence of routing-protocol dynamics, without needing to modify software of the diverse instances. Experiments with BGP message traces and open-source software running on our Linux-based router hypervisor demonstrate that our solution scales to large networks and efficiently masks buggy behavior.

Categories and Subject Descriptors
C.2.6 [Computer-Communication Networks]: Internetworking—Routers; C.4 [Performance of Systems]: [Fault tolerance, Reliability, availability and serviceability]

General Terms
Design, Reliability

Keywords
Routers, Bugs, Reliability, BGP

1. INTRODUCTION
The Internet is an extremely large and complicated distributed system. Selecting routes involves computations across millions of routers spread over vast distances, multiple routing protocols, and highly customizable routing policies. Most of the complexity in Internet routing exists in protocols implemented as software running on routers. These routers typically run an operating system, and a collection of protocol daemons which implement the various tasks associated with protocol operation. Like any complex software, routing software is prone to implementation errors, or bugs.

1.1 Challenges in dealing with router bugs
The fact that bugs can produce incorrect and unpredictable behavior, coupled with the mission-critical nature of Internet routers, can produce disastrous results. This can be seen from the recent spate of high-profile vulnerabilities, outages, and huge spikes in global routing instability [40, 39, 16, 22, 21, 13, 31]. Making matters worse, ISPs often run the same protocols and use equipment from the same vendor worldwide, increasing the probability that a bug causes simultaneous failures or a network-wide crash. While automated systems can prevent misconfigurations from occurring [23, 24], these techniques do not work for router bugs, and in fact the state-of-the-art solution today for dealing with router bugs involves heavy manual labor—testing, debugging, and fixing code. Unfortunately operators must wait for vendors to implement and release a patch for the bug, or find an intermediate work around on their own, leaving their networks vulnerable in the meantime.

Worse still, bugs are often discovered only after they cause serious outages. While there has been work on dealing with failures in networks [35, 33, 27], router bugs differ from traditional “fail-stop” failures (failures that cause the router to halt in some easily-detectable way) in that they violate the semantics of protocol operation. Hence a router can keep running, but behave incorrectly—by advertising incorrect information in routing updates, or by distributing the wrong forwarding-table entries to the data plane, which can trigger persistent loops, oscillations, packet loss, session failure, as well as new kinds of anomalies that can’t happen in correctly behaving protocols. This fact, coupled with the high complexity and distributed nature of Internet routing, makes router bugs notoriously difficult to detect, localize, and contain.

As networks become better at dealing with traditional failures, and as systems that automate configuration become more widely deployed, we expect bugs to become a major...
roadblock in improving network availability. While we acknowledge the long-standing debate in the software engineering community on whether it is possible to completely prevent software errors, we believe unforeseen interactions across protocols, the potential to misinterpret RFCs, the increasing functionality of Internet routing, and the ossification of legacy code and protocols will make router bugs a “fact-of-life” for the foreseeable future and we proceed under that assumption.

1.2 The case for diverse replication in routers

Unlike fail-stop failures, router bugs can cause Byzantine faults, i.e., they cause routers to not only behave incorrectly, but violate protocol specification. Hence, we are forced to take a somewhat heavy-handed approach in dealing with them (yet as we will find, one that appears to be necessary, and one that our results indicate is practical). In particular, our design uses a simple replication-based approach: instead of running one instance of routing software, our design uses a router hypervisor to run multiple virtual instances of routing software in parallel. The instances are made diverse to decrease the likelihood they all simultaneously fail due to a bug. We leverage data diversity (to manipulate the inputs to the router, for example by jittering arrival time of updates, or changing the layout of the executable in memory) and software diversity (given multiple implementations of routing protocols already exist, running several of them in parallel). We then rely on Byzantine-fault tolerant (BFT) techniques to select the “correct” route to send to the forwarding table (FIB), or advertise to a neighbor.

The use of BFT combined with diverse replication (running multiple diverse instances) has proven to be a great success in the context of traditional software, for example in terms of building robust operating systems and runtime environments [18, 28, 36, 44, 12]. These techniques are widely used since heterogeneous replicas are unlikely to share the same set of bugs [18, 28, 44]. In this paper, we adapt diverse replication to build router software that is tolerant of bugs.

A common objection of this approach is performance overheads, as running multiple replicas requires more processing capacity. However, BFT-based techniques provide a simple (and low-cost) way to leverage the increasingly parallel nature of multicore router processors to improve availability without requiring changes to router code. Network operators also commonly run separate hardware instances for resilience, across multiple network paths (e.g., multihoming), or multiple routers (e.g., VRRP [27]). Some vendors also protect against fail-stop failures by running a hot-standby redundant control plane either on multiple blades within a single router or even on a single processor with the use of virtual machines [19], in which case little or no additional router resources are required. Since router workloads have long periods with low load [9], redundant copies may be run during idle cycles. Recent breakthroughs vastly reduce computational overhead [45] and memory usage [26], by skipping redundancy across instances.

1.3 Designing a Bug-Tolerant Router

In this paper, we describe how to eliminate router bugs “virtually” (with use of virtualization technologies). We design a bug-tolerant router (BTR), which masks buggy behavior, and avoids letting it affect correctness of the network layer, by applying software and data diversity to routing. Doing so, however, presents new challenges that are not present in traditional software. For example, (i) wide-area routing protocols undergo a rich array of dynamics, and hence we develop BFT-based techniques that react quickly to buggy behavior without over-reacting to transient inconsistencies arising from routing convergence, and (ii) our design must interoperate with existing routers, and not require extra configuration efforts from operators, and hence we develop a router hypervisor that masks parallelism and churn (e.g., killing a faulty instance and bootstrapping a new instance).

At the same time we leverage new opportunities made available by the nature of routing to build custom solutions and extend techniques previously developed for traditional software. For example, (i) routers are typically built in a modular fashion with well-defined interfaces, allowing us to adapt BFT with relatively low complexity, and implement it in the hypervisor with just a few hundred lines of code, (ii) using mechanisms that change transient behavior without changing steady-state outcomes are acceptable in routing, which we leverage to achieve diversity across instances, and (iii) routing has limited dependence on past history, as the effects of a bad FIB update or BGP message can be undone simply by overwriting the FIB or announcing a new route, which we leverage to speed reaction by selecting a route early, when only a subset of instances have responded, and updating the route as more instances finish computing. Moreover, router outputs are independent of the precise ordering and timing of updates, which simplifies recovery and bootstrapping new instances.

The next section discusses how diversity can be achieved and how effective it is, followed by a description of our design (Section 3) and implementation (Section 4). We then give performance results in Section 5, consider possible deployment scenarios in Section 6, contrast with related work in Section 7, and conclude in Section 8.

2. SOFTWARE AND DATA DIVERSITY IN ROUTERS

The ability to achieve diverse instances is essential for our bug-tolerant router architecture. Additionally, for performance reasons, it is important that the number of instances that need to be run concurrently is minimal. Fortunately, the nature of routing and the current state of routing software lead to a situation where we are able to achieve enough diversity and that it is effective enough that only a small number of instances are needed (e.g., 3-5, as discussed below). In this section we discuss the various types of diversity mechanisms, in what deployment scenario they are likely to be used, and how effective they can be in avoiding bugs.

Unfortunately, directly evaluating the benefits of diversity across large numbers of bugs is extremely challenging, as it requires substantial manual labor to reproduce bugs. Hence, to gain some rough insights, we studied the bug re-
ports from the XORP and Quagga Bugzilla databases [8, 5], and taxonomized each into what type of diversity would likely avoid the bug and experimented with a small subset, some of which are described in Table 1.3

2.1 Diversity in the software environment

Code base diversity: The most effective, and commonly thought of, type of diversity is where the routing software comes from different code bases. While often dismissed as being impractical because a company would never deploy multiple teams to develop the same software, we argue that diverse software bases are already available and that router vendors do not need to start from scratch and deploy multiple teams.

First, consider that there are already several open-source router software packages available (e.g., XORP, Quagga, BIRD). Their availability has spawned the formation of a new type of router vendor based on building a router around open-source software [7, 8].

Additionally, the traditional (closed-source) vendors can make use of open-source software, something they have done in the past (e.g., Cisco IOS is based on BSD Unix), and hence may run existing open-source software as a “fallback” in case their main routing code crashes or begins behaving improperly. Router vendors that do not wish to use open-source software have other alternatives for code diversity, for example, router vendors commonly maintain code acquired from the purchase of other companies [38].

As a final possibility, consider that ISPs often deploy routers from multiple vendors. While it is possible to run our bug-tolerant router across physical instances, it is most practical to run in a single, virtualized, device. Even without access to the source code, this is still a possibility with the use of publicly available router emulators [1, 3]. This way, network operators can run commercial code along with our hypervisor directly on routers or server infrastructure without direct support from vendors. While intellectual property restrictions arising from their intense competition makes vendors reticent to share source code with one another, this also makes it likely that different code bases from different vendors are unlikely to share code (and hence unlikely to share bugs).

We base our claim that this is the most effective approach partially from previous results which found that software implementations written by different programmers are unlikely to share the vast majority of implementation errors in code [30]. This result can be clearly seen in two popular open-source router software packages: Quagga and XORP differ in terms of update processing (timer-driven vs. event-driven), programming language (C vs. C++), and configuration language, leading to different sorts of bugs, which are triggered on differing inputs. As such, code-base diversity is very effective and requires only three instances to be run concurrently.

However, effectively evaluating this is challenging, as bug reports typically do not contain information about whether inputs triggering the bug would cause other code bases to fail. Hence we only performed a simple sanity-check: we selected 9 bugs from the XORP Bugzilla database, determined the router inputs which triggered the bug, verified that the bug occurred in the appropriate branch of XORP code, and then replayed the same inputs to Quagga to see if it would simultaneously fail. We then repeated this process to see if Quagga’s bugs existed in XORP. In this small check, we did not find any cases where a bug in one code base existed in the other, mirroring the previous findings.

Version diversity: Another source of diversity lies in the different versions of the same router software itself. One main reason for releasing a new version of software is to fix bugs. Unfortunately, operators are hesitant to upgrade to the latest version until it has been well tested, as it is unknown whether their particular configuration, which has worked so far (possibly by chance), will work in the latest version. This hesitation comes with good reason, as often times when fixing bugs or adding features, new bugs are introduced into code that was previously working (i.e., not just in new features). This can be seen in some of the example bugs described in Table 1. With our bug-tolerant router, we can capitalize on this diversity.

For router vendors that fully rely on open-source software, version diversity will add little over the effectiveness of code-base diversity (assuming they use routers from three code bases). Instead, version diversity makes the most sense for router vendors that do not fully utilize code-base diversity. In this case, running the old version in parallel is protection against any newly introduced bugs, while still being able to take advantage of the bug fixes that were applied.

Evaluating this is also a challenge as bug reports rarely contain the necessary information. Because of this, to evaluate the fraction of bugs shared across versions (and thus, the effectiveness), we ran static analysis tools (splint, uno, and its4) over several versions of Quagga, and investigated overlap across versions. For each tool, we ran it against each of the earlier versions, and then manually checked to see how many bugs appear in both the earlier version as well as the most recent version. We found that overlap decreases quickly, with 30% of newly-introduced bugs in 0.99.9 avoided by using 0.99.1, and only 25% of bugs shared across the two versions. As it is not 100% effective, this will most likely be used in combination with other forms of diversity (e.g., diversity in the execution environment, described next).

2.2 Execution environment diversity

Data diversity through manipulation of the execution environment has been shown to automatically recover from a wide variety of faults [12]. In addition, routing software specific techniques exist, two of which are discussed below. As closed-source vendors do not get the full benefit from running from multiple code bases, they will need to rely on data diversity, most likely as a complement to version diversity.

This comes from the result of our study which showed version diversity to be 75% effective, so we assume that two versions will be run, each with two or three instances of that version (each diversified in terms of execution environment, which as we discuss below can be fairly effective).

Update timing diversity: Router code is heavily concurrent, with multiple threads of execution and multiple processes on a single router, as well as multiple routers simultaneously running, and hence it is not surprising that this creates the potential for concurrency problems. Luckily, we
Table 1: Example bugs and the diversity that can be used to avoid them. Note for the bug listed as Quagga XX, it was reported on the mailing list titled “quick route flap gets mistaken for duplicate, route is then ignored,” but never filed in Bugzilla.

<table>
<thead>
<tr>
<th>Bug</th>
<th>Description</th>
<th>Effective Diversity</th>
</tr>
</thead>
<tbody>
<tr>
<td>XORP 814</td>
<td>The asynchronous event handler did not fairly allocate its resources when processing events from the various file descriptors. Because of this, a single peer sending a long burst of updates could cause other sessions to time out due to missed keepalives.</td>
<td>Version (worked in 1.5, but not 1.6)</td>
</tr>
<tr>
<td>Quagga 370</td>
<td>The BGP default-originate command in the configuration file does not work properly, preventing some policies from being correctly realized.</td>
<td>Version (worked in 0.99.5, but not 0.99.7)</td>
</tr>
<tr>
<td>XORP 814</td>
<td>(See above)</td>
<td>Update (randomly delay delivery)</td>
</tr>
<tr>
<td>Quagga XX (see note)</td>
<td>A race condition exists such that when a prefix that is withdrawn and immediately re-advertised, the router only propagates to peers the withdraw message, and not the subsequent advertisement.</td>
<td>Update (randomly delay delivery)</td>
</tr>
<tr>
<td>XORP 31</td>
<td>A peer that initiates a TCP connection and then immediately disconnects causes the BGP process to stop listening for incoming connections.</td>
<td>Connection (can delay disconnect)</td>
</tr>
<tr>
<td>Quagga 418</td>
<td>Static routes that have an unreachable next hop are correctly considered inactive. However, the route remains inactive even when the address of network device is changed to something that would make the next hop reachable (e.g., a next hop of 10.0.0.1 and an device address that changed from 9.0.0.2/24 to 10.0.0.2/24).</td>
<td>Connection (can interpret change as reset as well)</td>
</tr>
</tbody>
</table>

can take advantage of the asynchronous nature of the routing system to increase diversity, for example, by introducing delays to alter the timing/ordering of routing updates received at different instances without affecting the correctness of the router (preserving any ordering required by the dependencies created by the protocol, e.g., announcements for the same prefix from a given peer router must be kept in order, but announcements from different peer routers can be processed in any order). We were able to avoid two of the example bugs described in Table 1 with a simple tool to introduce a randomized short delay (1-10ms) when delivering messages to the given instance. Further, by manually examining the bug databases, we found that approximately 30% of bugs could be avoided by manipulating the timing/ordering of routing updates.

**Connection diversity:** Many bugs are triggered by changes to the router’s network interfaces and routing sessions with neighbors. From this, we can see that another source of diversity involves manipulating the timing/order of events that occur from changes in the state or properties of the links/interfaces or routing session. As our architecture (discussed in Section 3) introduces a layer between the router software and the sessions to the peer routers, we can modify the timing and ordering of connection arrivals or status changes in network interfaces. For the two example bugs in Table 1, we found they could be avoided by simple forms of connection diversity, by randomly delaying and restarting connections for certain instances. By manually examining the bug database, we found that approximately 12% of bugs could be avoided with this type of diversity.

### 2.3 Protocol diversity

As network operators have the power to perform configuration modifications, something the router vendors have limited ability to do, there are additional forms of diversity that they can make use of. Here, we discuss one in particular. The process of routing can be accomplished by a variety of different techniques, leading to multiple different routing protocols and algorithms, including IS-IS, OSPF, RIP, etc. While these implementations differ in terms of the precise mechanisms they use to compute routes, they all perform a functionally-equivalent procedure of determining a FIB that can be used to forward packets along a shortest path to a destination. Hence router vendors may run multiple different routing protocols in parallel, voting on their outputs as they reach the FIB. To get some rough sense of this approach, we manually checked bugs in the Quagga and XORP Bugzilla databases to determine the fraction that resided in code that was shared between protocols (e.g., the zebra daemon in Quagga), or code that was protocol independent. From our analysis, we estimate that at least 60% of bugs could be avoided by switching to a different protocol.

### 3. BUG TOLERANT ROUTER (BTR)

Our design works by running multiple diverse router instances in parallel. To do this, we need some way of allowing multiple router software instances to simultaneously execute on the same router hardware. This problem has been widely studied in the context of operating systems, through the use of virtual machine (VM) technologies, which provide isolation and arbitrate sharing of the underlying physical machine resources. However, our design must deal with two new key challenges: (i) replication should be transparent and hidden from network operators and neighboring routers (Section 3.1), and (ii) reaching consensus must handle the transient behavior of routing protocols, yet must happen quickly enough to avoid slowing reaction to failures (Section 3.2).

#### 3.1 Making replication transparent

First, our design should hide replication from neighboring routers. This is necessary to ensure deployability (to maintain sessions with legacy routers), efficiency (to avoid requiring multiple sessions and streams of updates between
peers), and ease of maintenance (to avoid the need for operators to perform additional configuration work). To achieve this, our design consists of a router hypervisor, as shown in Figure 1. The router hypervisor performs four key functions:

![Diagram of architecture](image)

**Figure 1: Architecture of a bug-tolerant router.**

**Sharing network state amongst replicas:** Traditional routing software receives routing updates from neighbors, and uses information contained within those updates to select and compute paths to destinations. In our design, multiple instances of router software run in parallel, and somehow all these multiple router instances need to learn about routes advertised by neighbors. To compute routes, each internal instance needs to be aware of routing information received on peering sessions. However, this must happen without having instances directly maintain sessions with neighboring routers. To achieve this, we use a replicator component, which acts as a replica coordinator to send a copy of all received data on the session to each router instance within the system. Note that there may be multiple sessions with a given peer router (e.g., in the case of protocol diversity), in which case the replicator sends received data to the appropriate subset of instances (e.g., those running the same protocol). The replicator does not need to parse update messages, as it simply forwards all data it receives at the transport layer to each instance.

**Advertising a single route per prefix:** To protect against buggy results, which may allow the router to keep running but may cause it to output an incorrect route, we should select the majority result when deciding what information to publish to the FIB, or to advertise to neighbors. To do this, we run a voter module that monitors advertisements from the router instances, and determines the route the router should use (e.g., the majority result). Our design contains two instances of the voter: an update voter that determines which routing updates should be sent to neighbors, and a FIB voter that determines which updates should be sent to the router’s FIB (forwarding table). As with the replicator, the update voter may vote among a subset of instances, for example, those belonging to the same protocol. The FIB voter will vote among all instances, as all instances must come to the same decisions with regard to the FIB.

**Maintaining a set of running replicas:** BFT-based techniques rely on having a sufficient number of correctly-behaving replicas in order to achieve consensus. Hence, if an instance crashes or begins producing buggy output, we may wish to replace it with a new copy. To achieve this, our hypervisor is responsible for bootstrapping the new instance when it begins running. For traditional routers, bootstrapping involves establishing a session with a neighboring router, which causes the neighboring router to send out update messages for each of the prefixes it has an entry for in its RIB. To avoid introducing externally visible churn, the hypervisor keeps a history of the last update peers have sent for each prefix, and replays this for any new instance upon startup of that instance.

**Presenting a common configuration interface:** As there is no standardization of the configuration interface in routers, each router has ended up with its own interface. In the case where instances from different code bases are used, to keep the network operator from needing to configure each instance separately, a mechanism is needed to hide the differences in each configuration interface. Fortunately, this is not unlike today’s situation where ISPs use routers from multiple vendors. To cope with this, ISPs often run configuration management tools which automate the process of targeting each interface with a common one. As such, we can rely on these same techniques to hide the configuration differences.

**3.2 Dealing with the transient and real-time nature of routers**

The voter’s job is to arbitrate amongst the “outputs” (modifications to the FIB, outbound updates sent to neighbors) of individual router instances. This is more complex than simply selecting the majority result—during convergence, the different instances may temporarily have different outputs without violating correctness. At the same time, routers must react quickly enough to avoid slowing convergence. Here, we investigate several alternative voting strategies to address this problem, along with their tradeoffs.

**Handling transience with wait-for-consensus:** The extreme size of the Internet, coupled with the fact that routing events are propagated globally and individual events trigger multiple routing updates, results in very high update rates at routers. With the use of replication, this problem is potentially worsened, as different instances may respond at different times, and during convergence they may temporarily (and legitimately) produce different outputs. To deal with this, we use wait-for-consensus voting, in which the voter waits for all instances to compute their results before determining the majority vote. Because all non-buggy routers output the same correct result in steady-state, this approach can guarantee that if $k$ or fewer instances are faulty with at least $2k + 1$ instances running, no buggy result will reach the FIB or be propagated to a peer.

Note that in practice, waiting for consensus may also re-

---

4Since voting also reveals the set of misbehaving instances, our approach also simplifies diagnosis, as the hypervisor can explicitly report the set of buggy outputs it observes.
duce instability, as it has an effect similar to the MRAI (Minimum Route Advertisement Interval) timer (routers with MRAI send updates to their neighbors only when a timer expires, which eliminate multiple updates to a prefix that occur between timer expiries). Namely, forcing the voter to wait for all instances to agree eliminates the need to advertise changes that happen multiple times while it is waiting (e.g., in the presence of unstable prefixes). However, the downside of this is that reaction to events may be slowed in some cases, as the voter must wait for the \( k + 1 \) slowest instance to finish computing the result before making a decision.

**Speeding reaction time with master/slave:** Routers must react quickly to failures (including non-buggy events) to ensure fast convergence and avoid outages. At the same time, the effects of a bad FIB update or BGP message can be undone simply by overwriting the FIB or announcing a new route. To speed reaction time, we hence consider an approach where we allow outputs to temporarily be faulty. Here, we mark one instance as the master, and the other instances as slaves. The voter operates by always outputting the master’s result. The slaves’ results are used to cross-check against the master after the update is sent or during idle cycles. The benefit of this approach is that it speeds convergence to the running time of the master’s computation. In addition, convergence is no worse than the convergence of the master, and hence at most one routing update is sent for each received update. However, the downside of this approach is that if the master becomes buggy, we may temporarily output an incorrect route. To address this, when failing over to a slave, the voter readvertises any differences between the slaves’ routing tables and the routing table computed by the master. Hence, temporarily outputting an incorrect route may not be a problem, as it only leads to a transient problem that is fixed when the slaves overthrow the master.

Finally, we consider a hybrid scheme which we refer to as continuous-majority. This approach is similar to wait-for-consensus in that the majority result is selected to be used for advertisement or for population into the FIB. However, it is also similar to master/slave in that it does not wait for all instances to compute results before selecting the result. Instead, every time an instance sends an update, the voter reruns its voting procedure, and updates are only sent when the majority result changes. The benefit of this approach is that it may speed reaction to failure, and the majority result may be reached before the slowest instance finishes computing. The downside of this approach is that convergence may be worsened, as the majority result may change several times for a single advertised update. Another downside of this approach is that voting needs to be performed more often, though, as we show in our experiments (Section 5) this overhead is negligible under typical workloads.

### 4. ROUTER HYPERVISOR PROTOTYPE

Our implementation had three key design goals: (i) not requiring modifications to routing software, (ii) being able to automatically detect and recover from faults, and (iii) low complexity, to not be a source of new bugs. Most of our design is agnostic to the particular routing protocol being used. For locations where protocol-specific logic was needed, we were able to treat messages mostly as opaque strings. This section describes our implementation, which consists of a set of extensions built on top of Linux. Our implementation was tested with XORP versions 1.5 and 1.6, Quagga versions 0.98.6 and 0.99.10, and BIRD version 1.0.14. We focused our efforts on supporting BGP, due to its complexity and propensity for bugs. Section 4.1 describes how we provide a wrapper around the routing software, in order for unmodified routing software to be used, and Section 4.2 describes the various faults that can occur and how our prototype detects and recovers from them.

#### 4.1 Wrapping the routing software

To eliminate the need to modify existing router software, our hypervisor acts as a wrapper to hide from the routing software the fact that it is a part of a bug-tolerant router, and allows the routing instances to share resources such as ports, and access to the FIB. Our design (Figure 2) takes advantage of the fact that sockets are used for communicating with peer routers, and for communicating forwarding table (FIB) updates to the kernel. Hence, our implementation intercepts socket calls from the router instances using the LD_PRELOAD environment variable and uses a modified libc library, called hv-libc, to redirect messages to a user-space module, called virtd, which manages all communication.

![Figure 2: Implementation architecture.](http://example.com/figure2.png)

The two key functions the hypervisor then needs to manage are discussed below:

**Socket-based communications:** To connect to peer routers (with TCP) and for writing to the common FIB (with Netlink), the multiple routers need to share access to a common identifier space (e.g., port 179 in BGP). We handle this by intercepting socket system calls in hv-libc, performing address translation in hv-libc, and using virtd as a proxy (e.g., when a router instance listens on port 179, instead they are made to listen on a random port and virtd will listen on 179 and connect to each of the random ports when receiving an incoming connection).

**Bootstrapping new connections:** When the BTR initially starts up, the routing instances start with empty routing tables. In BGP, a session with a peer is established by creating a TCP connection, exchanging OPEN messages, and acknowledging the OPEN message with a KEEPALIVE message. After the session is established, the peers exchange routing information. However, when replacing a failed instance, we need to bootstrap it locally, to prevent the failure from being externally visible (e.g., sending a route-refresh to a peer). Additionally, we need to bootstrap it independently, to prevent the new instance starting in a faulty state (e.g., bootstrapping off another router instance). Since a router’s state only depends on the last received RIB advertised by its neighbors, we add some additional logic to the hypervisor...
to store the last-received update for each (prefix, neighbor) pair. Then when a new instance is started, the hypervisor replays its stored updates. To lower complexity, the hypervisor treats the (prefix, neighbor) fields and other attributes in the packets as opaque strings, and does not implement protocol logic such as route selection.

4.2 Detecting and recovering from faults

To deal with bugs, our hypervisor must detect which outputs are buggy (e.g., with voting), and recover from the buggy output (by advertising the voting result, and if necessary restarting/replacing the buggy instance).

Detection: One of our main goals is that the BTR should be able to automatically detect and recover from bugs affecting correctness of the router’s control or data planes.\(^5\) Since our design fundamentally relies on detecting differences in outputs of different instances, we need to handle every possible way their outputs could differ. All faults can be generalized to four categories: (i) an instance sending a message when it should not, (ii) an instance not sending a message when it should, (iii) an instance sending a message with incorrect contents, and (iv) bugs that cause a detectable faulty system event, such as process crashing or socket error. The first three categories are detected by using voting (the fourth category is easily detectable, so no further discussion is given). If an instance has a different output from the majority, we consider it a fault. For example, in case (i) above, the winning update will be the NULL update, in cases (ii) and (iii) the winning update will be the most-commonly advertised one. To avoid reacting to transient changes, voting is only performed across steady-state instance outputs, which have been stable for a threshold period of time. We then mark instances whose steady-state outputs differ from those of the majority or those that are not yet stable as being faulty (including in schemes like master/slave, which perform this step after advertising).\(^6\)

Recovery: In the common case, recovering from a buggy router simply involves using the output from the voting procedure. However, to deal with cases where the router is persistently buggy, or crashes, we need some way to kill and restart the router. As a heuristic, we modified our hypervisor with a fault threshold timeout. If an instance continues to produce buggy output for longer than the threshold, or if the router undergoes a faulty system event, the router is killed. To maintain a quorum of instances on which voting can be performed, the BTR can restart the failed instance, or replace it with an alternate diverse copy. In addition, to support the master/slave voting scheme, we need some way to overwrite previously-advertised buggy updates. To deal with this, our implementation maintains a history of previously-advertised updates when running this voting scheme. When the hypervisor switches to a new master, all updates in that history that differ from the currently advertised routes are sent out immediately.

4.3 Reducing complexity

It is worth discussing here the role the hypervisor plays in the overall reliability of the system. As we are adding software, this can increase the possibility of bugs in the overall system. In particular, our goals for the design are that (i) the design is simple, implementing only a minimal set of functionality, reducing the set of components that may contain bugs, and (ii) the design is scalable, opening the possibility of formal verification of the hypervisor – a more realistic task than verifying an entire routing software implementation. To achieve these goals, our design only requires the hypervisor to perform two functions: (i) acting as a TCP proxy, and (ii) bootstrapping new instances. Below, we described how these functions are performed with low complexity.

Acting as a TCP proxy: To act as a TCP proxy simply involves accepting connections from one end point (remote or local) and connecting to the other. When there is a TCP connection already, the hypervisor simply needs to accept the connection. Then, upon any exchange of messages (in or out) the hypervisor simply passes data from one port to another. In addition, our design uses voting to make replication transparent to neighboring routers. Here, the update messages are voted upon before being sent to the adjacent router. However, this is simply comparing opaque strings (the attributes) and does not involve understanding the values in the strings.

Overall, our implementation included multiple algorithms and still was only 514 lines of code. These code changes occur only in the hypervisor, reducing potential for new bugs by increasing modularity and reducing need to understand and work with existing router code. From this, we can see that the hypervisor design is simple in terms of functionality and much of the functionality is not in the critical section of code that will act as a single point of failure.

Bootstrapping new instances: To bootstrap new instances requires maintaining some additional state. However, bugs in this part of the code only affect the ability to bootstrap new instances, and do not affect the “critical path” of voting code. One can think of this code as a parallel routing instance which is used to initialize the state of a new instance. Of course, if this instance’s RIB is faulty, the new instance will be started in an incorrect state. However, this faulty state would either be automatically corrected (e.g., if the adjacent router sends a new route update that overwrites the local faulty copy) or it would be determined to be faulty (e.g., when the faulty route is advertised), in which case a new instances is started. Additionally, the RIB that needs to be kept is simply a history of messages received from the adjacent router and therefore is simple. Bootstrapping a new instance also requires intercepting BGP session establishment. Here, the hypervisor simply needs to observe the first instance starting a session (an OPEN message followed by a KEEPALIVE) and subsequent instances simply get the two received messages replayed.

5. Evaluation

We evaluate the three key assumptions in our work:

It is possible to perform voting in the presence of dynamic churn (Section 5.1): Voting is simple to do on fixed inputs, but Internet routes are transient by nature. To distinguish between instances that are still converging to the correct output from those that are sending buggy outputs, our system delays voting until routes become stable, introducing a tradeoff between false positives (incorrectly believing an unstable route is buggy) and detection time (during which
time a buggy route may be used). Since these factors are independent of the precise nature of bugs but depend on update dynamics, we inject synthetic faults, and replay real BGP routing traces.

It is possible for routers to handle the additional overhead of running multiple instances (Section 5.2): Internet routers face stringent performance requirements, and hence our design must have low processing overhead. We evaluate this by measuring the pass-through time for routing updates to reach the FIB or neighboring routers after traversing our system. To characterize performance under different operating conditions, we vary the routing update playback rate, the source of updates (edge vs. tier-1 ISP), and the number of peers.

Running multiple router replicas does not substantially worsen convergence (Section 5.3): Routing dynamics are highly dependent on the particular sequence of steps taken to arrive at the correct route – choosing the wrong sequence can vastly increase processing time and control overhead. To ensure our design does not harm convergence, we simulate update propagation in a network of BTRs, and measure convergence time and overhead. For completeness, we also cross-validate these against our implementation.

5.1 Voting in the presence of churn

To evaluate the ability to perform voting in the presence of routing churn, we replayed BGP routing updates collected from Route Views [6] against our implementation. In particular, we configure a BGP trace replacer to play back a 100 hour long trace starting on March 1st 2007 at 12:02am UTC. The replacer plays back multiple streams of updates, each from a single vantage point, and we collect information on the amount of time it takes the system to select a route. Since performance is dependent only on whether the bug is detected by voting or not, and independent of the particular characteristics of the bug being injected, here we use a simplified model of bugs (based on the model presented in Section 4.2), where bugs add/remove updates and change the next-hop attribute for a randomly-selected prefix, and have two parameters: (i) duration, or the length of time an instance’s output for a particular prefix is buggy, (ii) interarrival time, or the length of time between buggy outputs. As a starting point for our baseline experiments, we assume the length of time a bug affects a router, and their interarrival times, are similar to traditional failures, with duration of 600 seconds, and interarrival time of 1.2 million seconds [34].

5.1.1 Comparison of voting strategies

There is a very wide space of voting strategies that could be used in our system. To explore tradeoffs in this space, we investigated performance under a variety of alternative voting strategies and parameter settings. We focus on several metrics: the fault rate (the fraction of time the voter outputs a buggy route), waiting time (the amount of time the voter waits before outputting the correct route) and update overhead (the number of updates the voter output).

Fault rate: We investigate the fault rate of the voting strategies by injecting synthetic faults and varying their properties. First, we varied the mean duration and interarrival times of synthetic faults (Figures 3 and 4). We found that for very high bug rates, wait-3 (waiting for $K = 3$ out of $R = 3$ copies to agree before selecting the majority result) outperformed master/slave. This happened because wait-3 is more robust to simultaneous bugs than master/slave, as master/slave takes some short time to detect the fault, potentially outputting an incorrect route in the meantime. In addition, unless the bug rate is extremely high, continuous-majority performs nearly as well as wait-3, with similar robustness and update overhead.

Overall, we found that recovery almost always took place within one second. Increasing the number of instances running in parallel ($R$) makes the router even more tolerant of faults, but incurs additional overheads. Also, wait-for-consensus and continuous-majority gain more from larger values of $R$ than the master/slave strategy. For example, when moving from $R = 3$ to $R = 4$ instances, the fault rate decreases from 0.0006% to 0.003% with wait-for-consensus, while with master/slave the fault rate only decreases from 0.089% to 0.06%.

However, there may be practical limits on the amount of diversity achievable (for example, if there is a limited number of diverse code instances, or a bound on the ability to randomize update timings). This leads to the question—if we have a fixed number of diverse instances, how many should be run, and how many should be kept as standbys (not running, but started up on demand)? We found that standby routers were less effective than increasing $R$, but only for small values of $R$, indicating that for large numbers of diverse instances, most instances could be set aside as standbys to decrease runtime overhead. For example, if $R = 3$, under the continuous-majority strategy we attain a fault rate of 0.02%. Increasing $R$ to 4 reduced the fault rate to 0.0006%, while instead using a standby router with $R = 3$ reduced the fault rate to 0.0008%. This happens because buggy outputs are detected quickly enough that failing over to a standby is nearly as effective as having it participate.
in voting at every time step. Because of this, operators can achieve much of the benefits of a larger number of instances, even if these additional instances are run as lower-priority (e.g., only updated during idle periods) standbys.

**Waiting time:** Different voting algorithms provide different tradeoffs between waiting time (time from when a new best-route arrives, to when it is output by the voter) and the fault rate. The master/slave strategy provides the smallest waiting time (0.02 sec on average), but incurs a higher fault rate (0.0006% on average), as incorrect routes are advertised for a short period whenever the master becomes buggy. Continuous-majority has longer wait times (0.035 sec on average), but lower fault rate (less than 0.00001% on average), as routes are not output until multiple instances converge to the same result. The wait-for-consensus strategy’s performance is a function of the parameter $K$—larger values of $K$ increase wait time but decreases fault rate. However, we found that increasing $K$ to moderate sizes incurred less delay than the pass-through time for a single instance, and hence setting $K = R$ offered a low fault rate with only minor increases in waiting time.

**Update overhead:** Finally, we compare the voting strategies in terms of their effect on update overhead (number of routing updates they generate), and compare them against a standard router (std. router). Intuitively, running multiple voters within a router might seem to increase update overhead, as the voter may change its result multiple times for a single routing update. However, in practice, we find no substantial increase, as shown in Figure 5, which plots a CDF of the number of updates (measured over one second intervals). For the master/slave strategy this is expected, since a single master almost always drives computation. In wait-for-consensus, no updates are generated until all instances arrive at an answer, and hence no more than one outbound update is generated per inbound update, as in a standard router. Interestingly, the continuous-majority strategy also does not significantly increase update overhead. This happens because when an update enters the system, the voter’s output will only change when the majority result changes, which can only happen once per update.

### 5.1.2 Performance of fault detection

Protocols today often incorporate thresholds (such as BGP’s MRAI timer) to rate-limit updates. To evaluate the level of protection our scheme provides against unstable instances, as well as the ability to distinguish steady-state from transient behavior, we incorporated a configurable timeout parameter ($T$) in fault detection to identify when a route becomes stable. Figure 6 shows the tradeoff as this parameter varies between the false negative rate (the number of times a non-buggy instance is treated as buggy), and the fault rate (i.e., the false positive rate of the voter, or the fraction of time a buggy route is treated as non-buggy). We found that as $T$ increases, the false negative rate decreases, as larger values of $T$ reduce the probability that transient changes will be considered when voting. The false negative rate does not vary among different voting strategies, as fault detection is only performed on steady-state outputs, and the algorithmic differences between the strategies disappear when performed on outputs that are not dynamically changing. The fault rate increases with $T$, as when a bug does occur, it takes longer to detect it. Interestingly, the fault rate initially decreases with $T$; this happens because for low values of $T$, more instances are treated as buggy, giving fewer inputs to the voter and increasing the probability of an incorrect decision. Overall, we found that it was possible to tune $T$ to simultaneously achieve a low fault rate, low false negative, and low detection time.

### 5.2 Processing overhead

We evaluate the overhead of running multiple instances using our hypervisor with both XORP- and Quagga-based instances running on single-core 3 Ghz Intel Xeon machines with 2 GB RAM. We measure the update pass-through time as the amount of time from when the BGP replayer sends a routing update to when a resulting routing update is received at the monitor. However, some updates may not trigger routing updates to be sent to neighbors, if the router decides to continue using the same route. To deal with this case, we instrument the software router’s source code to determine the point in time when it decides to retain the same route. We also instrument the kernel to measure the FIB pass-through time, as the amount of time from when the BGP replayer sends an update to the time the new route is reflected in the router’s FIB (which is stored as the routing table in the Linux kernel).

Figure 7 shows the pass-through time required for a routing change to reach the FIB. We replayed a Routeviews update trace and varied the number of Quagga instances from 1 to 31, running atop our router hypervisor on a single-core machine. We found the router hypervisor increases FIB pass-through time by 0.08% on average, to 0.06 seconds. Our router hypervisor implementation runs in user space, instead of directly in the kernel, and with a kernel-based implementation this overhead would be further reduced. Increasing the number of instances to 3 incurred an additional 1.7% increase, and to 5 incurred a 4.6% increase. This happens because the multiple instances contend for CPU resources (we found that with multicore CPUs this overhead was substantially lower under heavy loads).
evaluate performance under heavier loads, we increased the rate at which the replayer played back routing updates by a factor of 3000x. Under this heavy load, FIB pass-through times slow for both the standard router and BTR due to increased queuing delays. However, even under these heavy loads, the BTR incurs a delay penalty of less than 23%. To estimate effects on convergence, we also measured the update pass-through time as the time required for a received routing change to be sent to neighboring routers. We found this time to be nearly identical to the FIB pass-through time when the MRAI timer was disabled. As updates are sent immediately after updating the FIB. When MRAI was enabled (even when set to 1 second, the lowest possible setting for Quagga), the variation in delay across instances was dwarfed by delay incurred by MRAI. Finally, we found that switching to the master/slave voting strategy reduces pass-through delay, though it slightly increases the fault rate, as discussed previously in Section 5.1.

5.3 Effect on convergence

Next, we study the effect of our design on network-wide convergence. We do this by simulating a network of BTRs (each with eight virtual router instances) across three network-level graphs: the entire AS-level topology (labeled AS in Figure 8) sampled on Jan 20 2008, AS 3967’s internal network topology as collected from Rocketfuel (labeled 3967), and cliques (labeled CQ) of varying sizes (since a clique contains the “worst case” for routing, allowing potential to explore all n! possible paths in a clique of size n). To determine ordering of when BTRs respond, we run our implementation over routing updates, record pass-through times, and replay them within our simulation framework. Since for the master/slave approach there is no effect on network operation unless a bug is triggered (since the slaves only operate as standbys), we focus our evaluation on the other strategies.

We found several key results. First, as shown in Figure 8, the voting schemes do not produce any significant change in convergence beyond the delay penalty described in previous sections, as compared to a network only containing standard routers. We found this delay penalty to be much smaller than propagation delays across the network, and to be reduced further when MRAI is activated. As the number of instances increases (up to the number of processor cores), continuous-majority’s delay decreases, because it becomes increasingly likely that one will finish early. The opposite is true for wait-for-consensus, as the delay of the slowest instances becomes increasingly large. Next, while we have thus far considered a virtual router level deployment, where voting is performed at each router, we also considered a virtual network deployment, where voting is performed at the edges of the network. In our experiments we ran eight virtual networks and found that this speeds up convergence, as routers do not have to wait for multiple instances to complete processing before forwarding updates. Hence, for small numbers of diverse instances, voting per-router has smaller convergence delay. However, virtual-network approaches require substantially more control overhead than the virtual-router voting schemes. To address this, we found that simple compression schemes [11] that eliminate redundancy across updates could reduce the vast majority of this overhead. Finally, to validate our simulations, we set up small topologies on Emulab [2], injected routing events, and compared with simulations of the same topology. We found no statistically significant difference.

6. DISCUSSION

For simplicity, this paper discusses the one particular design point. However, our architecture is amenable to deployment on varying levels of granularity:

Server-based operation: Instead of running the diverse instances within a single router, their computations may be offloaded to a set of dedicated servers running in the network (e.g., an RCP-like platform [15]). These servers run the router software in virtualized environments, and cross-check the results of routers running within the network. When a buggy result is detected, virtual router instances may be migrated into the network to replace the buggy instance. Alternatively, the servers may be configured to operate in read-only mode, such that they may signal alarms to network operators, rather than participate directly in routing.

Network-wide deployment: Instead of running instances of individual router software in parallel, ensembles of routers may collectively run entire virtual networks in parallel. Here, the outputs of a router are not merged into a single FIB, or as a single stream of updates sent to its neighbors. Instead, each router maintains a separate FIB for each virtual network, and voting is used at border routers to determine which virtual network data packets should be sent on. The advantage of this approach is it allows different routing protocols to be used within each virtual network, making it simpler to achieve diversity. For example, OSPF may be run in one network and IS-IS in another. In addition, convergence speed may be improved, as individual physical routers do not have to wait for their instances to reach a majority before sending a routing update.

Process-level deployment: Our design runs multiple instances of routing software in parallel, and hence incurs some memory overhead. On many Internet routers this is not an issue, due to low DRAM costs, and the fact that
DRAM capacity growth has far exceeded that of routing table growth. That said, if it is still desirable to decrease memory usage, router software may be modified to vote on a shared RIB instead of a FIB. We found the RIB is by far the largest source of memory usage in both Quagga and XORP, incurring 99.3% of total memory usage. Voting on a shared RIB would reduce this overhead by eliminating the need to store separate copies of the RIB across router instances. Here, voting could be performed across multiple routing daemons (e.g., multiple BGP processes within a single instance of Cisco IOS) to construct a single shared RIB. In addition to reducing memory usage, finer-grained diversity may speed reaction (by only cloning and restarting individual processes or threads), and finer-grained control (during times of load, only mission-critical components may be cloned to reduce resource usage). However, code development may become more challenging, since this approach relies on knowing which parts of code are functionally equivalent. To address this, router software could be written to a common API, to allow replication and composition of modules from different code bases while sharing state.

Leveraging existing redundancy: Instead of running multiple instances in parallel, a router may be able to leverage redundant executions taking place at other routers in the network. For example, networks often provision redundant network equipment to protect against physical failures. For example, the VRRP [27] protocol allows multiple routers to act collectively as a single router. Our architecture is amenable to leveraging physical redundancy, as the multiple instances may be deployed across the redundant router instances. In addition, all routers in the ISP compute the same egress set of BGP routes that are “equal” according to the first few steps of the decision process that deal with BGP attributes [24, 15]. To leverage this redundancy, it may be possible to extend our architecture to support voting across multiple router egress sets.

7. RELATED WORK

Software and data diversity has been widely applied in other areas of computing, including increasing server reliability [18], improving resilience to worm propagation [36], building survivable Internet services [28], making systems secure against vulnerabilities [20], building survivable overlay networks [44], building fault tolerant networked file systems [17], protecting private information [43], and recovering from memory errors [12]. Techniques have also been developed to minimize computational overhead by eliminating redundant executions and redundant memory usage across parallel instances [45, 26].

However as discussed in Section 1.3, routing software presents new challenges for SDD (e.g., routers must react quickly to network changes, have vast configuration spaces and execution paths, rely on distributed operations), as well as new opportunities to customize SDD (routers have small dependence on past history, can achieve the same objectives in different ways, have well-defined interfaces). We address these challenges and opportunities in our design. There has also been work studying router bugs and their effects [42, 32], and our design is inspired by these measurement studies. Also, [14] used a graph-theoretic treatment to study the potential benefits of diversity across physical routers (as opposed to diversity within a router). As work dealing with misconfigurations [23, 24] and traditional fail-stop failures [10, 35, 33, 27] becomes deployed we envision router bugs will make up an increasingly significant roadblock in improving network availability.

Our work can be contrasted to techniques which attempt to prevent bugs by formally verifying the code. These techniques are typically limited to small codebases, and often require manual efforts to create models of program behavior. For example, with manual intervention, a small operating system kernel was formally verified [29]. For routing, work has been done on languages to model protocol behavior (e.g., [25]), however the focus of this work is on algorithmic behaviors of the protocol, as opposed to other possible places where a bug can be introduced. In contrast, our approach leverages a small and low-complexity hypervisor, which we envision being possible to formally verify.

Our design leverages router virtualization to maintain multiple diverse instances. Router virtualization is an emerging trend gaining increased attention, as well as support in commercial routers. Our design builds on the high-level ideas outlined in [16] by providing a complete design, several algorithms for detecting and recovering from bugs, and an implementation and evaluation. In addition, our design is complementary to use of models of router behavior [23, 24] and control-plane consistency checks [41, 37], as these models/checks can be run in place of one or more of the router virtual instances. Finally, systems such as MARE (Multiple Almost-Redundant Executions) [45] and the Difference Engine [26] focus on reducing overheads from replication. MARE runs a single instruction stream most of the time, and only runs redundant instruction streams when necessary. The Difference Engine attains substantial savings in memory usage across VMs, through use of sub-page level sharing and in-core memory compression. These techniques may be used to further reduce overheads of our design.

8. CONCLUSIONS

Implementation errors in routing software harm availability, security, and correctness of network operation. In this paper, we described how to improve resilience of networks to bugs by applying Software and Data Diversity (SDD) techniques to router design. Although these techniques have been widely used in other areas of computing, applying them to routing introduces new challenges and opportunities, which we address in our design. This paper takes an important first step towards addressing these problems by demonstrating diverse replication is both viable and effective in building robust Internet routers. An implementation of our design shows improved robustness to router bugs with some tolerable additional delay.

9. REFERENCES

Matthew Caesar

Assistant Professor
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Avenue, Room 3118
Urbana, IL 61801, USA

Cell: (847) 323-2968
Fax: (217) 333-3501
Email: caesar@cs.illinois.edu
Web: http://www.cs.illinois.edu/~caesar/
1 FIELDS OF INTEREST
Simplifying management and improving reliability of distributed systems and networks through principles of self-organization and self-diagnosis, with an emphasis on wide-area networks and networked systems.

2 EDUCATIONAL BACKGROUND

<table>
<thead>
<tr>
<th>Degree</th>
<th>Year</th>
<th>Field</th>
<th>University</th>
<th>Dissertation</th>
<th>Advisors</th>
<th>Minors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ph.D.</td>
<td>2007</td>
<td>Computer Science</td>
<td>University of California, Berkeley</td>
<td>Identity-based Routing</td>
<td>Prof. Randy H. Katz, Prof. Ion Stoica</td>
<td>Statistics/Machine Learning, CS Theory</td>
</tr>
<tr>
<td>M.S.</td>
<td>2004</td>
<td>Computer Science</td>
<td>University of California, Berkeley</td>
<td>Root Cause Analysis of BGP Dynamics</td>
<td>Prof. Randy H. Katz</td>
<td></td>
</tr>
<tr>
<td>B.S.</td>
<td>2000</td>
<td>Computer Science</td>
<td>University of California, Davis</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3 PROFESSIONAL EXPERIENCE

<table>
<thead>
<tr>
<th>Title</th>
<th>Organization</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assistant Professor</td>
<td>University of Illinois at Urbana-Champaign</td>
<td>2008- Present</td>
</tr>
<tr>
<td>Postdoctoral Fellow</td>
<td>Princeton University</td>
<td>2007-2008</td>
</tr>
<tr>
<td>Research Assistant</td>
<td>University of California at Berkeley</td>
<td>2001-2007</td>
</tr>
<tr>
<td></td>
<td>Network Management and Engineering Group</td>
<td></td>
</tr>
<tr>
<td>Research Intern/Consultant</td>
<td>Microsoft Research</td>
<td>2003, 2005</td>
</tr>
<tr>
<td></td>
<td>Systems and Networking Group</td>
<td></td>
</tr>
<tr>
<td>Member of Technical Staff</td>
<td>iScale Inc.</td>
<td>2000-2001</td>
</tr>
<tr>
<td>Software Engineering Intern</td>
<td>Hewlett-Packard</td>
<td>1999</td>
</tr>
<tr>
<td>Software Engineering Intern</td>
<td>Nokia</td>
<td>1998</td>
</tr>
<tr>
<td>Research Intern</td>
<td>Center for Neuroscience</td>
<td>1998</td>
</tr>
<tr>
<td></td>
<td>University of California at Davis</td>
<td></td>
</tr>
<tr>
<td>Software Engineering Intern</td>
<td>Diamond Lane Communications</td>
<td>1997</td>
</tr>
<tr>
<td>Laboratory Assistant</td>
<td>CS Department, Santa Rosa Junior College</td>
<td>1995-1997</td>
</tr>
</tbody>
</table>
4 TEACHING

4.1 Courses Taught

<table>
<thead>
<tr>
<th>Quarter/Year</th>
<th>Course Number and Title</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fall 2008</td>
<td>CS 598 – Advanced Internetworking</td>
<td>New Syllabus</td>
</tr>
<tr>
<td>Fall 2008</td>
<td>Networks reading group</td>
<td></td>
</tr>
<tr>
<td>Spring 2009</td>
<td>CS 438 – Communication Networks</td>
<td></td>
</tr>
<tr>
<td>Fall 2009</td>
<td>CS 498 – Networks and Systems Laboratory</td>
<td>I2CS Course</td>
</tr>
<tr>
<td>Fall 2009</td>
<td>CS 591SN – Systems and Networking Seminar</td>
<td></td>
</tr>
<tr>
<td>Spring 2010</td>
<td>CS 591SN – Systems and Networking Seminar</td>
<td></td>
</tr>
<tr>
<td>Fall 2010</td>
<td>CS 591SN – Systems and Networking Seminar</td>
<td></td>
</tr>
<tr>
<td>Spring 2011</td>
<td>CS 591SN – Systems and Networking Seminar</td>
<td></td>
</tr>
<tr>
<td>Spring 2010</td>
<td>CS 438 – Communication Networks</td>
<td>I2CS Course</td>
</tr>
<tr>
<td>Spring 2011</td>
<td>CS 598 – Advanced Internetworking</td>
<td></td>
</tr>
<tr>
<td>Fall 2011</td>
<td>CS 241 – System Programming</td>
<td></td>
</tr>
<tr>
<td>Fall 2011</td>
<td>CS 591SN – Systems and Networking Seminar</td>
<td></td>
</tr>
<tr>
<td>Spring 2012</td>
<td>CS 498 – Networks and Systems Laboratory</td>
<td></td>
</tr>
<tr>
<td>Spring 2012</td>
<td>CS 591SN – Systems and Networking Seminar</td>
<td></td>
</tr>
<tr>
<td>Fall 2012</td>
<td>CS 591SN – Systems and Networking Seminar</td>
<td></td>
</tr>
</tbody>
</table>

4.2 Other Teaching Experience

Participant, Academy for Excellence in Engineering Education (AE3) Program, Fall 2011-Present. Teaching Assistant, EE 122: Computer Networks, taught by Prof. Scott Shenker and Prof. Ion Stoica at University of California at Berkeley, Fall 2003.
Teaching Assistant, ECS 152B: Computer Networks, taught by Prof. Demet Aksoy at University of California at Davis, Spring 2001.
Teaching Assistant, ECS 152B: Computer Networks, taught by Prof. Dipak Ghosal at University of California at Davis, Winter 2001.
Teaching Assistant, ECS 152B: Computer Networks, taught by Prof. Dipak Ghosal at University of California at Davis, Spring 2000.
Teaching Assistant, ECS 122A: Algorithm Design and Analysis, taught by Prof. Charles Martel at University of California at Davis, Winter 2000.

4.2.1 Postdoctoral Researchers Supervised

1. Theophilus Benson, Department of Computer Science, Fall 2012-Spring 2013. Now an Assistant Professor at Duke University.

4.2.2 Ph.D. Students Supervised

1. Fred Douglas, Department of Computer Science, Spring 2012 - Present.
2. Rashid Tahir, Department of Computer Science, Spring 2012 - Present.
3. Soudeh Ghorbani, Department of Computer Science, Fall 2011 - Present.
4. Wenxuan Zhou, Department of Computer Science, Fall 2010 - Present.
5. Jason Croft, Department of Computer Science, Fall 2010 - Present.
6. Chi-Yao Hong, Department of Computer Science, Fall 2009 - Present. Coadvised with Prof. Brighten Godfrey (CS).
7. Virajith Jalaparti, Department of Computer Science, Fall 2009 - Present.
9. Chia-Chi Lin, Department of Computer Science, Spring 2009 - Present.
10. Ahmed Khurshid, Department of Computer Science, Spring 2009 - Present.

4.2.3 M.S. Students Supervised
1. Brent Mochizuki, Department of Electrical and Computer Engineering, Fall 2008 - Summer 2010. Now at UC Berkeley Space Sciences Laboratory.
2. Firat Kiyak, Department of Computer Science, Fall 2008 - Fall 2009. Now at Microsoft.
3. Fatih Boyaci, Department of Computer Science, Fall 2008 - Fall 2009. Now at Microsoft.

4.2.4 Undergraduate Students Supervised
6. Frank Li, Summer 2012. Information Trust Institute Summer Fellow. Research on Sybil attacks. Currently in the BS program at MIT.


4.2.5 Student Award Winners


4. Chia-Chi Lin, Excellence in Teaching, Graduate Teaching Assistant Award, 2011.

5. Firat Kiyak and Fatih Boyaci, State Farm Prize in Siebel Center’s Computing Habitat Competition, 2009.

6. Chia-Chi Lin, Grand Prize, Qualcomm Q Award, 2009.


5 RESEARCH AND CREATIVE SCHOLARSHIP

5.1 Theses


5.2 Journal Publications


5.3 Conference Publications


3. Chi-Yao Hong, Matthew Caesar, Brighten Godfrey, Finishing Flows Quickly with Preemptive Scheduling, ACM SIGCOMM, August 2012. (acceptance rate=13%)

4. Chi-Yao Hong, Matthew Caesar, Nick Duffield, Jia Wang, Tiresias: Online Anomaly Detection for Hierarchical Operational Network Data, International Conference on Distributed Computing Systems (ICDCS), June 2012. (acceptance rate=13%)

5. Prateek Mittal, Matthew Caesar, Nikita Borisov, X-Vine: Secure and Pseudonymous Routing in DHTs Using Social Networks, Network and IT Security Conference (NDSS), February 2012. (acceptance rate=17.8%)

6. Wenxuan Zhou, Qingxi Li, Matthew Caesar, Brighten Godfrey, ASAP: A Low Latency Transport Layer, ACM CoNEXT, December 2011. (acceptance rate=18.8%)

7. Ahmed Khurshid, Firat Kiyak, Matthew Caesar, Improving Robustness of DNS to Software Vulnerabilities, Annual Computer Security Applications Conference (ACSAC), December 2011. (acceptance rate=16.3%)


15. Chi-Yao Hong, Chia-Chi Lin, Matthew Caesar, *Clockscalpel: Understanding root causes of Internet clock synchronization inaccuracy*, Passive and Active Measurement Conference, March 2011. (Received Best Paper Award)


17. Benny Applebaum, Haakon Ringberg, Michael Freedman, Matthew Caesar, Jennifer Rexford, *Collaborative, Privacy-Preserving Data Aggregation at Scale*, Privacy Enhancing Technologies Symposium (PETS), July 2010. (acceptance rate=28%)

18. Rachit Agarwal, Virajith Jalaparti, Matthew Caesar, Brighten Godfrey, *Stable Path(s) Assignment for Inter-domain Routing*, ICDCS, June 2010. (acceptance rate=14%)


5.4 Workshop Publications


6. Chia-Chi Lin, Matthew Caesar, Kobus van der Merwe, *Towards Interactive ISP Debugging*, ACM HotNets, October 2009. (acceptance rate=16%)


8. Matthew Caesar, Jiawei Han, *Leveraging Data Mining to Improve Internet Security*, NSF workshop on Data and Applications Security, February 2009.


5.5 Books and Book Chapters

5.6 Software


5.7 Research Proposals and Grants

1. **Cyclone: Dynamic Virtualization for Cloud Security**
   
   *Sponsor:* DARPA  
   *Investigators:* Matthew Caesar (PI), Jennifer Rexford  
   *Amount (my share):* $1,200,000 for 4 years  
   *Awarded:* January 2012

2. **Research in Trusted Hypervisors**
   
   *Sponsor:* Northrop Grumman  
   *Investigators:* Matthew Caesar (PI)  
   *Amount (my share):* $15,000 for 1 year (unrestricted gift)  
   *Awarded:* October 2011

3. **CyberShield: Network Virtualization for Cyber Defense**
   
   *Sponsor:* DARPA  
   *Investigators:* Matthew Caesar (PI)  
   *Amount (my share):* $400,000 for 2 years  
   *Awarded:* April 2011

4. **CAREER: Getting RID of Bugs: Realizing Interactive Debugging of Networked Systems**
   
   *Sponsor:* National Science Foundation  
   *Investigators:* Matthew Caesar (PI)  
   *Amount (my share):* $472,179 for 5 years  
   *Awarded:* February 2011
5. Simplifying Attribution in Modern Networked Software with Virtual Networks
   
   **Sponsor:** Boeing
   **Investigators:** David Nicol (PI), Matthew Caesar
   **Amount (my share):** $75,000 for 1 year
   **Awarded:** January 2011

6. FIA: Collaborative Research: NEBULA: A Future Internet that Supports Trustworthy Cloud Computing
   
   **Sponsor:** National Science Foundation Future Internet Architectures (FIA) Program
   **Investigators:** Thomas Anderson - University of Washington, Ken Birman - Cornell University, Matthew Caesar - University of Illinois, Doug Comer - Purdue University, Charles Cotton - U Delaware, Michael Freedman - Princeton University, William Lehr - MIT, David Mazieres - Stanford, Antonio Nicolosi - Stevens Inst. Technology, Jonathan Smith - U Pennsylvania (PI), Ion Stoica - UC Berkeley, Michael Walfish - UT Austin
   **Amount (my share):** $502,602 for 3 years
   **Awarded:** August 2010

7. Research in Network Virtualization
   
   **Sponsor:** AT&T Labs
   **Program:** AT&T Virtual University Research Initiative (VURI) program.
   **Investigator:** Matthew Caesar (PI)
   **Amount (my share):** Two internship slots for students, plus $25,000 for 1 year (unrestricted gift)
   **Awarded:** March 2010

8. Network Software Reliability
   
   **Sponsor:** Defense Advanced Research Projects Agency (DARPA), TCTO program
   **Program:** Computer Science Study Panel (Program to educate early-career Computer Science faculty on DoD needs. Included four week-long visits to military and industrial installations. I was selected as one out of twelve junior faculty in the United States).
   **Investigator:** Matthew Caesar (PI)
   **Amount (my share):** $100,000 for 1 year (unrestricted gift)
   **Awarded:** April 2010

9. Towards bug-tolerant Internet routers
   
   **Sponsor:** Cisco Research
   **Investigator:** Matthew Caesar (PI)
   **Amount (my share):** $100,000 for 1 year (unrestricted gift)
   **Awarded:** May 2010
10. **Fixing the Reliability Problem in Network Software from its Root**

   *Sponsor:* National Science Foundation NeTS-NECO Program  
   *Investigators:* Matthew Caesar (PI), Jennifer Rexford, Yuanyuan Zhou  
   *Amount (my share):* $350,000 for 4 years  
   *Awarded:* August 2008

6 Service and Recognition

6.1 Departmental Service

Preliminary Examination and Thesis Defense Committees:


Other Selected Committees:

1. Fall 2009-present: Student Awards Committee
2. Fall 2009-present: Program of Study Committee
3. Fall 2010-present: Capricious Grading Committee
4. Fall 2010-present: Undergraduate Study Committee
5. Fall 2009-present: Undergraduate Advising Committee
6. Spring 2010-present: Illinois Cyber Security Scholars Program (ICSSP) Admissions Committee and Faculty Advisor
7. Fall 2012-present: ITI DLS Speaker Selection Committee (Chair)

I have also been participating in the ECE Faculty Recruiting Committee meetings (Spring 2012).

6.2 Professional Activities

6.2.1 Memberships in Professional Organizations

Member Association for Computing Machinery (ACM)

Member ACM Special Interest Group on Data Communications (SIGCOMM)

Member USENIX Technical Association

Member Institute of Electrical and Electronics Engineers (IEEE)
6.2.2 Conference and Other Committee Activities

2013 Program Committee, ACM Special Interest Group on Data Communication (SIGCOMM)

2012 Program Committee, Hot Topics in Networks (HotNets-XI)

2012 Program Committee, USENIX Symposium on Networked Systems Design and Implementation (NSDI)

2012 Program Committee, IEEE International Conference on Network Protocols (ICNP)

2012 Panelist, National Science Foundation CNS Program Panel

2012 Judge, ACM Student Research Competition, Grand Finals

2012 Program Committee, ACM Special Interest Group on Data Communication (SIGCOMM)

2012 Program Committee, Passive and Active Measurements Conference (PAM)

2011 Program Committee, ACM International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)

2011 Judge, Student Research Competition, ACM Special Interest Group on Data Communication (SIGCOMM)

2011 Program Committee, IEEE International Conference on Network Protocols (ICNP)

2011 Program Committee, IEEE International Workshop on Quality of Service (IWQoS)

2010 Program Committee, Programmable Routers and Extensible Services of Tomorrow (PRESTO)

2010 Program Committee, IEEE International Conference on Network Protocols (ICNP)

2010 Program Committee, IEEE International Workshop on Quality of Service (IWQoS)

2010 Program Committee, Passive and Active Measurements Conference (PAM)

2009 Program Co-Chair, ACM International Conference on Emerging Networking Experiments and Technologies, Student Workshop (CoNEXT)

2009 Program Committee, ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)

2009 Program Committee, IEEE International Workshop on Quality of Service (IWQoS)

2008 Program Committee, Second ACM SIGCOMM Workshop on Networked Systems for Developing Regions (NSDR)

2008 Program Committee, 16th IEEE Workshop on Local and Metropolitan Area Networks (LANMAN)

2008 Panelist, National Science Foundation Expeditions Program Panel

6.3 Research Honors and Awards

1. ACM SIGCOMM Outstanding Reviewer Award, 2012
2. C.W. Gear Outstanding Junior Faculty Award, 2012
3. Nominee, 2012 National Center for Women & Information Technology REU Faculty Award.
5. UC Berkeley Institute for Preparing Future Faculty, Fellow, 2007.
9. UC Davis Computer Science Department Citation Award, 2000. Highest GPA among 150+ computer science majors in undergraduate program at UC Davis. Rotary Club scholarship, Doyle scholarship.

6.4 Patents


2009 “Method for Reduced Ethernet Table Size,” pending.


2006 “Route Control,” pending.

6.5 Press coverage

   http://www.networkworld.com/community/node/63586

   http://www.computer.org/portal/web/computingnow/archive/news071

   http://www.scientificamerican.com/blog/post.cfm?id=re-thinking-the-internet-with-secu-2010-08-31