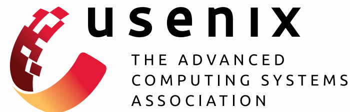


A General Approach to Network Configuration Analysis

Ari Fogel and Stanley Fung, *University of California, Los Angeles;*
Luis Pedrosa, *University of Southern California;* **Meg Walraed-Sullivan,** *Microsoft Research;*
Ramesh Govindan, *University of Southern California;* **Ratul Mahajan,** *Microsoft Research;*
Todd Millstein, *University of California, Los Angeles*

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Open Access to the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15) is sponsored by USENIX

A General Approach to Network Configuration Analysis

Ari Fogel Stanley Fung Luis Pedrosa Meg Walraed-Sullivan
Ramesh Govindan Ratul Mahajan Todd Millstein

University of California, Los Angeles University of Southern California Microsoft Research

Abstract— We present an approach to detect network configuration errors, which combines the benefits of two prior approaches. Like prior techniques that analyze configuration files, our approach can find errors proactively, before the configuration is applied, and answer "what if" questions. Like prior techniques that analyze data-plane snapshots, our approach can check a broad range of forwarding properties and produce actual packets that violate checked properties. We accomplish this combination by faithfully deriving and then analyzing the data plane that would emerge from the configuration. Our derivation of the data plane is fully declarative, employing a set of logical relations that represent the control plane, the data plane, and their relationship. Operators can query these relations to understand identified errors and their provenance. We use our approach to analyze two large university networks with qualitatively different routing designs and find many misconfigurations in each. Operators have confirmed the majority of these as errors and have fixed their configurations accordingly.

1 Introduction

Configuring networks is arduous because policy requirements (for resource management, access control, etc.) can be complex and configuration languages are low-level. Consequently, configuration errors that compromise availability, security, and performance are common [7, 21, 36]. In a recent incident, for example, a misconfiguration led to a nation-wide outage that impacted all customers of Time Warner for over an hour [3].

Prior approaches Researchers have developed two main approaches to detect network configuration errors. The first approach directly analyzes network configuration files [2, 5, 7, 24, 25, 28, 34]. Such a *static* analysis can flag errors proactively, before a new configuration is applied to the network, and it can naturally answer "what if" questions with respect to different environments (i.e., failures and route announcement from neighbors).

However, configurations of real networks are complex, with many interacting aspects (e.g., BGP, OSPF, ACLs, VLANs, static routing, route redistribution); existing configuration analysis tools handle this complexity by developing customized models for specific aspects of the configuration or specific correctness properties. For instance, rcc [7] produces a normalized representation of configuration that lets it check a range of properties that correspond to common errors (e.g., "route validity" of BGP, whether OSPF adjacencies are configured on both ends, and that there are no duplicate router identifiers). Similarly, FIREMAN [34] produces a "rule graph" structure to represent each ACL and analyzes these graphs. This selective focus makes configuration analysis practical, but it also limits the scope of what can be checked. Further, because many aspects of the configuration are not analyzed, it can be difficult for operators to assess how and whether identified errors ultimately impact forwarding.

Researchers have recently proposed a second approach that can be used to detect configuration errors: analyzing the data plane snapshots (i.e., forwarding behavior) of the network [13, 14, 22, 37]. Unlike with static analysis, any configuration error that causes undesirable forwarding can be precisely detected, because the data plane reflects the combined impact of all configuration aspects. Further, because the data plane has well-understood semantics and can be efficiently encoded in various logics, a wide range of forwarding properties can be concisely expressed and scalably checked with off-the-shelf constraint solvers.

Unfortunately, analysis of data plane snapshots cannot prevent errors proactively, before undesirable forwarding occurs. Further, once a problem is flagged, the operators still need to localize the responsible snippets of configuration. This task is challenging because the relationship between configuration snippets and forwarding behavior is complex. The responsible snippet is not necessarily

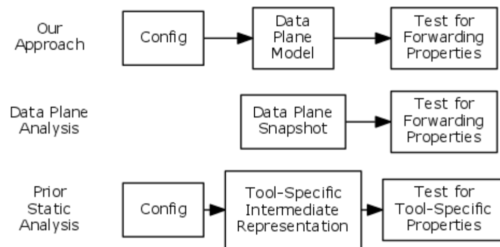


Figure 1: *Our approach versus prior approaches.*

the most recent configuration change either; the impact of an erroneous change may only manifest long after it is introduced. For instance, the impact of erroneously configured backup paths will manifest only after a failure.

Our approach We develop a new, general approach to statically analyze network configurations that combines the strengths of the approaches above. Instead of using a customized representation, our analysis *derives the actual data plane that would emerge* given a configuration and environment. Figure 1 illustrates our approach. With it, as with prior static approaches, operators can detect errors proactively and conduct "what if" analysis across different environments. Further, as with data-plane analysis approaches, they can easily express and check a wide range of correctness properties and directly understand the impact of errors on forwarding.

Realizing our approach The principal challenge that we face is the need to derive a faithful data plane for a given configuration and environment. Our analysis must balance two competing concerns. It must be detailed and low-level in order to produce an accurate data plane, which requires us to tractably reason about all aspects of configuration and their interactions, as well as a plethora of configuration parameters and directives. At the same time, the analysis must provide a high-level view that allows operators to understand the identified errors and map them back to responsible configuration snippets.

We address this challenge in our tool, called Batfish, by implementing our analysis fully declaratively. We translate the network configuration and environment into a variant of Datalog and also use this language to express the behaviors of the various protocols being configured. Executing the resulting Datalog program produces logical relations that represent the data plane as well as relations for various key concepts in the computation, e.g., the best route to a destination as determined by a particular protocol. We use an automatic constraint solver to check properties of the resulting data plane and produce concrete packets that violate these properties. Finally, those packets are fed back into our declarative model, inducing more relational facts (e.g., the path taken, the ACL rules encountered along the way). These relations and the ones described above provide a simple ontology for understanding errors and their provenance.

Operators can query Batfish for any correctness property that can be expressed as a first-order-logic formula over the data-plane relations. However, Batfish can find errors even without operator input; by default the tool checks three novel properties related to the consistency of forwarding. Our *multipath consistency* property requires that, in the presence of multipath routing, packets of a flow are either dropped along all paths they traverse or reach the destination along all paths. Our *failure consistency* and *destination consistency* properties uncover errors that respectively limit fault tolerance and make the network vulnerable to illegitimate route announcements.

We used Batfish to find violations of these three properties in the configurations of two large university campus networks. We find many violations of each type, the majority of which the operators confirmed to be configuration errors. Because of helpful provenance information provided by Batfish, several of the errors were fixed within a day of us reporting them.

Summary We develop a new approach and a practical tool to analyze network configurations. At its heart is a high-fidelity declarative model of low-level network configurations. We believe that this model is useful beyond detecting configuration errors. For instance, researchers have proposed high-level, declarative languages to program networks [9, 18, 19, 26], but a major hurdle in adopting them is migrating a network while faithfully preserving its forwarding policies. Our model can provide a migration path. Our tool is publicly available [1] for others to use and explore various use cases.

2 Background and Motivation

This section provides background on routing in today's networks and motivates our approach.

2.1 Background

A network forwards packets through a sequence of routers and switches. The *data plane* state of each device determines how packets with a given header are handled (e.g., dropped, forwarded to a specific neighbor, or load balanced across multiple neighbors). This state is generated by the *control plane*. In today's networks, the control plane is specified through device configuration, which uses vendor-specific languages and includes as-

pects such as ACLs that specify packet filtering policies, static routes for IP address prefixes that are directly connected, and directives for one or more routing protocols. Configurations of all devices, combined with the current topology and dynamic information exchanged between neighboring devices, determine the current data plane.

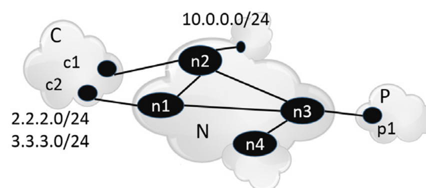
A network managed by some administrative entity is known as an *autonomous system* (AS). Within an AS, information on network topology and connected destinations is exchanged using interior gateway protocols such as OSPF [23], a protocol that computes least-cost paths. BGP [29], a protocol that accommodates policy constraints, is used across ASes. Routers announce destination IP address prefixes to which they are willing to carry traffic from a neighboring AS. Local policy determines if a received announcement is acceptable (e.g., whether the announcer can be trusted to have a path to the destination prefix) and which one among the multiple announcements for the same prefix should be selected (e.g., based on commercial relationships).

As an aside, in the SDN paradigm, which has gained significant attention of late, the control plane is specified using a control program instead of configuration. We focus on the configuration-based paradigm because it currently dominates and continues to be a cause of subtle errors. Even if SDNs become dominant, many networks will likely continue to be configuration-based, in the same way that legacy software is prevalent despite the advent of higher-level programming technologies.

2.2 Motivation

Given the complexity of network configurations, errors are common [21, 31, 36], and operators need good tools to flag potential errors. Consider network N pictured at the top of Figure 2, with two neighboring ASes. P is a large provider AS, and C is a customer AS that owns two destination prefixes. Router $n2$ is directly connected to an internal private network with prefix 10.0.0.0/24. The operators intend that this network be available to C, but not to P or other parts of N not servicing C.

The bottom of Figure 2 shows configuration snippets that implement this specification, loosely based on Cisco's IOS language. The first two lines of $n1$'s configuration specify that it runs OSPF on interfaces that connect it to $n2$ and $n3$, each with routing cost metric of 1. The next two specify that it runs BGP with $c2$ and will accept only announcements for prefixes that match the prefix list *PL_C*. Router $n2$ is similarly configured except that it also redistributes (i.e., advertises) connected net-



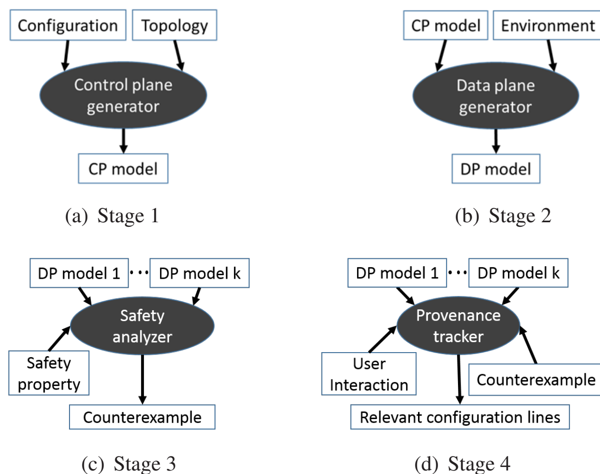

```
//----------Configuration of n1----------
1 ospf interface int1_2 metric 1
2 ospf interface int1_3 metric 1

3 prefix-list PL_C 2.2.2.0/24 3.3.3.0/24

4 bgp neighbor c2 AS C apply PL_C

//----------Configuration of n2----------
1 ospf interface int2_1 metric 1
2 ospf interface int2_3 metric 1
3 ospf-passive interface int2_5 ip 10.0.0.0/24
4 ospf redistribute connected metric 10

5 prefix-list PL_C 2.2.2.0/24

6 bgp neighbor c1 AS C apply PL_C

//----------Configuration of n3----------
1 ospf interface int3_1 metric 1
2 ospf interface int3_2 metric 1
3 ospf interface int3_4 metric 1

4 ospf redistribute static metric 10

5 bgp neighbor p1 AS P Accept ALL

6 static route 10.0.0.0/24 drop, log
```

Figure 2: *Example network configuration snippets.*

works through OSPF. Router $n3$ is configured to accept all prefix announcements from $p1$ and to redistribute into OSPF all statically configured networks. To isolate prefix 10.0.0.0/24 from nodes not on the path to C, the operator installs a static discard route with logging at $n3$ (line 6). This route is redistributed (line 4) so $n4$ need not be directly aware of this route. This setup prevents P and $n4$ (and hosts behind them) from accessing 10.0.0.0/24 and enables the operators to discover any attempts.

The example above is based on actual configurations of a large university network that we have analyzed using Batfish, and, despite its simplicity, it has at least two errors. The first error is that 3.3.3.0/24 is missing from the definition of *PL_C* in $n2$, and thus $n2$ will drop announcements and not provide connectivity for this prefix. This error may go unnoticed when the configuration is applied since connectivity to 3.3.3.0/24 is available through $n1$. But when $n1$, $c2$ or link $c2$-$n1$ fails, all connectivity to 3.3.3.0/24 will be lost. The end result of this error is lack of fault tolerance and poor load balancing (since link $c2$-$n1$ carries all traffic for 3.3.3.0/24).

The second error is more subtle. Because $n2$ and $n3$ redistribute connected and static networks, respectively, $n1$



(a) Stage 1

(b) Stage 2

(c) Stage 3

(d) Stage 4

Figure 3: *The four stages of* Batfish *workflow.*

will learn paths to 10.0.0.0/24 from both these neighbors, and the paths will have the same routing cost. Under these conditions, the default is multipath routing; that is, $n1$ will send packets to 10.0.0.0/24 through both neighbors. However, only packets sent through $n2$ will reach the destination since $n3$ will drop such packets. Thus, traffic sources will experience intermittent connectivity.[1]

No existing technique can find both of these errors proactively, before the buggy configuration is applied. Data plane analysis can detect reachability issues but it will not find the first error until a failure occurs that breaks reachability to 3.3.3.0/24. Prior static analysis techniques, which target specific misconfiguration patterns in particular protocols, will not detect the second error, as that requires a precise model of the semantics of OSPF, connected routes, static routes, and their interaction through redistribution. Batfish finds both errors proactively as violations of failure consistency and multipath consistency properties (discussed below), respectively. It can do this because it (a) statically analyzes configurations, and (b) derives a faithful model of the data plane from configurations.

3 An Overview of Batfish

We now overview our approach to static analysis of network configurations, as implemented in Batfish. Figure 3 shows the four stages of its workflow.

[1] Such intermittent connectivity can go unnoticed. To prevent reordering, multipath routing typically maps packets with the same 5-tuple (source and destination addresses and ports, and the protocol identifier) to the same path. If a connection gets unlucky and is initially mapped to the dropping path, subsequent retries (with a different source port) will likely map it to the valid path, after which all packets will be delivered.

```
//Part 1a: Facts on OSPF interface costs
OspfCost(n1, int1_2, 1)
...(remaining OSPF interfaces)
//Part 1b: Facts on OSPF adjacencies
OspfNeighbors(n1, int1_2, n2, int2_1).
OspfNeighbors(n1, int1_3, n3, int3_1).
OspfNeighbors(n2, int2_3, n3, int3_2).
...(symmetric facts)

//Part 2: Rules that capture basic OSPF logic
BestOspfRoute(node, network, nextHop, nhIp, cost) <-
   OspfRoute(node, network, nextHop, nhIp, cost),
   MinOspfRouteCost[node, network] = cost.

MinOspfRouteCost[node, network] = minCost <-
   minCost = agg<<cost = min(cost)>>:
      OspfRoute(node, network, _, _, cost).

OspfRoute(node, network, nextHop, nextHopIp, cost) <-
   OspfNeighbors(node, nodeInt, nextHop, nextHopInt),
   InterfaceIp(nextHop, nextHopInt, nextHopIp),
   ConnectedRoute(nextHop, network, nextHopConnInt),
   OspfCost(node, nodeInt, nodeIntCost),
   OspfCost(nextHop, nextHopConnInt, nextHopIntCost),
   cost = nodeIntCost + nextHopIntCost.

OspfRoute(node, network, nextHop, nextHopIp, cost) <-
   OspfNeighbors(node, nodeIntCost, nextHop, nhInt),
   InterfaceIp(nextHop, nhInt, nextHopIp),
   OspfNeighbors(nextHop, _, hop2, _),
   BestOspfRoute(nextHop, network, hop2, _, subCost),
   node != secondHop,
   cost = subCost + nodeIntCost.
```

Figure 4: *A subset of the control plane model for the OSPF portion of the configuration in Figure 2.*

3.1 From Configuration to Data Plane

The first two stages of Batfish transform the given network configuration into a concrete data plane. Stage 1 generates a logical model of the control plane. This model compactly represents the network configuration and topology and the computation that the network routers carry out collectively to produce the data plane.

Our control plane model is defined in a variant of Datalog called LogiQL, which is the language of the LogicBlox database engine [10, 17]. Beyond basic Datalog, LogiQL supports integers, arithmetic operations, and aggregation (e.g., minimum).

A key challenge addressed in our work is faithfully encoding the semantics of a range of low-level configuration directives in a high-level, declarative language. As we detail below, the declarative nature of our control plane and the resulting data plane models provides a simple ontology of relations that operators can query to understand the provenance of errors. While imperative code could have provided this capability, our declarative implementation gives us this information for free.

As an example, Figure 4 shows a portion of the control plane model for the configuration in Figure 2. Part 1 of

the model has logical facts that encode the configuration and topology information. In the figure, we show the OSPF-related information, namely the link costs and adjacencies. Part 2 has a generic set of rules that capture the semantics of the control plane for an arbitrary network. In the figure, we show some of the rules for OSPF routing. The first rule defines the best OSPF route to be the route with the minimum cost. The second rule defines the minimum cost by simply aggregating over all OSPF routes to find the minimal element. The last two rules effectively implement a shortest-path computation.

The second stage of Batfish takes an *environment* as an additional input, which facilitates performing "what if" analysis. The environment consists of the up/down status of each link in the network as well as a set of route announcements from each of the network's neighboring ASes. It is represented as a set of logical facts.

We derive the data plane by executing the LogiQL program that represents the control plane model and the environment. This execution is essentially a fixed point computation, i.e., all rules are fired iteratively to derive new facts, until no new facts are generated. The resulting data plane model includes the forwarding behavior of individual routers as logical facts that indicate whether a packet with certain headers should be dropped (e.g., `Drop(node, flow)`) or forwarded to a neighbor (e.g., `Forward(node, flow, neighbor)`). The data plane model also includes facts for all of the intermediate predicates used in the rules; this enables users to easily investigate the provenance of various aspects of the data plane. For instance, a particular `Forward` predicate may have been derived from a `BestOspfRoute` fact in the control plane model, meaning that the chosen route came from OSPF, and that fact in turn was derived from a particular set of OSPF link costs in the configuration.

Unlike prior static analysis techniques, the first two stages of Batfish analyze all aspects of network configuration that are relevant to the data plane, irrespective of the correctness properties of interest. The resulting data plane thus faithfully captures the forwarding behavior induced by the given configuration, topology, and environment (but see §3.3 for limitations).

3.2 From Data Plane to Configuration Errors

The last two stages of Batfish identify and localize configuration errors. In the third stage, we analyze one or more data planes to check desired correctness properties. The tool can check any property expressible as a first-order-logic formula over the relations that represent one or more data planes of interest. This is accomplished by translating the data-plane relations and the correctness property to the language of the Z3 constraint solver [20, 35], which then either verifies the property or provides one or more counterexamples, which consist of a concrete packet header and originating router.

In addition to user-specified properties, Batfish checks for traditional reachability properties such as the absence of black holes and loops, as well as three new properties that go beyond reachability to ensure correctness of paths through the network and their relation to one another (§4). Because the first two stages of Batfish are property-independent, we can generate the data planes of interest once and then check any number of properties over these data planes without having to re-create them.

The final stage helps operators understand property violations, in order to properly repair the network configuration. It works by logically simulating the behavior of counterexample packets through the network on top of our logical data plane model. As before, various logical facts will be produced during this simulation. Some of these facts directly provide provenance information to the user, such as the particular line of an ACL that caused the packet to be dropped. The user can also investigate additional provenance relationships by querying the full logical database, which contains facts about the control plane, the data plane, and their relationship, to understand why particular facts were generated.

To understand the process of uncovering the root cause of an error found by Batfish, consider the second error described for the example network in §2.2. Batfish detects this error as a *multipath inconsistency*. See §4 for the formal definition, but informally, it means that packets of a flow can be dropped along some paths but carried to destination along some others. This inconsistency is represented it by the following logical fact:

```
FlowMultipathInconsistent(Flow<src=n1, dstIp=10.0.0.0>)
```

The operator can then query the `FlowTrace` relation of Batfish, which produces a traceroute-like representation of the paths taken by the counterexample flow:

```
FlowTrace(Flow<src=n1, dstIp=10.0.0.0>,
   [n1:int1_2 -> n2:int2_1]:accepted])
FlowTrace(Flow<src=n1, dstIp=10.0.0.0>,
   [n1:int1_3 -> n3:int3_1]:nullRouted)
```

To understand why the flow was accepted by *n*2 but dropped by *n*3, the operator can then query the `FlowMatchRoute` relation to see which routes the flow matched at each router in the above paths:

```
FlowMatchRoute(Flow<src=n1, dstIp=10.0.0.0>, n1,
   Route<prefix=10.0.0.0/24, nextHop=n2, 10, ospfE2>)
```

```
FlowMatchRoute(Flow<src=n1, dstIp=10.0.0.0>, n1,
   Route<prefix=10.0.0.0/24, nextHop=n3, 10, ospfE2>)
FlowMatchRoute(Flow<src=n1, dstIp=10.0.0.0>, n2,
   Route<prefix=10.0.0.0/24, int=int2_5, connected>)
FlowMatchRoute(Flow<src=n1, dstIp=10.0.0.0>, n3,
   Route<prefix=10.0.0.0/24, DROP, static>)
```

Here we see that $n1$ has two external type-2 (redistributed, fixed-cost) OSPF routes to 10.0.0.0/24 with equal cost of 10. The first points to $n2$ where the network is directly-connected, and the second points to $n3$ which has a static discard route for the destination. To prevent the discard route at $n3$ from being active on $n1$, the operator may increase the exported cost of this route on $n3$ in line 4 of Figure 2.

3.3 Discussion

Since Batfish strives to model all aspects of configuration that impact forwarding, when checking for correctness our approach incurs no false positives and no false negatives; each identified error is a real violation of the checked property, and all violations are identified. However, this guarantee has three caveats from a pragmatic perspective. First, like other configuration analysis tools, we assume that routers behave as expected based on their configurations. We cannot catch errors due to bugs in router hardware or software (e.g., BGP implementation).

Second, Batfish analyzes a network under a given set of environments, which are a subset of all possible environments. Therefore, Batfish can miss errors that occur only in environments that the operator has not supplied. Further, operators may supply an infeasible environment to Batfish. For instance, the routing announcements from C1 and P1 in Figure 2 may be correlated in some complex way because those ASes are connected through a path that is not visible to our analysis. In this case, errors identified by Batfish may be spurious since a particular analyzed data plane might never occur in reality.

Finally, Batfish may encounter configuration features that are currently not implemented (e.g., the internal 'color' metrics of Juniper) but may influence local route selection. If that happens, the tool warns users that the guarantee may not hold. There is a qualitative difference, however, between the incompleteness of Batfish and of prior configuration analysis tools. Because Batfish uses the data plane as an intermediate representation, currently-unimplemented features can be mapped to this representation simply by adding logical rules to our control-plane model for how they impact forwarding. Because prior tools use custom intermediate representations or custom checkers, it may be difficult or impossible to use them to model and reason about some new features. Currently, Batfish models a rich enough subset of the configuration space (§6) to precisely analyze two large university networks.

4 Consistency Properties

Batfish can take as input any specification of intended network behavior and automatically check whether the network indeed behaves as expected. For instance, the operator might specify that the network should not carry packets from one particular neighboring AS to another. However, to simplify the task of finding potential errors, we also propose three safety properties that were motivated by discussions with network operators and require little or no input from users. These properties flag different forms of inconsistencies in the network behavior. Prior work on verification in several domains has shown that inconsistent behavior often points to bugs [6, 7].

Our properties are expressed using two auxiliary predicates which we define first. Let E be the environment used to generate the data plane model in Stage 2 of our pipeline. We define predicates $\mathsf{accepted}_E(H,S,D)$ and $\mathsf{dropped}_E(H,S,D)$, which hold if there is some path through the network for which header H is eventually accepted and dropped, respectively, at node D when injected into the network at node S. "Accepted" implies that the packet either reaches its destination or is forwarded outside the modeled network. We simulate packets as being sent along all equal-cost paths, so accepted and dropped are not mutually exclusive. It is straightforward to define these predicates in terms of the logical relations that comprise the data plane. Below we sometimes omit the last argument to the accepted predicate when it is irrelevant, as shorthand for the formula $\exists D$: $\mathsf{accepted}_E(H,S,D)$; a similar shorthand is used for the dropped predicate.

4.1 Multipath Consistency

Multipath consistency is a property that is relevant to networks that use multipath routing and it captures the following expected behavior: all packets with the same header should be treated identically in terms of being accepted or dropped, regardless of the path taken through the network. Formally, we say that the network with environment E exhibits *multipath consistency* if the following condition is true:

$$\forall H,S : \text{accepted}_E(H,S) \Rightarrow \neg\text{dropped}_E(H,S)$$

In other words, every packet is either accepted on all paths or dropped on all paths. A counterexample to this

formula consists of a concrete packet header and source node such that it is possible for the header to be both accepted and dropped depending on the path taken.

4.2 Failure Consistency

Networks are typically designed to be tolerant to some number of faults. For example, a particular node or link may have been intended to be used as a backup for another node or link. However, it can be difficult for operators to reason about whether the network configuration is indeed as fault tolerant as intended.

We define a general notion for verifying fault tolerance of a network configuration. Let E' be the network environment identical to E but with a subset of links or nodes considered failed. This subset is drawn from the class of failures to which the network is designed to be fault tolerant (e.g., all single-link failures). We say that the network exhibits *failure consistency* between E and E' if the following condition is true:

$$\forall H,S : \text{accepted}_E(H,S) \Rightarrow \text{accepted}_{E'}(H,S)$$

A counterexample to this formula is a concrete packet header and source node such that the packet is accepted under environment E but dropped under E'. Of course, packets destined for any interface that is failed in E' should not be considered counterexamples to failure consistency. Thus, the full property definition, which we omit for simplicity, includes an extra condition that requires H to be destined for an active interface in E'.

4.3 Destination Consistency

Customer ASes of a given network are often expected to have disjoint IP address spaces, sometimes assigned by the network itself. In such cases, the intended network configuration is to allow a customer AS to only send route announcements for its own address space, ensuring that it only receives packets destined to itself. Our destination consistency property captures this expectation. Let E be the network environment with only customer ASes (i.e., provider and peer AS nodes are considered failed) and E' be an identical environment but with all links to a customer AS C considered failed. Then we say that the network exhibits *destination consistency* for C if the following condition is true:

$$\forall H,S : \forall D \in C : \\ \text{accepted}_E(H,S,D) \Rightarrow \neg\text{accepted}_{E'}(H,S)$$

In other words, any packet that is accepted by some node D in the AS C should not be accepted once C is removed.

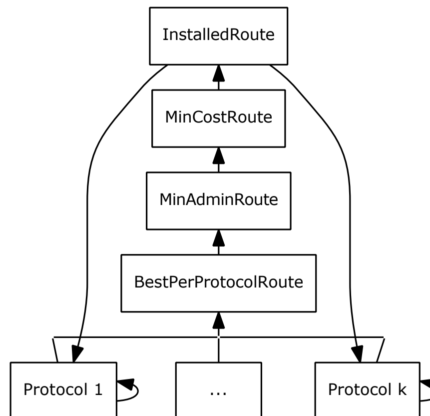


Figure 5: *Information flow for computation of the RIB.*

A counterexample to this formula consists of a concrete packet header, source node, and destination node D in AS C such that the packet is accepted at D under environment E and is accepted somewhere in E'.

5 The Four Stages of Batfish

In this section we present details on each of the four stages in the Batfish pipeline (Figure 3).

5.1 Modeling the Control Plane

Batfish's first stage takes configuration files and network topology as input, and it outputs a control plane model that captures the distributed computation performed by the network. The input information is first parsed into an intermediate data structure, which is then translated into a set of logical *facts*, each associated with a particular *relation*. For example, `SetIpInt(Foo, f0/1, 1.2.3.4, 24)` says interface `f0/1` of node `Foo` has IP address 1.2.3.4 with a 24-bit subnet mask.

These base facts are combined with a set of logical rules that specify how to infer new facts. These rules capture route computation for various protocols. In more detail, each node may be configured to run one or more routing protocols (e.g., OSPF, BGP, etc.). At each node, each protocol iteratively computes its best route to each destination in the network using information learned from neighbors. The available routes to destinations are stored in a routing information base (RIB). While RIB formats vary, a typical RIB entry minimally contains a destination network, the IP address of the next hop for that network, and the protocol that produced the entry. When multipath routing is being used, multiple best routes may be selected for a destination.

Our routing rules capture the process by which RIB entries are generated at each node. Figure 5 shows how we model this process. The model consists of four main relations, each representing a set of routes, and the edges denote the dependencies among these sets.

`BestPerProtocolRoute` is the set of routes that are optimal according to the rules of one of the routing protocols. Protocol-specific rules are defined in terms of a set of relations that represent facts from the configuration and topology information. For example, the OSPF rules shown earlier depend on configured link costs. As Figure 5 shows, our model is modular with respect to such protocols, and adding a new protocol simply requires rules for producing its optimal routes.

`MinAdminRoute` is the subset of `BestPerProtocolRoute` with only routes that have minimal *administrative distance*, a protocol-level configuration parameter. That is, `MinAdminRoute` contains a route *R* to destination *D* from `BestPerProtocolRoute` if the protocol that produced *R* has an administrative distance no higher than that of any other protocol that produced a route to *D*.

`MinCostRoute` is the subset of `MinAdminRoute` with only those routes that have minimal *protocol-specific cost*. That is, `MinCostRoute` contains a route *R* to destination *D* from `MinAdminRoute` if *R* has a protocol-specific cost no higher than that of any other route to *D* in `MinAdminRoute`.

`InstalledRoute` is the set of routes that are selected as best for the node. This set is identical to `MinCostRoute` but is given a new name for clarity.

In general, the set of candidate routes produced by a routing protocol may depend on the current state of the RIB, as well as the internal state of that protocol and the latest messages it has received. We have an edge from `InstalledRoute` to each protocol to illustrate the dependence on previous state, and also to model any redistribution of installed routes from one protocol to another. Thus, these edges signify that producing the RIB requires computing the fixed point of the function that generates the next intermediate state of the RIB.

Figure 6 shows key LogiQL rules for the relations in Figure 5. The `agg` keyword refers to an aggregation; in this case we are finding the tuples of a relation whose aggregated variable is minimal among all the tuples. In addition to such generic rules, we implement LogiQL rules for several routing protocols, and as noted above, a new protocol can be added completely modularly.

```
InstalledRoute(node, network, nextHop,
 nextHopIp, admin, cost, protocol) <-
  MinCostRoute(node, network, nextHop,
   nextHopIp, admin, cost, protocol)

MinCostRoute(node, network, nextHop,
 nextHopIp, admin, minCost, protocol) <-
  minCost = MinCost[node, network],
  MinAdminRoute(node, network, nextHop,
   nextHopIp, admin, minCost, protocol)

MinCost[node, network] =  minCost <-
   agg<<minCost = min(cost)>>
   MinAdminRoute(node, network, _, _, _, cost, _)

MinAdminRoute(node, network, nextHop,
 nextHopIp, minAdmin, cost, protocol) <-
  minAdmin = MinAdmin[node, network],
  BestPerProtocolRoute(node, network,
   nextHop, nextHopIp, minAdmin, cost,
   protocol)

MinAdmin[node, network] =  minAdmin <-
   agg<<minAdmin = min(admin)>>
   BestPerProtocolRoute(node, network,
    _, _, admin, _, _).
```

Figure 6: *LogiQL code for route-selection.*

5.2 Building the Data Plane

The data plane of the network is the forwarding information base (FIB) for each node. The FIB determines an appropriate action to take when a packet reaches a particular interface. For the purposes of this paper, that action is either to forward the packet out of one or more interfaces, to accept the packet, or to drop the packet. The second stage of Batfish generates one data plane per user-specified environment.

In Batfish, the FIB for a node consists of the node's RIB, the configured ACLs for the node's interfaces, and rules for using these items to forward traffic. The data-plane generator starts by simply executing the LogiQL program that is the output of Stage 1, which is the control-plane model, to produce the RIB for each node. Before doing so, LogiQL facts to represent the provided environment are added to the model. Specifically, the facts indicate which interfaces in the network are up, allowing us to model network failures, and which routes are being advertised by neighboring networks.

A LogiQL program consisting of a set of base facts and rules is executed as follows. When a rule body (to the right of <− in Figure 6) is satisfiable by existing facts, a new fact is derived and added to the relation in the rule head (to the left of <−). This process repeats until quiescence. At this point, the facts in the `InstalledRoute` relation represent the RIB for each node. We then represent the FIB as a new set of logical rules that make forwarding decisions, given the RIB information as well as the per-interface ACLs, which were converted to logical

facts in Stage 1.

The rules for the FIB are as follows. When a packet arrives on an interface, the rules first check whether the interface has an incoming ACL. If so, and if the packet's header is not allowed by that ACL, the packet is dropped. Otherwise, if the destination IP address of the header is assigned to any interface of the node, then the packet is accepted. Otherwise, the rules check the RIB for entries with networks that are longest-prefix matches for the destination IP address of the header. For each such route, the interface corresponding to that route's next hop is determined as follows: if the route is directly connected on an interface, that interface is selected. Otherwise, the rules use the next hop of the route that is a longest prefix match for the address of the original next hop, recursively, until a directly connected route is found. Finally, the packet is forwarded out that interface if the interface's outgoing ACL permits it, and dropped otherwise.

5.3 Property Checking

After Stage 2, users have access to the full power of LogiQL to ask queries about both the control and data planes. Moreover, these queries can directly employ the relations in our high-level conceptual model. For example, users can query the `BestOspfRoute` relation to find the best OSPF route(s) to a particular destination on a particular node. Further, by employing multiple relations in a query users can easily obtain even richer information, such as the set of all BGP advertisements for a particular prefix that were rejected by an incoming route-map on at least two nodes. In this way, users can interactively investigate various aspects of the network's forwarding behavior as well as their provenance.

In addition to user-directed exploration, Batfish supports systematic checking of correctness properties, to find errors and to prove their absence. By default it checks the properties in §4, but operators can supply additional properties, expressed as first-order formulas over the relations in our data plane model. Depending on the property, Batfish requires one or more data plane models that differ in their environment (e.g., link failures).

Batfish uses Network Optimized Datalog (NoD) [20], a recent extension of the Z3 constraint solver, to identify violations of correctness properties. The properties we check are decidable and can be expressed precisely in NoD and Z3, so Batfish is guaranteed to find a counterexample if one exists, modulo resource limitations. In the rest of the paper, we use *NoD* to refer to the NoD extension to Z3 and use *Z3* to refer to the vanilla Z3 solver (which we also use). To check a property P, we ask NoD if its negation $\neg P$ is satisfiable in the context of the given data plane models. If not, the property holds. If so, NoD provides the complete boolean formula expressing how to satisfy the negation of the property. This formula is a set of constraints on a packet header and the interface at which the packet is injected into the network. We then query Z3 to solve these constraints, thereby producing a concrete counterexample that violates P.

5.4 Provenance Tracking

The final stage of Batfish helps users to localize the root cause of identified property violations. First, each counterexample from the previous stage is converted into a concrete test flow in terms of our LogiQL representation of the data plane. Then, this test flow is "injected" into our logical model, causing LogicBlox to populate relevant relations with facts that indicate the path and behavior of the flow through the network. Many of the produced facts include explicit provenance information, and as demonstrated in §3.2, users can iteratively query the populated relations to map errors back to their sources in the configuration files.

6 Implementation

We implemented Batfish using Java and the Antlr [27] parser generator. Its source comprises 21,504 lines of Java code, 13,214 lines of Antlr code across 2,410 grammar rules, and 5,696 lines of LogiQL code across 386 relations. The bulk of the Java and Antlr code corresponds to Stage 1 of Batfish, which converts configurations to LogiQL facts.

To manage the complexity supporting diverse configuration languages and diverse directives within a language (with overlapping functionality), we devised a vendor- and language-agnostic representation for control plane information. We first translate the original configuration files to our representation, and the rest of the analysis uses this representation exclusively. Therefore, support for new languages or directives can be added by implementing appropriate translation routines, without having to change the core analysis functionality.[2,3]

We currently support configuration languages of Cisco

[2] This analysis structure is akin to LLVM [16], which facilitates analysis of code written in multiple programming languages by first converting the code to a common representation.

[3] We hope that in the future router vendors would supply the translation routines as they best understand the semantics of their languages and directives.

IOS, Cisco NXOS, Juniper JunOS, Arista, and Quanta. Our models of the control and data plane are rich enough to capture the behavior of many real, large networks. We faithfully model static routes, connected networks, interior gateway protocols (e.g., OSPF, including areas), BGP, redistribution of routes between routing protocols, firewall rules, ACLs, multipath routing, VLANs, forwarding based on longest-prefix matching, and policy routing. We currently do not model MPLS [30] or packet modification (e.g., NATs).

A semantic mismatch in encoding configuration directives in LogiQL is for regular expression matching. Such matching may be used for BGP communities and AS-paths but is not supported by LogiQL. We implement community-matching by precomputing the result of the match for all communities mentioned in configuration files and the environment. This strategy does not work for AS-path matching because AS-paths are lists (where order matters; communities are sets) and all possible AS-paths are not known statically.

Based on the observation that regular expressions in configuration files tend to be simple, we implement matching only for regular expressions that match sub-paths of size two or less. For example, if the regular expression is `.*[5-10][10-15].*`, we use LogiQL predicates that are true when the AS-path, encoded as a LogiQL list, has an item between 5 and 10 followed by one between 10 and 15. This limited support sufficed for the networks we analyzed, but it can be extended to longer subpaths.

7 Evaluation

"P.S. WRT the prefix that was dual assigned from yesterday, one of my NOC [network operations center] guys stopped by today to ask what voodoo I was using to find such things :)"

– email from the head of the Net1 NOC

To evaluate Batfish, we used it on the configuration of two large university networks with disparate designs. We call them Net1 and Net2 in this paper as the operators requested anonymity. We aim to ascertain whether Batfish can scale to handle such real-world networks and whether it can find configuration errors in them.

7.1 Analyzed Networks

We analyzed recent network configurations from Net1 and Net2. They were working, stable configurations for which the operators were unaware of any bugs.

Net1 The routing design of Net1 uses BGP internally, modeling academic departments and a few other organizational entities (e.g., libraries, dorms) as ASes. The campus core network consists of 21 routers in 3 tiers: 3 border routers, 5 core routers, and 13 distribution routers. All routers run OSPF for internal connectivity. The border routers have eBGP peering sessions with two provider ASes and iBGP peering sessions with the core routers. The distribution routers have eBGP peering sessions with 52 internal ASes which are treated as customers of the core network. By design, each department AS is expected to have redundant peering connections with the Net1 core network, and each department should have its own distinct address space. Distribution routers also have iBGP peering sessions with the core routers.

As mentioned earlier, the environment for analysis of a network includes the route announcements from neighboring ASes. We used a single set of route announcements for all of the experiments on Net1. These route announcements were defined by creating stub configuration files for a new set of routers that represent Net1's BGP peers; this has the effect of populating the appropriate relations of our control plane model in Stage 1. The provider AS routers were simply configured to advertise a default route (i.e., the AS is willing to carry any traffic). The department AS routers were configured to advertise every network that their Net1 peer would accept but drop all traffic that was not destined to their own delegated address space. This approach ensures that we do not assume department ASes are "well behaved" when checking for vulnerabilities in Net1. Including these new routers, the topology we analyzed has 75 nodes.

Net2 The routing design of Net2 is qualitatively different. It employs VLANs to model the network as a large layer-2 domain. The network consists of 17 routers, of which three are core routers on the main campus and the rest interconnect the main campus with satellite campuses. All routers run OSPF for internal connectivity.

Since Net2 does not use BGP internally, we did not model the network's neighbors explicitly, as was done for Net1. Rather, the environment we used contained no route announcements from neighbors, and the analyzed topology included just the original 17 nodes.

7.2 Experiments

We checked for each consistency property in §4.

Multipath Consistency This property was encoded as a logical formula described in §4. We posed one NoD query pair per source node in the network, which asks for the existence of a header exhibiting a multipath inconsis-

tency when injected at the given node. Whenever such a header was identified, it was fed into Stage 4 of Batfish, which produced provenance information that pointed to the source of the inconsistency in the original configurations. We then patched the configurations and iterated until all queries were unsatisfiable.

Failure Consistency For this experiment, we generated the data plane corresponding to no failures as well as one data plane for each possible failure of a single (non-generated) interface (199 for Net1, 279 for Net2). We used NoD to separately obtain constraints on packets that are accepted in the no-failure scenario and constraints on packets that are not accepted in each failure scenario, again with separate queries per each possible source node. Finally we asked Z3 to find a concrete header satisfying the constraints of both the no-failure scenario and the failure scenarios, for each possible failure scenario and each source node in the network.

Destination Consistency For Net1 we generated 53 separate data planes: one corresponding to the unchanged configurations and one corresponding to the removal of each of Net1's 52 customer ASes. We excluded the provider ASes from this analysis altogether, since in general a provider may appear to provide an alternate path to any prefix that is part of a separate AS. We then used NoD and Z3 in the same way as described above for failure consistency, to identify headers that are accepted in the original data plane and also accepted after the destination's associated peer is removed from the network.

Destination consistency is not applicable to Net2, since it has no customer ASes.

7.3 Results

Batfish found a variety of bugs in both networks. Many of the concrete counterexamples it reported had different headers but were due to the same underlying configuration issue or an analogous issue on a different router. This makes counting the number of distinct issues somewhat difficult, so we provide two different metrics. First, we count one *bug* for each inconsistency related to an explicitly declared space of packet headers or source IPs in the network configuration. Second, we group bugs of a similar nature into *bug classes*. For instance, if a prefix list is incorrectly defined in two routers, we may find two unique bugs but we consider them to be in the same class. In general, the relationship between bugs and bug classes is complex: a change to a network configuration may remove one, two, or more bugs from the same class.

Table 1 summarizes our results for both the number of

		Total violations	Undesired behaviors	Fixed violations
Net1	Multipath	32 (4)	32 (4)	21 (3)
	Failure	16 (7)	3 (2)	0 (0)
	Destination	55 (6)	55 (6)	1 (1)
Net2	Multipath	11 (3)	11 (3)	11 (3)
	Failure	77(26)	18(7)	0(0)

Table 1: *Number of bugs (bug classes) for each property.*

bugs and bug classes (in parenthesis). We reported each property violation with its provenance information to the operators. The "Total violations" column gives the number of bugs and bug classes we reported. The "Undesired behaviors" column contains the subset of total violations that the operators confirmed caused undesired behaviors in the network. The only difference in these columns occurs for failure consistency. As explained below, this difference is not due to false positives in the analysis but reflects an intentional lack of fault tolerance in portions of the network or lack of fidelity in modeling network topology. "Fixed violations" is the subset of undesired violations that were fixed after we reported them. Not all behaviors that were confirmed as undesired by operators could be immediately fixed because the change was complex or the operators feared collateral damage.

Finally, a fix to a configuration may eliminate violations of multiple consistency properties. For instance, we see cases in which a fix that the operator applied for multipath consistency also removed some violations of failure consistency. In Table 1, we count such violations only once (for the property listed highest).

7.3.1 Understanding the discovered bugs

We now provide insight into issues that were uncovered by Batfish.

Multipath Consistency For Net1, a serious issue we found was a typo in the name of a prefix list on a Cisco router used to filter advertisements from one of the departments. The semantics for an undefined prefix list are to accept all advertisements. We found that this bug allowed the department to partially divert all traffic destined to other Net1 departments, as well as all traffic destined to arbitrary Internet addresses from any department. This bug was confirmed and fixed by the operators.

We show a sample of the provenance information for this bug below for a single hijacked prefix:

```
FlowAccepted(Flow<srcNode=nS, dstIp=10.0.0.1>, nV)
FlowDeniedIn(Flow<srcNode=nS, dstIp=10.0.0.1>, nA,
   Ethernet0, filter, 4)
FlowMatchRoute(Flow<srcNode=nS, dstIp=10.0.0.1>, nS,
```

```
   Route<prefix=10.0.0.0/24, nextHop=nA, ibgp>)
FlowMatchRoute(Flow<srcNode=nS, dstIp=10.0.0.1>, nS,
   Route<prefix=10.0.0.0/24, nextHop=nV, ibgp>)
```

Here, *nA* is an adversarial department that can send arbitrary advertisements, and *nV* is a victim department whose network has been hijacked. This indicates that some source node *nS* has iBGP routes to the victim's prefix through either the victim or the adversary. This would not be possible if advertisements from the adversary were filtered properly.

We also discovered three bug classes in which ACLs intended for the same department on different routers did not match. Two of these bug classes were fixed. The third bug class was confirmed as a problem, but the operators did not immediately fix it. The network operator stated that the ACLs in those cases matched the prefixes the peer wanted to announce at each connection point. He further commented, however, that should the peer change where these prefixes are announced without notice, traffic could get dropped. Therefore, he decided to change the policy in the future to accept all peer-delegated prefixes at each connection point, leaving it to the peer to decide what gets announced where.

For Net2, all of the multipath consistency bugs were due to inconsistent handling of routes redistributed into OSPF. In some cases, a connected route and a null static route (one configured to drop traffic for a prefix) for the same prefix would be redistributed by two different routers, and both of these routers would be installed as next hops for this prefix by a third adjacent router (§2.2). In the other cases, two routers would redistribute connected routes to a link they shared, but the path through one router would allow some traffic while the path through the other might deny it due to the ACLs applied on that path.

Failure Consistency For Net1, all of the failure consistency violations that Batfish found occurred when the interface that connected a department peer was failed. This situation indicates that the peer's only connection to the core network was through the interface disabled for the experimental run, which implies an absence of fault tolerance. The network operator reported that several such cases were known and due to economic reasons. The peer could not afford to maintain multiple links, or laying another line would be prohibitively expensive. We did not count these cases as "Undesired behaviors."

For Net2 Batfish found 26 bug classes for failure consistency, as shown in Table 1. But 19 were not deemed as undesired behaviors by the network operator. 5 were due to a bad assumption in how we currently model VLANs. We assume one-to-one mapping between logical VLAN interfaces and physical interfaces, but in reality the relationship was one-to-many for some VLANs, which led Batfish to underestimate fault tolerance. 14 bug classes represented intentional absence of fault tolerance. In 6 of them, providing backup paths was deemed prohibitively expensive. Interestingly, in 8 cases, backup paths existed but certain types of traffic were not allowed to use it.

Batfish found 7 bug classes that represented unexpected lack of fault tolerance. In 5 cases, it was due to VLAN implementation using a single physical interface. In the rest, only a single link served certain paths, which surprised the operators. These inconsistencies could not be fixed immediately because the solution needed new hardware and links in addition to configuration changes.

Destination Consistency For Net1, we found one bug (class) which the operators fixed: advertisements for a particular prefix were erroneously permitted from both the dorms and an academic department. This situation allowed the dorms to hijack the department's traffic.

The other discovered cases of destination consistency were confirmed by the operator as undesirable but were also known. These were cases in which advertisements for a prefix were permitted from several peers, but these peers actually fell under one administrative unit; they were separated into multiple ASes because of legacy considerations, and/or an unwillingness on the part of the peer operators to disturb a working system. The operator noted that ideally they would all fall under a single AS and wants to start consolidating them. Thus, the discovered violations represent fragility in the face of changes on the other end, but should not disrupt traffic as is.

7.4 Performance benchmarks

The time to analyze a network using Batfish depends on the size and complexity of the network and the correctness properties checked, as well as the performance of third-party tools such as NoD and LogicBlox. But we provide some insight by reporting on what we observed for our networks. We focus on the second and third stages of Batfish, as the other two stages take relatively little time (under a minute).

First consider multipath consistency. On an Intel E5-2698B VM, data plane generation (Stage 2) takes 238 (37) minutes for Net1 (Net2). Checking multipath consistency (Stage 3) requires making 75 (17) NoD and Z3 query pairs, each component of which takes under 90 seconds on a single core. Each query is completely independent of the others, so Batfish performs them in paral-

lel. A significant portion of the time to compute the data plane for Net1 is due to the large number of routes advertised by the generated department configurations; we believe this computation can be optimized significantly.

Failure consistency is the most onerous of our properties to check, since it requires one data plane per failure case of interest. There are 199 (279) such failure cases for Net1 (Net2); each can be checked independently. With an optimal number of processing nodes, i.e. 1 per dataplane, the computation time will not be appreciably more than that for multipath consistency.

Operators that have access to only modest hardware resources can use Batfish as follows. Before applying a configuration change, they can check for only multipath consistency and other properties that do not require additional data planes. This provides important correctness guarantees for the common case of no failures. Then, after applying the configuration change, the operators can continue to check for other properties in the background.

8 Related Work

Our work builds on several threads of prior work. One such thread is the static analysis of network configurations, which, as detailed in §1, has focused on specific aspects of the configuration or specific properties, enabling customized solutions [2, 7, 11, 24, 25, 34]. For instance, rcc [7] and IP Assure [24] perform a range of checks that pertain to particular protocols or configuration aspects (e.g., the two ends of an OSPF link are in the same area, link MTUs match, the two ends of an encrypted tunnel use the same type of encryption-decryption). While violations identified by such static analysis tools likely represent poor practices, the tools cannot, unlike Batfish, indicate whether or how violations impact the network's forwarding. On the other hand, for a violation that occurs only in specific environments (e.g., when certain kinds of external routes are injected in the network), Batfish can detect it only when given a concrete instance of one of these environments, but a specialized tool for checking particular properties may be able to uncover such a violation even without these concrete inputs by leveraging specific characteristics of those properties.

Closer to our work are approaches that directly model network behavior from its configuration. For example, Feamster et al. [8] develop a tool to compute the outcome of BGP route selection for an AS. Xie et al. [33] outline how to infer reachability sets, which are sets of packets that can be properly carried between a given source and destination node in the network. Benson et al. [4] extend this notion of reachability to assess the complexity of a network. Batfish is similar in spirit but broader in scope, handling all aspects of configuration that affect forwarding and producing a complete data plane.

The C-BGP [28] and Cariden [5] tools also generate a data plane from network configuration, but they use an imperative, simulation-based approach, and focus on specific configuration aspects (BGP and traffic engineering, respectively). We employ a declarative approach, which provides a way to tractably reason about all aspects of the configuration. More importantly, Batfish provides provenance information and the ability to query intermediate control plane relations.

Anteater [22] and Hassel [14] analyze data plane snapshots, obtained by pulling router FIBs and parsing portions of configuration that map directly to forwarding state (e.g., ACLs). More recent data plane analysis tools focus on SDNs and faster computations [13, 15, 20, 37]. By starting from the network configuration, Batfish can find forwarding problems proactively and enable "what if" analysis across different environments. However, data-plane snapshot analysis is not rendered expendable by our approach. Such analysis can find forwarding problems due to router software bugs, while we assume that the router faithfully implements the configurations. Thus, both types of analyses are valuable in the network verification toolkit.

Batfish employs NoD [20] to perform data-plane analysis in Stage 3 of its pipeline. We picked NoD because it had better performance and usability than prior tools. NoD has been used by its creators for "differential reachability" queries, one of which is analogous to our notion of multipath consistency. Their queries and our properties were developed independently.

9 Conclusions

We develop a new approach to analyze network configuration files that can flag a broad range of forwarding problems proactively, without requiring the configuration to be applied to the network. For two large university networks, our instantiation of the approach in the Batfish tool found many misconfigurations that were quickly fixed by the operators. Our approach is fully declarative and derives, from low-level network configurations, logical models of the network's control and data planes. We believe that these models are useful beyond finding configuration errors, for instance, to migrate a network toward high-level programming frameworks while faithfully preserving its existing policies.

Acknowledgments We thank the NSDI reviewers and our shepherd Nick Feamster for feedback on this paper, Nuno Lopes and Nikolaj Bjorner for help with NoD and Z3, Martin Bravenboer for help with LogicBlox, and the operators of the networks we analyzed for their assistance and feedback. This work is supported in part by the National Science Foundation award CNS-1161595.

References

[1] Batfish. `http://www.batfish.org`, February 26, 2015.

[2] E. Al-Shaer and H. Hamed. Discovery of Policy Anomalies in Distributed Firewalls. In *Proceedings of the Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, INFOCOM '04, New York, NY, USA, March 2004. IEEE.

[3] M. Anderson. Time Warner Cable Says Outages Largely Resolved. New York, NY, USA, August 2014. Associated Press.

[4] T. Benson, A. Akella, and D. Maltz. Unraveling the Complexity of Network Management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, Berkeley, CA, USA, April 2009. USENIX Association.

[5] Cariden Technologies, Inc. IGP Traffic Engineering Case Study, October 2002.

[6] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, New York, NY, USA, October 2001. ACM.

[7] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '05, Berkeley, CA, USA, May 2005. USENIX Association. Tool source code at `https://github.com/noise-lab/rcc/`.

[8] N. Feamster, J. Winick, and J. Rexford. A Model of BGP Routing for Network Engineering. In E. G. C. Jr., Z. Liu, and A. Merchant, editors, *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04, New York, NY, USA, June 2004. ACM.

[9] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, New York, NY, USA, September 2011. ACM.

[10] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and Emerging Applications: An Interactive Tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, New York, NY, USA, June 2011. ACM.

[11] K. Jayaraman, N. Bjorner, G. Outhred, and C. Kaufman. Automated Analysis and Debugging of Network Connectivity Policies. Technical Report MSR-TR-2014-102, Microsoft Research, July 2014.

[12] C. R. Kalmanek, S. Misra, and Y. R. Yang, editors. *Guide to Reliable Internet Services and Applications*. Springer, 2010.

[13] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, Berkeley, CA, USA, April 2013. USENIX Association.

[14] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, Berkeley, CA, USA, April 2012. USENIX Association.

[15] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, Berkeley, CA, USA, April 2013. USENIX Association.

[16] The LLVM compiler infrastructure. `http://llvm.org/`, February 26, 2015.

[17] LogicBlox, Inc. LogicBlox 4 Reference Manual. `https://developer.logicblox.com/content/docs4/core-reference/html/index.html`, February 26, 2015.

[18] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, New York, NY, USA, June 2006. ACM.

[19] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of the ACM SIGCOMM 2005 Conference*, SIGCOMM '05, New York, NY, USA, June 2005. ACM.

[20] N. P. Lopes, N. Bjorner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking Beliefs in Dynamic Networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, Berkeley, CA, USA, May 2015. USENIX Association.

[21] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP Misconfiguration. In *Proceedings of the ACM SIGCOMM 2002 Conference*, SIGCOMM '02, New York, NY, USA, August 2002. ACM.

[22] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, New York, NY, USA, August 2011. ACM.

[23] J. Moy. OSPF Version 2. RFC 2328, RFC Editor, April 1998.

[24] S. Narain, R. Talpade, and G. Levin. *Guide to Reliable Internet Services and Applications*, chapter Network Configuration Validation. In Kalmanek et al. [12], 2010.

[25] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave Tool for Firewall Analysis. In *Proceedings of the 24th Large Installation System Administration Conference*, LISA '10, Berkeley, CA, USA, November 2010. USENIX Association.

[26] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, Berkeley, CA, USA, April 2014. USENIX Association.

[27] T. J. Parr and R. W. Quong. ANTLR: A Predicated-LL (k) Parser Generator. *Software: Practice and Experience*, 25(7), 1995.

[28] B. Quotin and S. Uhlig. Modeling the Routing of an Autonomous System with C-BGP. *IEEE Network: The Magazine of Global Internetworking*, 19(6), 2005.

[29] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, RFC Editor, January 2006.

[30] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031, RFC Editor, January 2001.

[31] S. Shenker. The Future of Networking, and the Past of Protocols. Open Networking Summit, April 2012.

[32] O. Tange. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine*, 36(1), February 2011.

[33] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On Static Reachability Analysis of IP Networks. In *Proceedings of the Twenty-fourth Annual Joint Conference of the IEEE Communications Society*, volume 3 of *INFOCOM '05*, New York, NY, USA, March 2005. IEEE.

[34] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra. FIREMAN: A Toolkit for Firewall Modeling and Analysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, New York, NY, USA, May 2006. IEEE.

[35] Z3 theorem prover. `https://z3.codeplex.com/` (opt branch), February 26, 2015.

[36] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. A Survey on Network Troubleshooting. Stanford HPNG Technical Report TR12-HPNG-061012, Stanford University, June 2012.

[37] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, Berkeley, CA, USA, April 2014. USENIX Association.