A General Approach to Network Configuration Verification

Ryan Beckett
Princeton University

Aarti Gupta
Princeton University

Ratul Mahajan
Microsoft Research & Intentionet

David Walker
Princeton University

ABSTRACT

We present Minesweeper, a tool to verify that a network satisfies a wide range of intended properties such as reachability or isolation among nodes, waypointing, black holes, bounded path length, load-balancing, functional equivalence of two routers, and fault-tolerance. Minesweeper translates network configuration files into a logical formula that captures the stable states to which the network forwarding will converge as a result of interactions between routing protocols such as OSPF, BGP and static routes. It then combines the formula with constraints that describe the intended property. If the combined formula is satisfiable, there exists a stable state of the network in which the property does not hold. Otherwise, no stable state (if any) violates the property. We used Minesweeper to check four properties of 152 real networks from a large cloud provider. We found 120 violations, some of which are potentially serious security vulnerabilities. We also evaluated Minesweeper on synthetic benchmarks, and found that it can verify rich properties for networks with hundreds of routers in under five minutes. This performance is due to a suite of model-slicing and hoisting optimizations that we developed, which reduce runtime by over 460x for large networks.

CCS CONCEPTS

• **Networks** → *Network reliability*;

KEYWORDS

Network verification; Control plane analysis

ACM Reference format:
Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21-25, 2017,* 14 pages.
DOI: 10.1145/3098822.3098834

1 INTRODUCTION

The control plane of traditional (non-SDN) networks is a complex distributed system. Network devices use one or more protocols to exchange information about topology and paths to various destinations. How they process this information and select paths to use for traffic depends on their local configuration files. These files tend to have thousands of lines of low-level directives, which makes it hard for humans to reason about them and even harder to reason about the network behavior that emerges through their interactions.

As a result, configuration errors that lead to costly outages are all-too-common. Indeed, every few months configuration-induced outages at major networks make the news [1, 5, 29, 32]. Systematic surveys also show that configuration error is the biggest contributor to such network outages [20, 26].

To address this problem, researchers have developed many tools for finding errors in network configurations. We broadly classify these tools along two dimensions: *i*) *network design coverage*—types of network topologies, routing protocols and other features the tool supports; and *ii*) *data plane coverage*—how many (or how much) of the possible data planes the tool can analyze. The network control plane dynamically generates different data planes as its environment (*i.e.*, up/down status of links and routing announcements received from external neighbors) changes. Tools with higher data plane coverage can analyze more such data planes.

Some of the earliest network diagnostic tools such as *traceroute* and *ping* can help find configuration errors by analyzing whether and how a given packet reaches its destination. These tools are simple and have high network design coverage—they can analyze forwarding for any network topology or routing protocol. But they have poor data plane coverage—for each run, they analyze the forwarding behavior for only a single packet for the data plane that is currently installed in the network.

A more recent class of *data plane analysis* tools such as HSA [18] and Veriflow [19] have better data plane coverage. They can analyze reachability for all packets between two machines, rather than just one packet. However, the data plane coverage of such tools is still far less than ideal because they analyze *only* the data plane that is currently installed in the network. That is, they can only find errors *after* the network has produced the erroneous data plane.

Control plane analysis tools such as Batfish [13] can find configuration errors proactively, before deploying potentially buggy configurations. Batfish takes the network configuration (*i.e.*, its control plane) and a specific environment (*e.g.*, a link-failure scenario) as input and analyzes the resulting data plane. This ability allows operators to go beyond the current data plane and analyze future data planes that may arise under different environments. Still, each run of Batfish allows users to explore at most one data plane, and given the large number of possible environments, it is intractable to guarantee correctness for all possible data planes.

Most recently, several control plane analysis tools have gone from *testing* individual data planes to *verification*—that is, reasoning about many or all data planes that can be generated by the control plane. However, each such tool trades network design coverage for

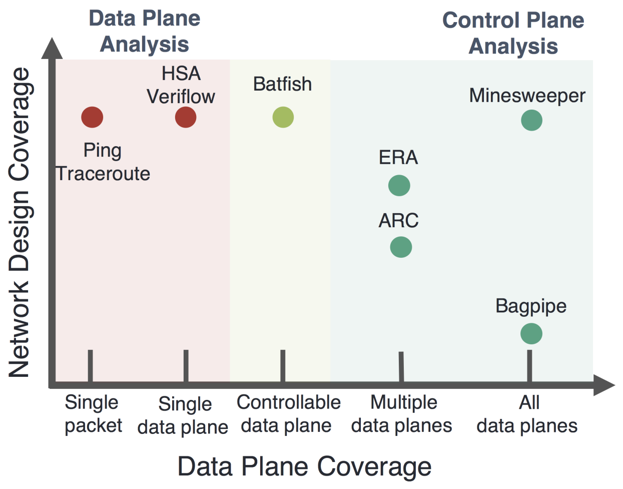

Figure 1: Landscape of network analysis tools.

higher data plane coverage. For instance, while Bagpipe [34] can symbolically simulate the message-passing semantics of BGP in all possible environments, it assumes that the network is a single autonomous system (AS) connected in an iBGP full mesh, and does not model any internal routing. Another tool, ARC [14], translates configurations to a weighted graph where the weighted-shortest paths capture the network forwarding behavior. A single run of ARC can efficiently analyze multiple data planes by considering the consequences of all possible failures but not all possible sets of external routing messages. Further, many networks, such as those using iBGP or using certain features such as BGP local preference can not be reduced to simple weighted graphs. ERA [11] compactly represents a concrete set of control plane messages using binary decision diagrams (BDDs) and propagates this set along a path through the network by transforming the set as dictated by the network configuration. In this way, ERA can efficiently check reachability in certain large symbolic environments (*e.g.*, the environment with all possible eBGP advertisements), but using ERA to verify configurations in the face of all environments is an open problem [11]. Further, the path-based approach of ERA cannot faithfully analyze reachability for certain networks such as those running iBGP.

In summary then, while there has been great progress toward analyzing network configurations, a fundamental scientific question is still open:

> *Is it possible to build a verification tool that achieves both high network design coverage and high data plane coverage while remaining scalable enough to enable verification of many real networks?*

We answer this question in the affirmative by developing a configuration verification tool called Minesweeper. Figure 1 situates Minesweeper and prior tools with respect to network design and data plane coverage. Minesweeper has both high network design coverage in that it works for a large collection of network protocols, features and topologies as well as high data plane coverage in that it can verify a large number of properties for all possible data planes that might emerge from the control plane.

The main challenge in developing Minesweeper was scaling such a general tool. We addressed it by combining the following ideas from networking and verification literature:

Graphs (not paths). Most existing tools reason about individual network paths. While this approach has proven effective for stateless data plane analysis (*e.g.*, HSA [18]), it creates substantial problems for control plane analysis. The distinction is that, in stateless data planes, packets on one path never interfere with those on a different path; but in the control plane, two route announcements can interfere. A routing message along one path may be less preferred than a message over another path, causing it to be dropped when the other message is present. For accuracy, interactions along all paths must be modeled, but there can be an intractably large number of paths. We avoid this problem by using a graph-based model, where rich logical constraints on its edges and nodes encode all possible interactions of route messages.

In addition to its better accuracy, our model can verify a much richer set of properties, expressed over graphs, rather than over paths alone. For example, it can reason about equivalence of routers, load balancing, disjointedness of routing paths, and if multiple paths to the same destination have equal lengths. Such properties are difficult or impossible for path-based models to check, and we show that they are valuable in finding bugs in real configurations.

Combinational search (not message set computation). Existing tools that analyze multiple environments [11, 34] eagerly compute the sets of routing messages that can reach various points in the network. However, these full sets are not typically needed and computing them is expensive. Fortunately, the symbolic model checking community has encountered this type of problem before. Rather than iteratively computing sets of messages, one can instead ask for a satisfying assignment to a logical formula that represents all possible message interactions. Suppose a variable $x_{m,l}$ represents whether a message m reaches a location l in the network and N encodes the network semantics logically. If there exists a satisfying assignment to the formula $N \wedge x_{m,l}{=}true$, then m can reach l and all the constraints N imposed by interacting messages are also satisfied. The advantage of this formula-based approach is that while model checking with message set computation is PSPACE-complete [7, 30], the search for a satisfying assignment in the related bounded model checking problem [6] is NP-complete. The intuition behind lower complexity is that searching for a satisfying assignment avoids computing many intermediate message sets. In practice too, modern SAT [23] and SMT (Satisfiability Modulo Theories) [9] solvers routinely solve large instances of such combinational search problems in hardware and software verification.

Stable paths problem. To realize an approach based on graphs and combinational search, we need to convert the distributed message-passing of the control plane into an equivalent logical formula. Here, we turn to the work of Griffin *et al.* [16], who showed that network control planes (BGP in particular) solve the *stable paths problem*, and these paths can be described by constraints on edges. Consequently, rather than encoding message exchanges, we can encode the corresponding set of edge constraints in our formula, such that satisfiable assignments correspond to stable paths in the control plane. Our formula captures all possible environments as

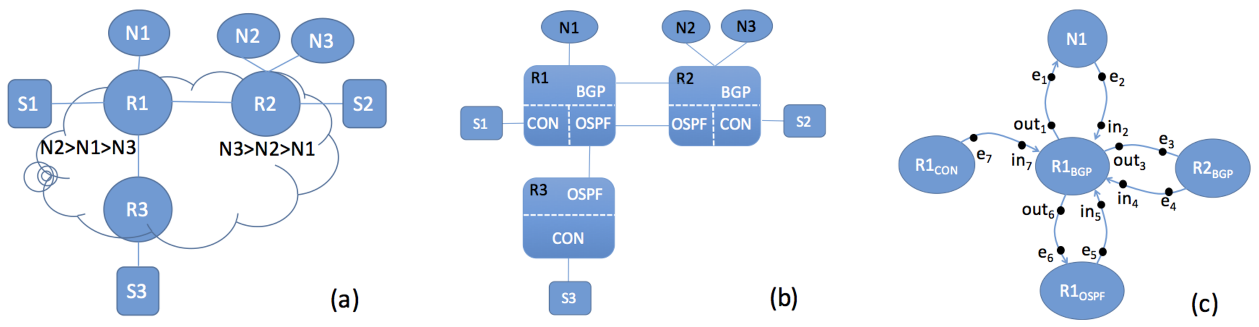

Figure 2: (a) An example network. (b) Its protocol-level decomposition. (c) Routing information flow for BGP at R1.

symbolic variables, and we add constraints related to properties of interest to perform verification.

Slicing and hoisting optimizations. Our default encoding of the network control plane produces large formulas that cannot be solved quickly for real networks. We have designed a range of highly effective optimizations that reduce the number of variables and constraints in our generated formulae enormously. One class of optimizations is *slicing*, which analyzes the formula to remove variables and constraints that cannot affect the final outcome. A second class of optimizations is *hoisting*, which lifts repeated computations out of their logical context and precomputes them once. Intuitively, such optimizations are effective because real networks have simpler control planes than the theoretical worst case. For instance, in theory, messages can be arbitrarily modified when sent to neighbors (implying the need for different variables for messages to different neighbors), but in practice the same message is sent to multiple neighbors (allowing shared variables). Similarly, while different routers may have arbitrarily different control plane logic in theory, in practice many routers share parts of their configurations.

We implement the concepts above in Minesweeper, and apply it to many real and synthetic networks. Across the 152 small- and medium-sized networks that we analyzed for four properties, we found 120 violations. One class of violations poses a serious security threat: the management interface IP of the routers could be "hijacked" by external neighbors by sending specific routing announcements. Our experiments with synthetic networks show that Minesweeper can verify rich properties such as many-to-one reachability, bounded path length, and device equivalence in under 5 minutes on networks with 100s of routers. Our optimizations are key to this performance. They help reduce verification time by a factor of up to 460x for large networks.

2 MOTIVATION

Our approach represents two significant departures from existing work on configuration analysis: *i*) using constraint-based graphs, instead of source-destination paths; and *ii*) using combinational search, instead of eagerly computing message sets. This section provides intuition behind these choices.

2.1 Paths vs. graphs

Consider the network in Figure 2(a). It has three internal routers, R1 to R3, that run OSPF. It connects to three external neighbors, N1 to N3, via BGP. The internal routers are connected to subnets, S1 to S3, whose address prefixes they redistribute into OSPF and BGP. R1 and R2 connect via iBGP, to share the BGP routes they hear from N[1..3]. They also redistribute BGP destinations into OSPF, so that R3 can reach those destinations, and OSPF into BGP so that internal subnets are announced externally. The BGP preferences of R1 and R2 are as shown: R1 (R2) prefers routes through N2, N1, and N3 (N3, N2, N1) in that order. Recall that in BGP, when multiple routes are available to the same destination, a router will select and share the most preferred one according to the local configuration.

Suppose we want to ensure that the subnet S3 uses N1 to reach any external destination even when all three of N1, N2 and N3 announce a path to that destination. Does this property hold in our network? The correct answer is positive, but interestingly, the answer a configuration analysis tool delivers depends on the sophistication with which it reasons about the interactions of control plane messages on different paths.

- If the analysis only considers the path N1-R1-R3, it will conclude that the property holds. R1 will select the route through N1 since no other route is available and pass it to R3. Thus, R3 (and S3) will send traffic through N1. (Data flows in the opposite direction to routing information.)
- If the analysis additionally considers the routing path N2-R2-R1-R3 (which interferes with the first path at router R1), it will conclude that the property does *not* hold. R1 will select the route through N2 and thus the route through N1 will not reach R3.
- If the analysis also considers N3-R2-R1-R3 (which interferes with the second path at R2 and the first path at R1), it will conclude once again that the property holds. R2 will select the route through N3, and thus R1 will select and propagate to R3 the route through N1.

In the general case, all possible paths can interfere with one another, and for correct analysis, all mutual interactions should be considered. But the number of paths can be enormous: $O(V^{\frac{E}{V}})$, where V and E are the number of nodes and edges (and thus $\frac{E}{V}$ is the average node degree). Existing path-based tools circumvent this problem by restricting the networks they can analyze (*e.g.*, Bagpipe) or conducting a potentially unsound analysis (*e.g.*, ERA).

Our model avoids this problem by constructing a compact representation for all possible paths—a graph. The complexity of this structure is $O(V+E)$. Our graph accurately (and symbolically) models all interactions between different paths and supports a richer set of properties (described later).

2.2 Message sets vs. combinational search

One possible approach to control plane verification is to simulate all possible outcomes of the distributed control plane computation by computing (symbolic) sets of messages for all destinations. Once all outcomes of the control plane computation have been computed, one can analyze the complete set of possible final states and judge if the property of interest holds. Unfortunately, this approach often leads to a lot of unnecessary work.

In many cases, computing a full solution to the control plane computation is often unnecessary as the validity of the property may not depend upon many parts of that solution. In contrast, our approach encodes both the network and the property in question as a logical formula. As an SMT solver searches for a satisfying assignment to the formula, it will take the property into account. If the property does not require knowledge of some aspects of the control plane, the search process may ignore that part of the model. For example, if router R3 had an ACL that drops traffic sent to R1, then the solver might quickly learn that S3 will not be able to reach N1 without reasoning about the full control plane behavior. In §8, we show that many properties can be checked much more efficiently for this reason.

In addition, approaches that compute message sets represent and store *all* possible outcomes of the control plane's full fixed point computation and they find *all* violations of the property. In contrast, our approach *searches* for just *one* outcome of the control plane computation that violates the given property. The latter can be done extremely efficiently by modern SMT solvers in many domains. While our approach will not find *all* violations at once, finding just one violation can help pinpoint a bug. When that bug has been fixed, one can apply the procedure again.

3 THE BASIC NETWORK MODEL

Our goal is to enable network operators to verify the behavior of their network under any possible environment. To provide this capability, we model the network with respect to a packet as a function of its environment. Because the packet and the environment are symbolic, our model can verify the control and data plane behavior of the network relevant to *any* packet under *any* environment.

More specifically, we generate F, a system of SMT constraints defined as the conjunction of N, the behavior of the network, given the current configurations of all routers, and $\neg P$, a negated property of interest to the operator. Satisfying solutions for N correspond to stable forwarding paths in the network. Thus, *any* stable solution (even among multiple ones) that violates the property will be reported as a satisfying solution for F. However, if F is unsatisfiable, then either all stable paths satisfy the property, or the network has no stable paths for the destination(s) of interest.

This section describes the techniques we use to generate a basic network model N using Figure 2 as an example. We explain, in turn, how to model (1) a data packet, (2) the interactions between routing protocols, (3) the control plane information, (4) the import filters in router configurations, (5) the route selection process, (6) the export filters in configurations, and (7) the access control lists that apply to data packets. We end this section with (8) an example encoding of a property P. Throughout this section, we refer to Figure 3, which lists the main symbolic variables used in our generated formulae. §4 discusses extensions to the basic network model, and §5 discusses many additional properties.

Variable	Description	Rep.
Data plane		
dstIp	Packet destination IP addr	$[0, 2^{32})$
srcIp	Packet source IP addr	$[0, 2^{32})$
dstPort	Packet destination port	$[0, 2^{16})$
srcPort	Packet source port	$[0, 2^{16})$
protocol	Packet Protocol	$[0, 2^{8})$
Control plane		
prefix_r	Prefix for record r	$[0, 2^{32})$
length_r	Prefix length for r	$[0, 2^{5})$
ad_r	Administrative distance for r	$[0, 2^{8})$
lp_r	BGP local preference for r	$[0, 2^{32})$
metric_r	Protocol metric for r	$[0, 2^{16})$
med_r	BGP MED attribute for r	$[0, 2^{32})$
rid_r	Neighbor router ID for r	$[0, 2^{32})$
bgpInternal_r	Was r learned via iBGP	1 bit
valid_r	Is the record r valid	1 bit
Decision		
$\text{controlfwd}_{x,y}$	x fwds to y (ignores ACLs)	1 bit
$\text{datafwd}_{x,y}$	x fwds to y (includes ACLs)	1 bit
Topology		
$\text{failed}_{x,y}$	Is the link from x to y failed	$[0, 1]$

Figure 3: Selected symbolic variables from the model

(1) Modeling data plane packets. The first section of Figure 3 lists several of the variables used to represent a symbolic data packet. The packet's destination IP is modeled by an integer variable dstIP, which ranges from 0 to $2^{32}-1$. We model other fields similarly. If operators wish to ask about a specific destination, such as 10.0.0.0, they may issue a query that constrains our model to consider only packets with that destination (*e.g.*, using the formula dstIP = 10.0.0.0 in their property P). If they instead wish to ask about packets with *any* destination IP, they may leave the dstIP field unconstrained. Traditional (non-SDN) networks do not typically modify packet headers[1]—they only forward or block them. Consequently, we use only one, global copy of each of these variables in our formula.

In order to determine what happens to such packets in the network, we must, of course, model the control plane protocols and how they decide to forward packets.

(2) Modeling protocol interactions. Routers commonly run multiple protocol instances, each of which operates independently and selects a best route for a destination prefix based on the information from its remote peers and redistribution from other, local routing instances. Figure 2(b) presents a protocol-level view of the internal

[1]Except for TTL and CRC fields, which we do not currently model.

routers in our example. Routers R1 and R2 run BGP to exchange routes with the outside world and OSPF to communicate internally. CON denotes connected routes, *i.e.*, those known through a directly connected interface. We model them as if they are another protocol to avoid special cases.

Figure 2(c) zooms into R1's BGP instance. Each node is a protocol instance and each edge represents information flow between two instances. For example, the nodes $R1_{OSPF}$ and $R1_{BGP}$ represent protocols OSPF and BGP on router R1. Since OSPF redistributes into BGP, and vice versa, there are edges back and forth between $R1_{OSPF}$ and $R1_{BGP}$. The outgoing edge from $R1_{CON}$ indicates that the connected routes are redistributed into BGP. Since R1 uses BGP with the external neighbor N1 and R2, there are edges in both directions between $R1_{BGP}$ and N1 and $R2_{BGP}$.

(3) Encoding control plane information. To model the control plane, we need to encode the information in the messages exchanged by protocol instances. We do so using records of symbolic values, which roughly correspond to protocol messages. As with the data packets, constraints may map these variables to specific concrete values (*e.g.*, the prefix 10.1.0.0/24) or may leave them fully or partially unconstrained.

Unlike the single symbolic data packet, there are *many* control plane records in our encoding. The edge labels in Figure 2(c) indicate the presence of a specific record. Consider the edge between $R2_{BGP}$ and $R1_{BGP}$. The label e_4 represents the message exported by R2's BGP process on the link to R1; and the label in_4 represents the message after traversing R1's BGP import filter on the link from R2. Naturally, the messages defined by in_4 and e_4 are closely related. We encode the relationship using SMT constraints generated from import filters in R1's configuration.

Routing messages from the environment are represented as records from an external neighbor. For example, the record e_2 is the export from neighbor N1. When left unconstrained, it represents the fact that N1 could send any message.

The second section of Figure 3 lists the main fields of symbolic control plane records. Each record is for a destination prefix of a particular length. Announcements for that prefix are annotated with the administrative distance (ad). When multiple protocol instances offer a route to the same prefix, this measure (which is configured for each protocol) determines which one is used for forwarding. These records also contain the local preference (lp) for BGP, and the metric. The metric is a protocol-specific measure of the quality of the route. For instance, it is path length for BGP and path cost for OSPF. When routes are redistributed from one protocol to another, the configuration determines the initial metric that the router will use. The router id (rid) is used to break ties among equally-good routes. Other protocol-specific attributes such as the BGP multi-exit discriminator (med), and whether a BGP route was learned via iBGP (bgpInternal), are included in each such symbolic record. Finally, every record contains one special boolean field, called valid. If valid is true, then a message is present and the remaining contents of the record are meaningful; otherwise, they are not meaningful (*e.g.*, no message arrives at this location).

Because we are interested in the behavior with respect to a single symbolic packet, we only want to consider control plane messages for prefixes that impact this packet. The valid field of a control plane record will be true if and only if *i*) a message is present (*e.g.*, advertised from a neighbor and not filtered), and *ii*) the *control plane* destination prefix applies to the *data plane* destination IP of the packet of interest. We capture the latter dependence with the following constraint:

$$e.\text{valid} \implies \text{FBM}(e.\text{prefix}, \text{dstIP}, e.\text{length})$$

The function FBM (first bits match) tests for equality of the first *e*.length bits of the prefix (*e*.prefix) and destination IP, thus capturing the semantics of prefix-based forwarding.[2]

```
if $e_4$.valid $\wedge$ $\text{failed}_{R1,R2} = 0$ then
    if $\neg$ (FBM($e_4$.prefix, 192.168.0.0, 16) $\wedge$
        $16 \leq e_4.\text{length} \leq 32$)
    then
        $in_4$.valid = true
        $in_4$.lp = 120
        $in_4$.ad = $e_4$.ad
        $in_4$.prefix = $e_4$.prefix
        $in_4$.length = $e_4$.length
        $in_4$.bgpInternal = true
        ...
    else $in_4$.valid = false
else $in_4$.valid = false
```

Figure 4: Translation of the R1 to R2 BGP import filter

(4) Encoding import filters. Each router configuration defines (possibly per neighbor) filters that can either drop or modify protocol messages. As an example, consider the following configuration fragment for router R1.

```
ip prefix_list L deny 192.168.0.0/16 le 32
ip prefix_list L allow

route-map M 10
  match ip address prefix-list L
  set local-preference 120
```

This fragment blocks control plane announcements for any prefix that matches the first 16 bits of 192.168.0.0, and has prefix length between 16 and 32. It sets the local preference attribute to 120 for any other prefix. Assuming R1's BGP process is configured with this fragment as an import filter, we use it to constrain the relationship between the symbolic records e_4 and in_4 in Figure 2(c). More specifically, the filter is realized by the formula shown in Figure 4. The first line in this formula ensures that there can be an advertisement at in_4 only if R2 exports an advertisement to e_4 and the R1–R2 link is not failed. The second condition implements the import filter. If the two conditions are met, then information from R2 will arrive at R1. Hence, we set the valid bit of in_4, constrain the local preference to 120, and constrains in_4's other fields to be the same as e_4's. In all other cases, no advertisement arrives at R1, so its valid bit is set to false.

[2]The constraint FBM(p_1,p_2,n) is surprisingly tricky to encode efficiently. A naive solution that represents p_1 and p_2 as bit-vectors of size 32 is slow. See §6 for an efficient solution.

```
**if** $\text{best}_{\text{BGP}}.\text{valid} \land \text{failed}_{\text{R1,R2}} = 0$ **then**
    **if** $\neg\,\text{best}_{\text{BGP}}.\text{bgpInternal} \land \text{best}_{\text{BGP}}.\text{length} + 1 \leq 255$
    **then**
        $\text{out}_3.\text{valid} = \text{true}$
        $\text{out}_3.\text{lp} = \text{best}_{\text{BGP}}.\text{lp}$
        $\text{out}_3.\text{ad} = \text{best}_{\text{BGP}}.\text{ad}$
        $\text{out}_3.\text{prefix} = \text{best}_{\text{BGP}}.\text{prefix}$
        $\text{out}_3.\text{length} = \text{best}_{\text{BGP}}.\text{length} + 1$
        ...
    **else** $\text{out}_3.\text{valid} = \text{false}$
**else** $\text{out}_3.\text{valid} = \text{false}$
```

Figure 5: Translation of the R1 to R2 BGP export filter

Such translation of import filters to symbolic constraints can also capture route redistribution between protocols. Users can set custom metric and administrative distance values for route redistribution, which would be updated as before.

(5) Encoding route selection. Each protocol instance selects a best route for each IP prefix among those available.[3] For example, the routes available to R1_{BGP} include routes from its neighbors and thus defined by the status of the symbolic records in_2, in_4, in_5, and in_7. The available routes are ordered by the decision process in a standard way. For instance, BGP first prefers the route with the highest administrative distance, and if those are equal, the highest local preference, then highest metric, *etc.* We implement this ordering via a relation $r_1 \preceq r_2$, which may be read as "r_1 is at least as preferred as r_2." The selected route is the one that is both available (the valid bit is set) and highest in the ordering. Logically, our encoding introduces a new symbolic record $\text{best}_{\text{prot}}$ for each protocol instance prot. Each such record is equated with the highest available route in the order. For instance, for R1_{BGP}, if no input in_i is valid then best_{BGP} is not valid. Otherwise:

$$\bigwedge_{i \in \{2,4,5,7\}} \text{best}_{\text{BGP}} \preceq \text{in}_\text{i} \quad \land \quad \bigvee_{i \in \{2,4,5,7\}} \text{best}_{\text{BGP}} = \text{in}_\text{i}$$

This constraint states that best record is less than or equal to all alternatives and equal to at least one of them.

Each router installs only one route in its data plane, which is then used to forward traffic. Thus, it chooses a best route among all routing protocols. Once again, this can be modeled with a new symbolic record $\text{best}_{\text{overall}}$, which is similarly constrained to be the best among all the $\text{best}_{\text{prot}}$ records.

To represent the final forwarding decision of the router, we introduce a new boolean variable $\text{controlfwd}_{\text{x,y}}$ for each edge in the network between routers x and y. The variable indicates that router x decides to forward traffic for the destination to router y. Intuitively, router x will decide to forward to router y if the message received from y is equal to the best choice. For example, to determine if R1 will forward to R2, we use the following constraint:

$$\text{controlfwd}_{\text{R1,R2}} = (\text{e}_4.\text{valid} \land \text{e}_4 = \text{best}_{\text{overall}})$$

[3]Assume for now that a single best route is selected. We outline the extension for multipath routing in §4.

(6) Encoding route export. After selecting a best route, each protocol will export messages to each of its peers after potentially processing these messages through peer-specific export filters. Figure 5 shows the route export constraint for R1_{BGP}'s export to R2_{BGP} assuming the default export filter. The encoding of route export is similar to that of an import filter, but with some differences. First, the export constraint will connect the record for the protocol's best route (best_{BGP}) with a record on an outgoing edge of a router (e.g., out_3). Second, the route export constraint accounts for the fact that iBGP routes should not be re-exported to other iBGP peers by checking if the best route was learned via iBGP. Third, the path metric is updated according to the protocol (e.g., adding 1 for BGP). Finally, the route is only exported if the new path metric does not overflow the maximum protocol path length (e.g., 255 for BGP).

(7) Encoding data plane constraints. Although routers decide how to forward packets in the control plane through their decision process, the actual data plane forwarding behavior can differ due to the presence of an access control list (ACL), which lets a router block traffic directly in the data plane. To handle ACLs, we create additional variables to represent the final data plane forwarding behavior of the network. For each variable $\text{controlfwd}_{\text{x,y}}$, we create a corresponding $\text{datafwd}_{\text{x,y}}$ variable. The data plane forwarding will be the same as the control plane forwarding modulo any ACLs. For example, consider the following ACL:

```
access-list 1 deny ip  172.10.1.0  0.0.0.255
```

The mask 0.0.0.255 signifies the wildcard bits for the match. This ACL will thus block any packets that match destination IP 172.10.1.* in the data plane. This constraint is captured by first translating the ACL to a formula and then conjoining it with the control plane decision in the following way:

$$\text{datafwd}_{\text{R1,R2}} = \text{controlfwd}_{\text{R1,R2}} \land \neg\,\text{FBM}(\text{dstIP}, 172.10.1.0, 24)$$

(8) Encoding properties. While the model above captures the joint impact of all possible network interactions, to verify properties of interest we can instrument it with additional variables as needed. For example, suppose we wish to check that router R3 can reach N1 regardless of any advertisements received from neighbors N2 and N3. For each router x in the network, we add a variable reach_x representing that x can reach the destination subnet. For R1, which is directly connected to N1, we add:

$$\text{canReach}_{\text{R1}} \iff \text{datafwd}_{\text{R1,N1}}$$

For every other router, we say it can reach N1 if it can forward to some neighbor that can reach N1. For router R3:

$$\text{canReach}_{\text{R3}} \iff \bigvee_{R \in \{R1\}} (\text{datafwd}_{\text{R3,R}} \land \text{canReach}_{\text{R}})$$

Since we are interested in checking that the property holds for any possible packet, we leave the packet fields (e.g., dstIp) unconstrained. Finally, we would assert the negation of the property we are interested in, namely $\neg\text{canReach}_{\text{R3}}$ and ask the solver to prove unsatisfiability, thereby ensuring that the property holds for all packets and environments.

4 GENERALIZING THE MODEL

This section describes how we encode several additional features of network configurations.

Link-state protocols. In link-state protocols, such as OSPF and ISIS, routers share information about the cost and state (up or down) of each link. Each router then builds a global view of the network and computes the least-cost path to each destination. These least-cost paths are a special case of stable paths. Each router along the shortest path will send traffic to a neighbor only if that neighbor has a path to the destination and no other neighbor offers a lower cost path. Based on this observation, we model link-state protocols the same way as path-vector protocols, using configured link costs.

Distance-vector protocols. Like link-state protocols, distance-vector protocols such as RIP also compute a shortest path tree. However, unlike link-state protocols, they do so without maintaining a global view of the network, instead passing information about path length to the destination between neighbors. We can model the solution to a distance vector protocol the same way as for link-state protocols but where each link has a weight of 1.

Static routes. Static routes are used to tell a router to always forward to a particular next hop IP address, or out a particular interface. As with connected routes, we model static routes as their own routing instance that makes forwarding decisions based on the destination IP address. By modeling static routes this way, we can treat them similarly to other protocols and easily model route redistribution where static routes are injected into other protocols.

Aggregation. Aggregation, in which routers announce a less-specific prefix that covers many, more-specific prefixes, helps reduce the size of the routing tables. We model aggregation as a modification to the prefix length attribute. If a prefix is valid for the destination IP address before aggregation, it remains valid after aggregation, but with a shorter prefix length. For example, if a /24 prefix is relevant for the packet's destination IP then so is its aggregated /16 prefix.

Multipath routing. The encoding in §3 assumed that routers select a single best path, but multipath routing, where traffic is spread over multiple equally-good routes to balance load, is common in modern networks. To encode multipath routing, we relax the best route comparison so that it does not compare the router ID. This relaxation no longer requires a total ordering of preferred routes, and any route as good as the best route will be used.

BGP communities. BGP communities are strings that can be attached to (or removed from) route advertisements. We model communities using a new variable $\text{community}_{\text{x,c}}$ for each router x and community c that appears in some router's configuration. Vendors allow community values to be added or removed arbitrarily by any router. We encode the semantics of these transformations simply by updating the value of $\text{community}_{x,c}$ according to the import/export filters at the router.

iBGP. Modeling iBGP is challenging because it introduces cross-destination dependencies through recursive lookup. In order to determine the forwarding behavior for a particular packet p over a network using iBGP, one first has to determine the forwarding behavior for each user-defined next-hop destination IP address configured between iBGP peers. For example, if router A has no IGP route to router B's iBGP-configured next-hop IP address, then the peers can not exchange BGP advertisements about packet p.

To model iBGP, we create N additional copies of the network where N is the number of routers configured to run iBGP. Each copy of the network encodes the forwarding behavior for a packet destined to the next-hop IP address associated with one of the iBGP-configured routers. We add the constraint that router A only propagates routes to router B over an iBGP connection if A can reach B in the network copy corresponding to B's configured next-hop destination IP address.

The variable (bgpInternal) indicates whether or not a route was learned from an iBGP peer. Routes learned via iBGP are allowed to be exported to eBGP peers but not to other iBGP peers. If a router decides to forward traffic to an iBGP peer, we lookup the actual IGP forwarding behavior from the copy of the network corresponding to that neighbor's next hop destination IP address.

Route reflectors. Route reflectors help scalably disseminate iBGP information among BGP routers by acting as an intermediary. To model route reflectors, we use a slightly modified scheme from that described above for iBGP. Each symbolic record includes a variable (originatorId) indicating the router that initially sent the advertisement. Routes are then exported according the the route-reflector semantics (*e.g.*, route reflectors reflect routes with a Non-Client originatorId to Clients). Client routers then lookup next-hop forwarding reachability based on the copy of the network corresponding to the value of originatorId. Loops (*e.g.*, those prevented with the CLUSTER_ID attribute) are handled similarly to BGP (see §6).

Multi-exit discriminator (MED). The MED attribute of BGP routes allows an AS to indicate preferences for paths for incoming traffic (i.e., "cold potato" routing). There are multiple ways in which MEDs may be used by a router depending on the configuration options and router vendor. In one usage, the MED values are compared independent of the next-hop AS. We model this case by ensuring that MEDs are compared when computing the best route (*e.g.*, $\text{best}_{\text{BGP}}.\text{med} \leq \text{in}_1.\text{med}$). In another usage, the MED values are compared only for routes with the same next hop AS. To model this case, we first add a variable to each symbolic control plane record that "remembers" what neighboring AS the route was learned from. The import function from an external neighbor will set the value of the next hop AS. The best route constraints then only compare the MED when the AS is the same. For example, we generate the constraint:

$$(\text{best}_{\text{BGP}}.\text{asn} \neq \text{in}_1.\text{asn}) \vee (\text{best}_{\text{BGP}}.\text{med} \leq \text{in}_1.\text{med})$$

In yet another usage, the age of a route determines the route comparison order, which means that routes with worse MED values may be chosen over those with better values even when the routes have the same next hop AS. Rather than model the age of each route, we overapproximate this behavior by selecting any best route without comparing MEDs.

MEDs are also non-transitive, i.e., the AS that receives them does not export them to other ASes. We model non-transitivity similarly to iBGP. We add a variable indicating whether a MED was learned from an external peer, or set within the current AS. Routes with MEDs learned from a peer are not exported to other ASes.

Design Decisions and Limitations. Our verification approach is general and flexible, but it does have limitations. The most critical design choice involves the fact that our system describes the stable solutions to which the control plane will converge; it does not simulate the execution of the control plane as a message-passing system. This choice improves performance, but it also means we give up the possibility of verifying properties about transient states of the network, prior to convergence. Other verification tools such as ERA [11] and ARC [14] share this limitation.

A second important design decision is that we only consider elements of the control plane that influence the forwarding decisions pertaining to a *single* symbolic packet at a time. As a result, it is more expensive to model a few features that introduce dependencies among destinations. For example, it is possible for static routes to specify a next hop IP address that does not belong to a directly-connected interface, thereby requiring the model to understand how to route to that next hop. In this case, we must create a separate copy of every control plane variable to determine the forwarding for a second packet corresponding to the next hop address. Likewise, modeling iBGP requires one additional copy of every control plane variable for every router configured with iBGP. This additional complexity appears inherent since such features introduce cross-destination dependencies. We are not aware of any other verification tool that models them at all.

5 PROPERTIES

As noted earlier, our model allows us to express a range of properties using SMT constraints. We now show how to encode some common properties of interest.

Reachability and Isolation. We focus on answering reachability queries for a fixed destination port and set of source routers. To answer such a query, each router x is instrumented with an additional variable canReach_x representing the fact that the router can reach the destination port. We then add constraints as in §3. Isolation is checked by asserting that a collection of routers are not reachable.

One benefit of the graph-based encoding is that queries can involve many routers at once and the solver will analyze their joint impact. For example, to check if two routers r_1 and r_2 can both either reach or not reach the destination, one would assert $\text{canReach}_{r_1} \iff \text{canReach}_{r_2}$. Similarly, the user can check if all routers from a set S can reach the destination in a single query by checking: $\bigwedge_{s \in S} \text{canReach}_s$.

In contrast, in existing data plane and control plane verification tools, to answer questions about reachability between all pairs of n devices, one is often required to run n^2 separate queries, which can be very expensive [27].

Waypointing. Suppose we want to verify that traffic will traverse a chain of devices $m_1, \ldots, m_k$. Rather than adding one variable for each router as with reachability, instead we add k variables for each router to indicate how much of the service chain has been matched. If a router forwards to neighbor m_j and its $(j-1)$th variable is true, then the jth variable must be true for that router. Routers where the kth variable is true will send traffic through the service chain.

Bounded or Equal Path Length. In many settings, it is desirable to guarantee that traffic follows paths of certain length. For example, for a data center with a folded-Clos topology, an operator may wish to ensure that traffic never traverses a path longer than four hops. A violation of such an invariant likely indicates a configuration bug. Similarly, the operator may want to ensure that all top-of-rack routers in a pod use equal length paths to the destination.

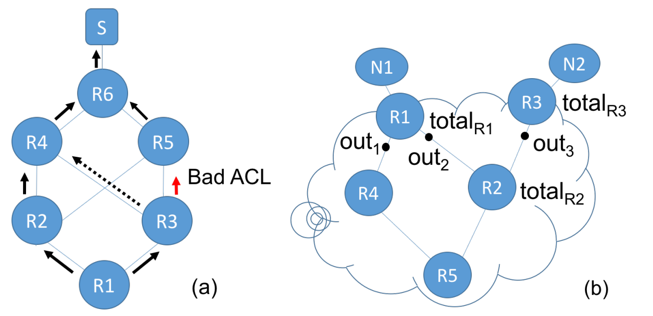

Figure 6: Example networks for encoding (a) multipath consistency, and (b) load balancing.

Similar to reachability, path length is easily instrumented in the model by adding a new integer variable for each router in the network. Each router has path length n to the destination if it forwards to some neighbor with path length $n - 1$.

Disjoint Paths. It is possible to ensure that two different routers use edge-disjoint paths to a destination. Given two routers, we add two bits to each edge indicating whether either router ever forwards through that edge. A constraint then states that both bits are never set for any edge. A similar approach can be used to guarantee that paths do not share nodes or other shared-risk elements (*e.g.*, fiber conduits), by introducing a variable for each risk factor.

Forwarding Loops. Forwarding loops in the network can arise from configuration errors when using features like route redistribution and static routes. To detect forwarding loops for a particular router r, we add a single control bit to say whether each other router will eventually send traffic through r. If r sends traffic to any neighbor with this bit true, then there will be a forwarding loop. As an optimization, we analyze configurations to identify routers where a forwarding loop is possible (*e.g.*, due to the presence of static routes). We then add control bits only for these routers.

Black Holes. Black holes occur when traffic is dropped because it arrives at a router that does not have a corresponding forwarding entry. This behavior may be intentional (*e.g.*, in the case of ACLs) or unintentional. We can find black holes by checking if any router has a neighbor that forwards to it, yet the router itself does not forward to any neighbor.

Multipath Consistency. Batfish [13] introduced a property called multipath consistency, which ensures that traffic along all paths from a source is treated the same. A violation of multipath consistency occurs when traffic is dropped along one path but not the other. Consider the example in Figure 6(a). Router R1 is configured to use multipath routing, yet an ACL on router R3 prevents traffic from using the link to R5. We encode multipath consistency as follows.

$$\begin{aligned}\text{canReach}_{R1} &\implies \textstyle\bigwedge_{R \in \{R2, R3\}} (\text{controlfwd}_{R1,R} \implies \\ &\qquad \text{datafwd}_{R1,R} \wedge \text{canReach}_R) \\ \text{canReach}_{R3} &\implies \ldots\end{aligned}$$

The first constraint says that if R1 can reach the destination S at all, then forwarding to R2 (R3) in the control plane implies that R2 (R3) should also be able to reach the destination, and this also aligns with forwarding in the data plane to R2 (R3). In the example presented in Figure 6(a) this constraint will fail since R3 cannot reach the destination, due to the bad ACL to R5. Suppose now that R3 can also use multipath routing, and can therefore reach the destination via R4 (shown as the dotted edge). Now the first constraint at R1 will succeed, but the second constraint for R3 will fail, because R3 can forward through R4 but not through R5.

Neighbor or Path Preferences. Operators often want to enforce preferences among external neighbors based on commercial relationships. For example it is common to prefer routes learned from customers over peers over providers. Given a router R with three edges to neighbors n1, n2, and n3 with import records e1, e2, and e3, we can verify that n1 is preferred over n2 over n3 in the following way. For each neighbor, we add a constraint that, if a message survives the import filter, and all other more preferred neighbor advertisements do not, then the presence of the message implies that we will choose that neighbor in the selection process:

$$\begin{aligned}\text{e1.valid} &\implies \text{controlfwd}_{R,N1} \\ \neg\text{e1.valid} \wedge \text{e2.valid} &\implies \text{controlfwd}_{R,N2} \\ \neg\text{e1.valid} \wedge \neg\text{e2.valid} \wedge \text{e3.valid} &\implies \text{controlfwd}_{R,N3}\end{aligned}$$

This type of reasoning can be lifted to entire paths. For example suppose we want to verify that the network prefers to use $path_1 = x_1, \ldots, x_m$ over $path_2 = y_1, \ldots, y_n$. What we want to check is that if the less preferred path is used, then the more preferred path was not available:

$$\bigwedge_{i=1}^{n-1} \text{controlfwd}_{y_i, y_{i+1}} \implies \bigvee_{i=1}^{m-1} \neg\text{e}_i\text{.valid}$$

That is, whenever traffic flows along $path_2$ it is because $path_1$ is not available due the advertisement being rejected along one of the edges. A straightforward generalization of the above can help enforce preferences over classes of neighbors, instead of individual neighbors.

Load Balancing. Consider the example network in Figure 6b. Suppose router R1 is configured to use ECMP to send traffic to R2 and R4. We can roughly model the effect of load distribution with the following steps. First, for each router R in the network we introduce a symbolic real number called total_R representing the portion of traffic going through R. For each source router of interest (*e.g.*, R1 and R3), we set the load to some initial value based on traffic measurements (*e.g.*, 1.0 in this example):

$$\text{total}_{R1} = 1.0 \wedge \text{total}_{R3} = 1.0$$

For each outgoing interface in the network, we add a variable out_i representing the fraction of the load sent out that interface, which depends on the forwarding behavior.

$$\begin{aligned}\text{out}_1 &= \text{if datafwd}_{R1,R4} \text{ then } x \text{ else } 0.0 \\ \text{out}_2 &= \text{if datafwd}_{R1,R2} \text{ then } x \text{ else } 0.0 \\ \text{total}_{R1} &= \text{out}_1 + \text{out}_2\end{aligned}$$

Each interface's load is equal to the (same) value defined by a single new variable x if traffic is forwarded out the interface, otherwise it is 0. This new variable x ensures the loads are all equal.[4] The total at non-source routers is simply the sum of their incoming totals:

$$\text{total}_{R2} = \text{out}_2 + \text{out}_3$$

Now we can ask questions about the load on each node/edge. For example, we can check that the difference between the loads on R2 and R4 is always within some threshold k:

$$-k \leq \text{total}_{R2} - \text{total}_{R4} \leq k$$

Aggregation and Leaking Prefixes. We can ensure that prefixes are aggregated properly (*e.g.*, a /32 is not leaked to an external network) by checking: whenever the network advertises record e to an external neighbor, then e.length = l where l is prefix length after aggregation.

Local Equivalence. In many networks (*e.g.*, data centers), several devices will perform a similar "role" (*e.g.*, aggregation router) and have similar configurations. Checks for equivalence can help detect inconsistencies. For example, we might want to know that a particular community value is always attached to advertisements sent to external neighbors.

Because we fully model each router's interactions with all of its neighbors, we can check if two routers are behaviorally equivalent for some notion of equivalence. In particular, we ask if given equal environments (i.e., peer advertisements), the routers will make the same forwarding decisions and export the same new advertisements. For example, if two routers R1 and R2 both have the same two peers P1 and P2 with import records in_1 and in_2, and output records out_1 and out_2, then we check the following:

$$\begin{aligned}\text{in}_1 = \text{in}_2 \implies\; &(\text{out}_1 = \text{out}_2) \wedge \\ &(\text{datafwd}_{R1,P1} = \text{datafwd}_{R2,P1}) \wedge \\ &(\text{datafwd}_{R1,P2} = \text{datafwd}_{R2,P2})\end{aligned}$$

Full Equivalence. It is also possible to check full equivalence between two sets of router configurations. This is done in a similar way as the local equivalence check, by first making two separate copies of the network encoding, and then relating the environments. As before, we check that all the final data plane forwarding decisions and all exports to neighboring networks must be the same as a result.

Fault Tolerance. Configurations that work correctly in the absence of failures may no longer work correctly after one or more links fail. For each property above, we can verify that it holds for up to k failures by adding the following constraint on the number of links that are failed:

$$\sum_{(x,y) \in \text{edges}} \text{failed}_{x,y} \leq k$$

[4]This could be easily extended to weighted ECMP by scaling x by a constant according to the fraction of traffic split.

Because link failures are part of the network model, the solver will learn facts about the impact of failures on the rest of the network control plane. This behavior means that properties involving failures can often be checked more efficiently than iterating over failure cases using a failure-free model (*i.e.*, verifying a property multiple times independently, once for each failure case).

Fault-Invariance Testing. We can use the same strategy as full equivalence checking to instead check if the same property holds in a single network regardless of failures. For example, even if we do not know whether two routers should be able to reach one another (a possible problem when analyzing networks without specifications), we can check that the two routers are reachable if and only if they are reachable after any single failure. Such a test can find instances where network behavior differs after failures. To check fault-invariance with respect to a property P, we create two copies of the network. For the first copy, we require that there are no failures. For the second copy, we allow there to be any k failures. We then check that P holds in the first copy of the network exactly when it holds in the second copy.

6 OPTIMIZATIONS

While conceptually simple, the naive encoding of the control plane described in §3 does not scale to large networks. We present two types of optimizations that dramatically improve the performance of the control-plane encoding.

6.1 Hoisting

Hosting lifts repeated computations outside their logical context and precomputes them once. Two main optimizations of this class that we use are:

Prefix elimination. Our naive encoding does not scale well in large part because of the constraints of the form FBM($p1$, $p2$, n), which checks that two symbolic variables have the first n bits in common. The natural way to represent $p1$ and $p2$ for this check is to use 32-bit bitvectors and check for equality using a bit mask. However, bitvectors are expensive and solvers typically convert them to SAT. In our model, this would introduce up to 128 new variables for every topology edge in the network (4 records per edge) thereby introducing an enormous number of additional variables.

To avoid this complexity, we observe that the prefix received from a neighbor does not actually need to be represented explicitly. In particular, because we know (symbolically) the destination IP address of the packet and the prefix length, there is a unique valid, corresponding prefix for the destination IP. For example, if the destination IP is 172.18.0.4 and the prefix length is /24, and the route is valid for the destination, then the prefix must be 172.18.0.0/24[5].

However, we must still be able to check if a prefix is matched by a router's import or export filter. Somewhat unintuitively, we can safely replace any filter on the destination prefix with a test on the destination IP address directly, thereby avoiding the need to explicitly model prefixes. Consider the following prefix filter:

```
ip prefix_list L allow 192.168.0.0/16 ge 24 le 32
```

Its semantics is that it succeeds only if the first 16 bits of 192.168.0.0 match the prefix, and the prefix length is greater than or equal to 24 and less than or equal to 32. In general, for a prefix filter of the form `P/A ge B le C` to be well formed, vendors require that $A < B \leq C$. A simple translation of this for SMT record e is:

FBM(e.prefix, 192.168.0.0, 16) $\wedge$ ($24 \leq e$.length ≤ 32)

Suppose now, we replace the test on the prefix contained in the control plane advertisement with a test directly on the destination IP address of a packet of interest:

FBM(dstIP, 192.168.0.0, 16) $\wedge$ ($24 \leq e$.length ≤ 32)

There are two cases to consider. First, if e.length is not between 24 and 32, then both tests fail, so they are equivalent. Suppose instead, e.length is in this range. Recall that, because we are considering a slice of the network with respect to the destination IP address, for the advertisement corresponding to e to be valid, it must be the case that the prefix contains the destination IP. That is: FBM(e.prefix, dstIP, e.length). However, because we know the prefix length falls in the range between 24 and 32, it must be greater than 16. Since the first bits up to the prefix length are common between the destination IP and the prefix, the first 16 bits must also be the same. Therefore the above substitution is equivalent.

Further, because the test FBM is now purely in terms of constants in the configuration (not the symbolic prefix length variable), we can represent the destination variable as an integer and implement the test using the efficient theory of integer difference logic (IDL). Thus, we would test that:

(192.168.0.0 $\leq$ dstIP $<$ 192.168.0.0 + 2^{32-16}) $\wedge$
($16 \leq e_4$.length ≤ 32)

Loop Detection. In protocols that support policy-based routing (e.g., BGP), path length alone does not suffice to prevent loops. For this reason, BGP tracks the ASNs (autonomous system numbers) of networks along the advertised path and routers reject paths with their own ASN. We can model this by maintaining, for each BGP router, a control bit saying whether or not the advertised path already went through that router. However, doing so can be expensive since the number of control bit variables grows with the square of the number of routers. Instead, we observe that any BGP router that uses only default local preferences (*i.e.*, only makes decisions based on path length) will never select a route where it is already part of the AS path. This is because the path containing the loop is strictly longer than the path without the loop. For example, if AS 1 uses shortest path routing only, then the AS path 1 2 1 3 can never arise in our model since AS 1 would prefer the path 1 3 instead. Similarly, BGP local preferences for external neighbors and for iBGP peers will not create loops. This optimization makes it possible to forgo modeling loops in most cases.

6.2 Network Slicing

Slicing removes bits from the encoding that are unnecessary for the final solution. We use the following slicing optimizations:

- Remove symbolic variables that never influence the decision process. For example, if BGP routers never set a local preference, then the local preference attribute will never affect the decision and can be removed.
- Keep a single copy of import and export variables for an edge when there is no import filter on the edge. The two variable sets will simply be copies of each other.

[5]Alternatives such as 172.18.0.1/24 are treated identically.

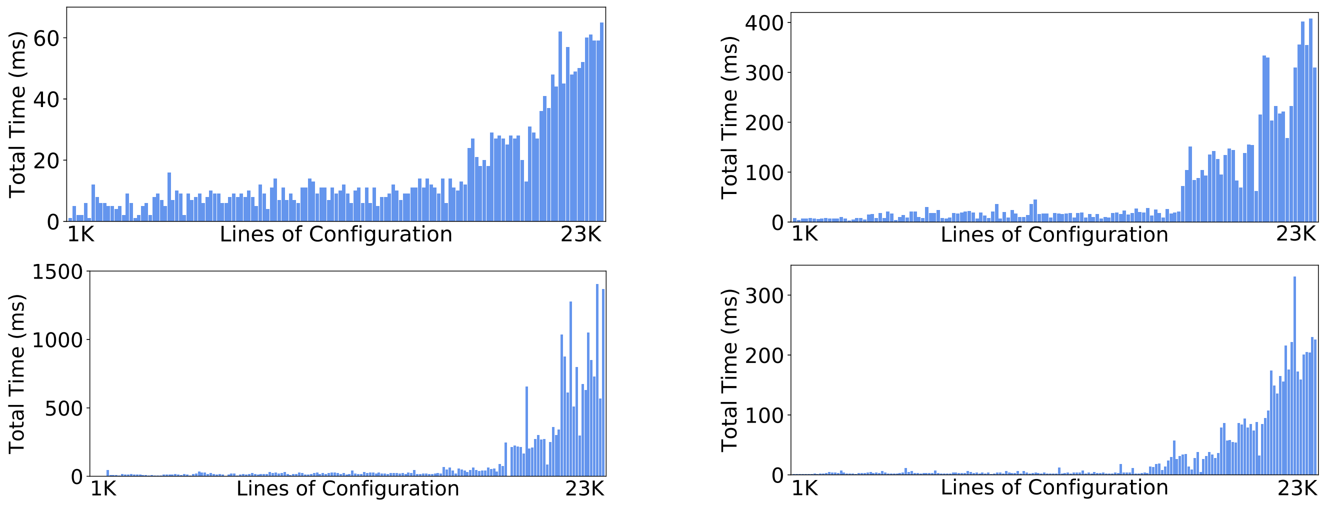

Figure 7: Verification time for management interface reachability (upper left), local equivalence (upper right), blackholes (lower left), and fault-invariance (lower right) for real configurations sorted by total lines of configuration.

• Keep a single, merged copy of the export record for a protocol when there is no peer-specific export policy.
• Do not model directly connected routes for a router whose interface addresses can never overlap with the destination IP range of interest to the query.
• Merge the data plane and control plane forwarding variables along edges that do not have ACLs.
• Merge per-protocol and overall best records when there is only a single protocol running on a router.

Together, these optimizations are effective at removing a lot of redundant information that the SMT solver might otherwise have to discover for itself.

7 IMPLEMENTATION

Minesweeper uses Batfish [13] to parse vendor-specific configurations. It then translates Batfish's representation into a symbolic model. To check model (un)satisfiability, we use the Z3 SMT solver [8]. Our encoding exploits Z3's support for integer difference logic, and its preprocessor. Our implementation supports all of the features and properties described in the paper. We have validated its correctness empirically by comparing its output to that of the Batfish simulator on a large collection of networks. As Batfish does not currently support IPv6, Minesweeper does not either. Minesweeper is available as open source software [3].

8 EVALUATION

We evaluate Minesweeper by using it to verify a selection of the properties described in §5 on both real and synthetic network configurations. In particular, we are interested in measuring (1) the ability of Minesweeper to find bugs in real configurations, which are otherwise hard to find; (2) its scalability for answering various queries on large networks; and (3) the impact of the optimizations described in §6 on performance. All experiments are run on an 8 core, 2.4 GHz Intel i7 processor running Mac OSX 10.12.

8.1 Finding Errors in Real Configurations

We demonstrate Minesweeper's ability to find bugs in real configurations by applying it on a collection of configurations for 152 real networks. We obtained these from a large cloud provider, and they represent different networks within their infrastructure. The networks range in size from 2 to 25 routers with 1–23K lines of configuration each. The networks use a combination of OSPF, eBGP, iBGP, static routes, ACLs, and route redistribution for layer-3 routing and are part of a data set described in detail in prior work [15]. These networks have been operational for years, and thus we expect that all easy-to-find bugs have already been ironed out. This data set was also analyzed by ARC [14].

Properties checked. Since we do not have the operator-intended specifications, we focus on four properties expected to hold in such networks:
• Management interface reachability: All nodes in the network should be able to reach each management interface, irrespective of the environment. Management interfaces are used to log into the devices, manage their firmware and configuration, and collect system logs. Uninterrupted access to it is important for the network's security and manageability.
• Local equivalence: Routers serving the same role (e.g., as "top-of-rack") should be similar in how they treat packets. We identify routers in the same role by leveraging the networks' naming convention and check that all pairs of routers in the network in a given role are equivalent.
• No blackholes: When traffic is dropped due to ACLs, such dropping should always occur at the edge of the network.
• Fault-invariance: All pairs of routers in the network should be reachable from one another if and only if they are reachable after a single failure. A violation of this property would indicate that the network is highly vulnerable to failures.

Violations. We found 67 violations of management interface reachability. In each case, the violation occurs because of a "hijack," i.e.,

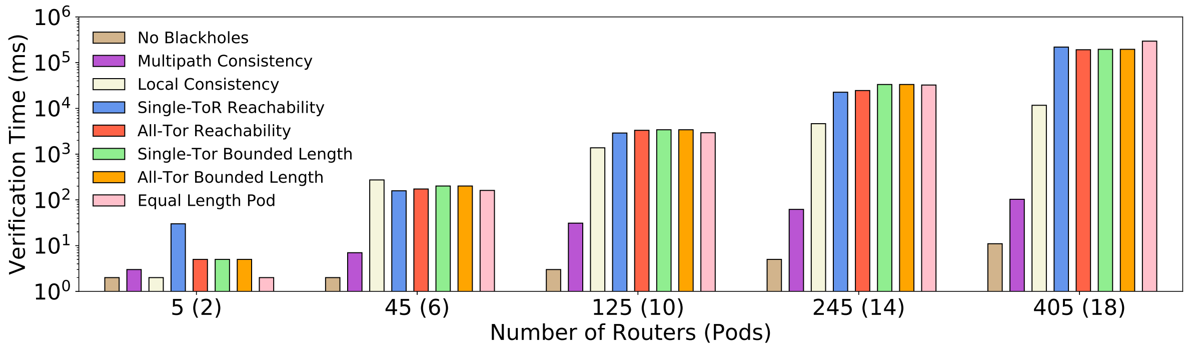

Figure 8: Verification time for synthetic configurations for different properties and network sizes.

external neighbors sending particular announcements. For example, an external BGP advertisement for the same /32 interface prefix with path length ≤ 1 would result in a more preferred route for the destination that would ultimately divert traffic away from the correct interface.

The checks for local equivalence revealed 29 violations. Upon further investigation, we found that each violation was caused by one or more exceptions in ACLs where almost all routers in a given role would have identical ACLs except for a single router with an extra or a missing entry. Such differences are possibly caused by copy-and-paste mistakes.

The blackholes check found 24 violations. Most violations were not serious issues with routing, but instead revealed optimization opportunities. Traffic being dropped deep in the network could have been dropped near the source.

We found no violations of fault-invariance.

8.2 Verification Performance

We evaluate the performance of Minesweeper to verify different properties on real and synthetic configurations.

Real configurations. We benchmarked the verification time for the networks and properties described above. Figure 7 (upper left) shows this time for management-interface reachability for each network that is configured with at least one management interface. The networks are sorted by total lines of configuration, with more complex networks appearing farther right. We see that the checks take anywhere from 2 to 60 ms for every network tested. Figure 7 (upper right) shows the verification time for local equivalence among routers in each unique role, for all networks with at least two routers in any particular role. Verification time ranges anywhere from roughly 5 to 400 ms. This check is more expensive than management-interface reachability, in part, because it requires more queries. Finally, the lower row of Figure 7 shows the time for verification of the absence of blackholes and fault-invariance queries. Both queries take under a second for most networks. The worst case is under 1.5 seconds. While the networks we studied are small, the sub-second verification times we observe are encouraging. They point to the ability of Minesweeper to verify many real configurations in an acceptable amount of time. Next, we stress test our tool by running it on larger, albeit synthetic networks.

Synthetic configurations. To test the scalability of our tool on larger networks, we use a collection of synthesized, but functional, configurations for data center networks of increasing size. In terms of their structure and policy, these networks are similar to those described in Propane [4]. Each data center uses a folded-Clos topology and runs BGP both inside the network as well as to connect to an external backbone network. Each top-of-rack router in the data center is configured to advertise a /24 prefix corresponding to the shared subnet for its hosts. All routers are configured to enable multipath routing to evenly distribute load across all of its available peers. Spine routers in the data center connect to external neighbors in the adjacent backbone network and are configured to use route filters on all externally connected interfaces to block certain advertisements.

For each network, we use Minesweeper to check a large collection of the properties described in §5. First, we fix a destination ToR and use queries to check both single-source and all-source reachability from other ToRs. Similarly, we also check that both some and all other ToRs will always use a path to the destination ToR that is bounded by four hops, to ensure that traffic never uses a “valley" path that goes down, up, and then down again. To demonstrate a query that asks about more than a single path, we verify that all ToRs in a separate pod from the destination will always use paths that have equal length. This ensures a certain form of symmetry in routing. In addition to path-based properties, we also verify the multipath-consistency property that every router in the network will never have different forwarding behavior along different paths. We also check that every spine router in the network is equivalent using the local-consistency property. To ensure that all n spine routers are equivalent, we check for local equivalence among pairs using $n-1$ separate queries. If all routers are equivalent, then transitively they are equivalent as well. Finally, we verify the absence of black holes in the data center.

Figure 8 shows the time to check each property for data centers of different size. Multipath consistency and the no-blackholes properties are the fastest to check, taking under a second to verify in all cases. This speed is in most part due to the minimal use of ACLs in the configurations. The solver quickly determines that the properties cannot be violated because the control and data planes stay in sync. The next fastest property to verify is local equivalence among spine routers. This check takes under 2 minutes for the largest network. In this case, each pairwise equivalence check takes roughly 145 milliseconds. The most expensive properties pertain to reachability and path-length. For the largest network it takes under 5 minutes to verify such properties. Interestingly, queries checking all-source vs single-source take approximately the same amount of time. Instead of checking the property by issuing multiple queries, as is the case in many prior, path-based tools [11, 34], all-source reachability is a single query in our graph-based formulation.

8.3 Optimization Effectiveness

We evaluated the effectiveness of the optimizations described in §6 by using a benchmark that involved verification of single-source reachability queries. The prefix-hoisting optimization that replaces symbolic variables representing an advertised prefix with instances of the global destination IP variable has a large impact on performance, speeding up verification by over 200x on average. This is due to the fact that bitvectors are expensive for SMT solvers. Solvers typically deal with bitvectors by "bit blasting" them into SAT. However, this introduces 32 additional variables into the model for every edge in the graph. The next two optimizations: merging common import and export records of variables and specializing variables by protocol, are both forms of slicing optimizations. Together, these optimizations improve the performance of the solver roughly 2.3x on average over prefix hoisting alone.

9 RELATED WORK

Our work builds on prior work on network configuration analysis, which we divide into three classes:

(1) Analysis without network models. Tools such as rcc [12], IP Assure [25], and Minerals [2], focus on finding common mistakes and inconsistencies in configurations of different protocols. While this approach can find a range of configuration errors, because it does not build a model of the network, it can have both false positives and false negatives and cannot answer questions about specific network behaviors.

(2) Analysis of individual environments. Configuration testing tools such as Batfish [13] and C-BGP [28] take as input the network's configuration and a concrete environment, simulate the resulting network behavior, and produce the data plane. The resulting network behavior for the data plane can then be analyzed for properties of interest.

The primary disadvantage of testing is that it can feasibly analyze only a small number of environments, while many configuration errors occur only in specific environments. However, unlike our approach, testing can support a more detailed analysis of individual environments (*e.g.*, it can count the exact size of routing tables).

(3) Analysis of many environments. Our approach belongs to this class which can simultaneously analyze multiple environments by building a symbolic network model. Prior work in this class includes FSR [33], ARC [14], ERA [11], and Bagpipe [34]. We borrow heavily from these works. FSR encodes BGP preferences using SMT constraints, our multi-protocol, logical view of the network (Figure 2) is similar to ARC, and our protocol-independent symbolic records (Figure 3) are similar to ERA. But our work goes beyond prior efforts in its scope and generality. We support the entire control plane functionality and a much wider range of properties.

Data plane analysis. Tools for data plane analysis like Anteater [22], HSA [18], Veriflow [19], NoD [21], SymNet [31] and Delta-net [17], have a simpler task than configuration analysis—they do not have to model the control-plane dynamics that produce many possible data planes. In fact, the configuration testing tool Batfish, first simulates the control plane on a concrete environment to produce a single data plane, and then uses existing data plane analysis tools to verify properties for this data plane. Hence, data plane can be thought of as a special case (or subcomponent) of configuration testing, though the specifics differ greatly.

Methodologically, the data plane analysis tool most similar to our work was developed by Zhang and Malik [35]. They encode the data plane as a SAT formula and use combinational search, like Minesweeper, to find errors.

Configuration synthesis.

Network configuration synthesis [4, 24, 25] is complementary to verification. Synthesis tools *produce* configurations from high-level specifications; verification tools *analyze* configurations (produced manually or by synthesis tools).

Our SMT-based control plane model has some similarity with a contemporary synthesis project [10], but there are significant differences as well. That effort uses a symbolic representation of network protocols based on stratified Datalog, such that the fixed point of the Datalog program represents the forwarding state of the network. The synthesis problem, of finding configuration inputs that satisfy specified properties, is effectively reduced to satisfiability checking of an SMT formula that is generated by using a specialized solver for stratified Datalog. In contrast, we do not restrict ourselves to stratified Datalog and use first order theories supported by SMT solvers to symbolically model the stable states of the network. Our network model might also be useful for finding configuration inputs that satisfy network-wide properties, but we leave an exploration of this topic to future work.

10 CONCLUSIONS

We present a general-purpose, symbolic model of the network control and data planes that encodes the stable states of a network as a satisfying assignment to an SMT formula. Using this model, we show how to verify a wide variety of properties including reachability, fault-tolerance, router equivalence, and load balancing, for all possible packets and all possible data planes that might emerge from the given control plane. We have implemented our approach in a tool called Minesweeper to verify properties of real network configurations. We use Minesweeper on a collection of real and synthetic configurations, showing that it is effective at finding issues in real configurations and can scale to large networks.

Acknowledgements. We thank the SIGCOMM reviewers and our shepherd Laurent Vanbever, whose extensive feedback helped improve this paper. This work was supported in part by NSF Grants 1703493 and 1525936, and a gift from Cisco. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect those of the NSF or Cisco.

REFERENCES

[1] M. Anderson. Time warner cable says outages largely resolved. http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved, 2014.
[2] L. Bauer, S. Garriss, and M. K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. *ACM Trans. Information and System Security*, 14(1), 2011.
[3] R. Beckett. Minesweeper source code. https://batfish.github.io/minesweeper, 2017.
[4] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *SIGCOMM*, 2016.
[5] News and press | BGPMon. http://www.bgpmon.net/news-and-events/.
[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, 1999.
[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Programming Languages and Systems*, 8(2), 1986.
[8] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
[9] L. De Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9), 2011.
[10] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev. Network-wide configuration synthesis. In *CAV*, 2017.
[11] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *OSDI*, 2016.
[12] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, 2005.
[13] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.
[14] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *SIGCOMM*, 2016.
[15] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan. Management plane analytics. In *Internet Measurement Conference (IMC)*, 2015.
[16] T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Networking*, 10(2), 2002.
[17] A. Horn, A. Kheradmand, and M. Prasad. Delta-net: Real-time network verification using atoms. In *NSDI*, 2017.
[18] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.
[19] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.
[20] D. Kline. Network downtime results in job, revenue loss. http://www.avaya.com/en/about-avaya/newsroom/news-releases/2014/pr-140305/, 2014.
[21] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *NSDI*, 2015.
[22] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *SIGCOMM*, 2011.
[23] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM*, 52(8), 2009.
[24] S. Narain, G. Levin, S. Malik, and V. Kaul. Declarative infrastructure configuration synthesis and debugging. *Journal of Network Systems Management*, 16(3), 2008.
[25] S. Narain, R. Talpade, and G. Levin. *Guide to Reliable Internet Services and Applications*, chapter Network Configuration Validation. Springer, 2010.
[26] J. Networks. As the value of enterprise networks escalates, so does the need for configuration management. https://www-935.ibm.com/services/au/gts/pdf/200249.pdf, 2008.
[27] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling network verification using symmetry and surgery. In *POPL*, 2016.
[28] B. Quoitin and S. Uhlig. Modeling the routing of an autonomous system with C-BGP. *IEEE Network*, 19(6), 2005.
[29] S. Sharwood. Google cloud wobbles as workers patch wrong routers. http://www.theregister.co.uk/2016/03/01/google_cloud_wobbles_as_workers_patch_wrong_routers/, 2016.
[30] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3), 1985.
[31] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: Scalable symbolic execution for modern networks. In *SIGCOMM*, 2016.
[32] Y. Sverdlik. Microsoft: misconfigured network device led to azure outage. http://www.datacenterdynamics.com/content-tracks/servers-storage/microsoft-misconfigured-network-device-led-to-azure-outage/68312.fullarticle, 2012.
[33] A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. L. Talcott. FSR: Formal analysis and implementation toolkit for safe inter-domain routing. *IEEE/ACM Trans. Networking*, 20(6), 2012.
[34] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock. Formal semantics and automated verification for the border gateway protocol. In *NetPL*, 2016.
[35] S. Zhang and S. Malik. SAT based verification of network data planes. In *Automated Technology for Verification and Analysis (ATVA)*, 2013.