UNIVERSITY OF CALIFORNIA

Los Angeles

Exploiting Modularity to Scale Verification of Network Router Configurations

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Alan Tang

2023

© Copyright by
Alan Tang
2023

ABSTRACT OF THE DISSERTATION

Exploiting Modularity to Scale Verification of Network Router Configurations

by

Alan Tang
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2023
Professor Todd Millstein, Co-Chair
Professor George Varghese, Co-Chair

Network router configuration errors are a common cause of network failures. Verifying configurations using static analysis can prevent errors from occurring, but existing verifiers, especially for the control plane, have two problems: they cannot scale to large networks, and they cannot localize errors to their source in the configurations. This dissertation proposed the use of modular techniques for verifying control plane properties in the network. While previous verifiers model the behavior of the network holistically, modular verification guarantees useful network properties while executing checks on individual routers and connections, allowing it to scale better and achieve better localization. First, I present Campion, a tool for modularly checking equivalence between a pair of router configurations by looking at corresponding components, allowing it to localize the source of the differences to lines and structures in the configuration. Next, I present a tool that verifies network-wide BGP properties using only local checks on configurations. The technique uses local constraints to bridge the gap from local properties to network properties. Both of these techniques have been deployed and have found numerous previously unknown bugs in real-world networks.

The dissertation of Alan Tang is approved.

Ryan Beckett

Jens Palsberg

Yuval Tamir

Todd Millstein, Committee Co-Chair

George Varghese, Committee Co-Chair

University of California, Los Angeles

2023

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I do not think I understood what I was getting into when I started graduate school. Looking back, I think it was a good experience, and I am grateful to all the people I have met and worked with along the way. First, I want to thank my advisors Todd and George for all the guidance and support they have provided. They have opened up many opportunities for me, and there were many moments when I would have been lost without all their help. I would like to thank Ryan Beckett, Jens Palsberg, and Yuval Tamir for serving as members of my committee. I am thankful to the many other members of the PL lab who I have had conversations with over the years. And of course, there were many co-authors and collaborators I have worked with over the last few years including Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Siva Kesava Reddy Kakarla, Tejas Patil, Yuval Tamir, and Ennan Zhai. The work presented would not have been possible without their contributions and expertise.

CHAPTER 1

Introduction

The Internet provides an unbelievably wide range of services and has become an indispensable part of modern life. As the usage of Internet services increases, it becomes more and more important that the networks providing these services operate reliably. At the heart of the Internet is its ability to connect thousands of independently owned networks, allowing packets to reach from one location to another. This enables reliable communication between users and services across the globe. However, sometimes problems arise, breaking the connections in the Internet. When these occur, users are deprived of their access to the service, and companies can suffer significant financial losses.

Devices determine where to send a packet through the process of *routing*, using one or more routing protocols. As part of the protocol, routers exchange information about the locations they can reach, allowing them to compute the path that a packet takes across the network in a distributed fashion. The end result of this process is a forwarding table installed in each router that determines which neighbor to send a packet to. The processes and policies that produce the forwarding tables are known as the *control plane*, whereas the network processes that actually forward the packets are known as the *data plane*.

Routing can be accomplished by computing the shortest path between different locations, but often there are additional considerations. For example, networks may have backup paths for reliability. They may reject messages that they believe are malicious or erroneous. Or they may prefer some routes over others because of financial reasons. These are all handled by the control plane. To implement the various network policies, each router is loaded

with a *configuration* that defines its behavior. These are responsible for determining which protocols the router supports and which routes and packets are rejected or propagated.

Many network failures are caused by routing errors stemming from the router configuration. Typically, human operators use manual, low-level configuration directives at individual routers to enforce complex policies for access control and routing. Manual configuration can introduce subtle configuration errors that induce costly and disruptive outages. These have occurred in companies like United Airlines [38], Google [32, 52], Microsoft [9], Twitter [43], and more [12]. As the importance of the Internet and the scale of its services increases, outages like these will become more costly for users and companies.

Often, errors occur when network operators are updating the network policies and the configurations are being modified. To prevent this, some companies have implemented operational approaches such as reviewing configuration changes or using templates to ensure consistency among different configurations. This is combined with constant monitoring to quickly find and fix any problems that arise. While this can help reduce and resolve problems, it cannot completely prevent errors or the problems that they cause.

Another approach to preventing costly network configuration errors is verification. Many verification tools have been developed to check both the data plane and the control plane using a variety of techniques. Given a set of network configurations and a specification for what polices the network should implement, these tools can automatically check the correctness of the configurations. While tools for checking data plane properties have been successfully deployed, tools for verifying control plane properties have not been widely adopted, even though many existing techniques can provide strong guarantees, frequently reasoning about network behavior over all possible external routing announcements and/or link failures.

There are two challenges in getting network verification techniques widely adopted. The first challenge is scalability. Previous network control plane verification tools typically model the network holistically, implicitly or explicitly considering all possible states of the network. These approaches fundamentally cannot scale to large networks. Verifying a system requires

exploring a large state space that grows exponentially in the size of the network. This results in a trade off between scalability and the types of properties that can be checked. The second challenge is making the tool outputs usable. The end goal of verification to create robust systems, so it is not enough to simply identify if the behavior does not match a specification. Tools should provide the necessary information to allow users to fix errors. While researchers have developed many verification tools that can analyze network configurations to find errors, it is difficult to trace the results of these tools to the source of the errors since it is hard to map back to the configuration text from the holistic models they use.

For this thesis, I propose the use of modular verification of network configurations for checking control plane behavior. Instead of the monolithic approach used in previous verification tools, I introduce approaches to verifying properties that use multiple simple checks, each on smaller blocks of configuration. The combination of these checks either verifies a network property or provides feedback to help operators fix errors. With a modular approach, each individual check is limited in size. Since each check is simple, verification scales roughly linearly in the total number of devices and links. Modular checking improves the quality of results by being able to provide better localization. When an error is found in a local check, it can be traced to the location that caused the error. In a real world context where inexact specifications or implementation errors can lead to false positives, it even becomes difficult to know whether there is truly an error.

The key insight is that networks and network policies are already designed in a modular fashion. They are not designed in a chaotic or completely unorganized manner, as that would make it difficult for them to be maintained by operators. Rather, each router in the network has a particular role and a set of policies that it has to enforce. The global network policy is a combination of individual router policies, which are each a combination of policies for the various interfaces and protocols. Since operators configure their networks in a modular way, it should be possible to check behaviors in a modular fashion.

The challenge of this approach is that it becomes more difficult to reason about network-

wide behavior resulting from multiple routers, such as whether a packet can reach from one end of the network to the other. My work overcomes this problem by showing how the behavior of smaller modules in a network can imply behavior in a larger scope. The trade off is that checking properties may require more local specification from the user, similar to how users need to provide loop invariants when verifying software [21].

The remainder of this introduction gives more background on control plane verification and summarizes my contributions. Section 1.1 explains the basics of the control plane, routing, and configurations. Section 1.2 shows how misconfigurations may impact the network and the challenges of network verification. Finally section 1.3 gives an overview to the modular verification techniques that I develop.

1.1 Network Control Plane

1.1.1 What is the Control Plane?

In the Internet, routers typically perform two tasks: (1) *routing*, the process of selecting paths through the network, and (2) *forwarding*, the process of sending a data packet along its selected path. The Internet is designed to send packets from one location to another. Packets contain headers, and these headers contain fields like the source IP address and destination IP address that identify the sender and intended receiver of the packet. The *data plane* comprises the part of the network responsible for forwarding packets. When a router forwards packets, it uses a computed *forwarding table* to determine the path to send each packet along. The forwarding table consists of a number of entries that match an IP prefix to an outgoing interface. When a router receives a packet, it finds the longest matching prefix in the forwarding table for that packet's destination IP address, and it sends the packet along the corresponding outgoing interface in the forwarding table. For example, the prefix $10.1.2.0/24$ represents the set of IP address in which the first 24 bits match 10.1.2. If there exists an entry in the forwarding table with that prefix, a packet with destination IP address 10.1.2.100 arrived at the router, and there was no longer prefix matching that address, the

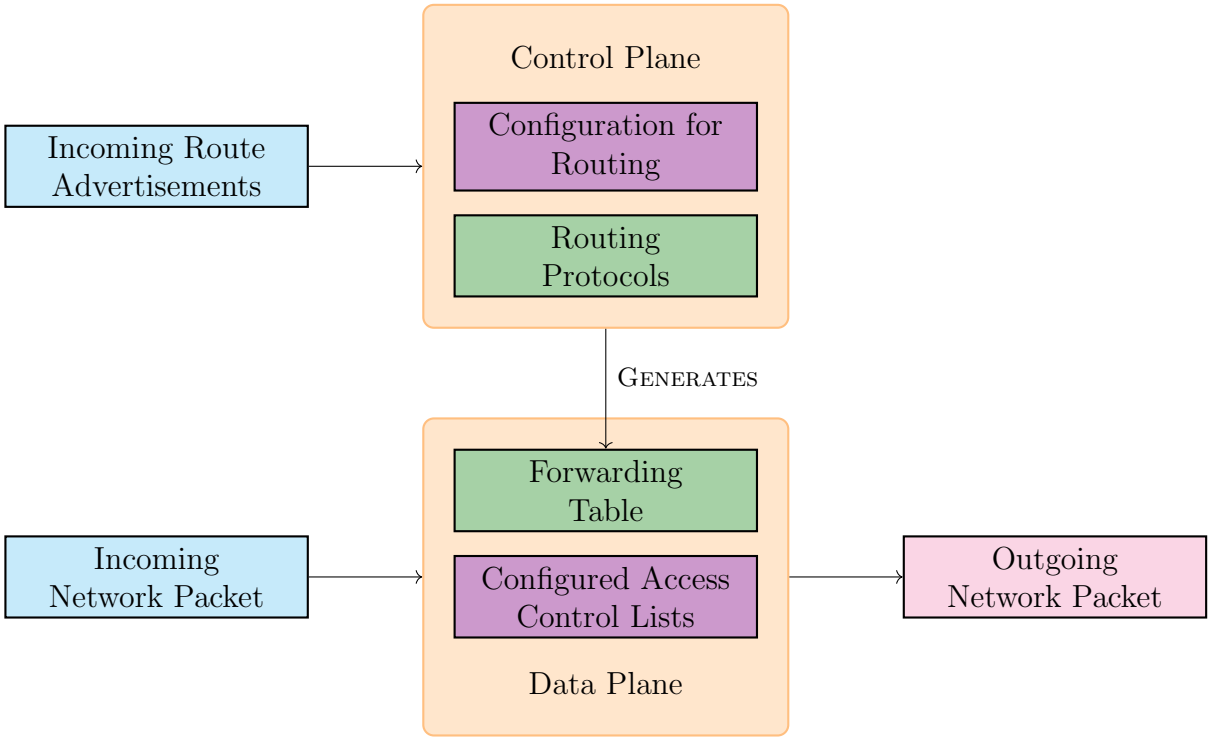


Figure 1.1: Relationship between network control and data planes

packet would get forwarded along the interface specified by the forwarding table.

Routers may additionally be configured with access control lists (ACLs) that specify packets to be dropped. These can be used to filter packets based on destination IP, source IP, or a few other fields. As shown in the bottom of figure 1.1, the data plane can be thought of as a system that takes an incoming network packet and sends an outgoing packet in the appropriate directions.

The *control plane*, on the other hand, comprises the part of the network responsible for determining how packets should be routed. To determine where a packet should be sent, routers coordinate using routing protocols to compute a path for each packet. Routing protocols are fixed processes designed to compute a best path. Routers typically have a number of IP prefixes that they can reach directly along an interface. As part of the protocol, routers in a network will advertise routes to their neighbors, with each route consisting of an IP prefix along with additional attributes. This information gets propagated through

the network. Routers then select a best route for each prefix based on the information they receive, and install those chosen routes into the forwarding table. The results of this routing process can be influenced by the router configurations and the route advertisement messages that are received. In some cases, failures can impact the computation of paths as well. As shown in the top of figure 1.1, the control plane can be thought of as system that uses a set of incoming route advertisements to generate a forwarding table.

Much of this thesis is devoted to verifying the control plane behavior of the network router configurations. The problem setting for control plane verification is as follows:

1. **Configurations for routers inside a network are provided.** These configurations define a number of policies, settings, and routes which impact the control plane.

2. **A *specification* is provided.** The specification indicates some desired property of the forwarding table or of intermediate results such as the routing messages that are sent. For example, users may specify that a particular router has a route for the prefix `10.0.0.0/8` installed.

3. **All possible *environments* are considered.** An environment consists of the set of route advertisements received from outside the network. These can change the results of the control plane, and verifiers must consider all possible cases.

4. **The verifier returns whether the specification holds for all possible environments.** In the case that the property does not hold, it produces a counterexample, showing the inputs and relevant forwarding table states that violate the specification.

5. **Failure cases may be considered.** Some properties may be designed to hold even when some links or routers in the network fail, so a verifier may reason about what occurs during a failure.

A number of factors make the control plane verification difficult. One of the difficulties of this is dealing with the large number of possible environments. The routers in a network can

receive route advertisements from each of its neighbors, and the combinations of routes that the network receives can greatly affect the route that is ultimately selected. This is because routing protocols choose a single best route for each prefix. This is different from the data plane, where the result of handling each packet is independent of the result of handling other packets. Another difficulty is that verifying the control plane requires reasoning about the result of routing protocols. As described below, routing protocols, especially BGP, can perform complex computation and can be configured in a number of different ways, complicating the task of control plane verification. Verifying the data plane, in contrast, is a much easier task. Each packet is forwarded independently, and while routers perform some computation using ACLs and forwarding tables, it is considerably simpler than the routing protocols and configured policies in the control plane.

1.1.2 Network Routing Protocols and BGP Operation

Network routing protocols are a major part of the control plane. During the routing process, each router learns routes to various different IP prefixes. That information is installed into the Routing Information Base (RIB) or routing table. Each entry in a RIB consists of a IP prefix, a next-hop, and other protocol-specific fields. This information is then used to generate the forwarding table, which has less information and faster processing times. Routes in the RIB can come from a few different sources. Each router initially installs a *connected route* for each destination reachable along an interface, and operators can configure *static routes* to be directly installed into the RIB. However, a large number of routes are learned through routing protocols.

The Internet is divided into thousands of autonomous systems (ASes), which are each managed separately. These are typically controlled by different organizations with their own management goals. Routing protocols either operate within a single AS or between different ASes. In both cases, routers announce information about the IP addresses that they can reach, and that information is used to determine how packets get forwarded. When

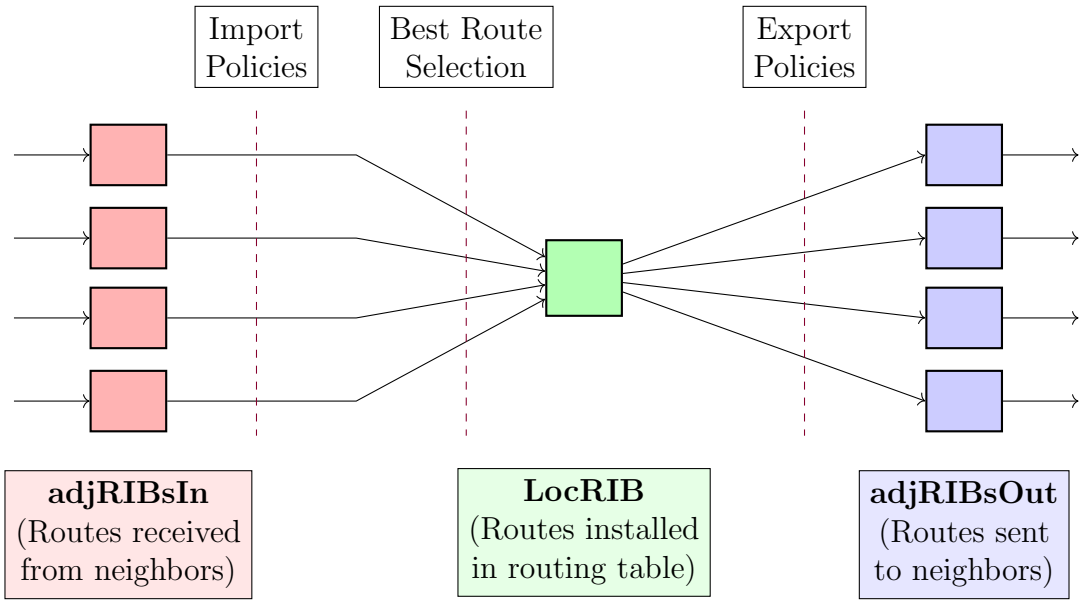


Figure 1.2: BGP operation in a single router.

routing within a single AS, networks typically use an interior gateway protocol such as Open Shortest Path First (OSPF) or Intermediate System to Intermediate System (IS-IS). These use a distributed algorithm such as link-state routing or distance-vector routing to compute the shortest path to each destination in the network.

Routing between ASes, on the other hand, uses the Border Gateway Protocol (BGP), which allows routers to route based on more complex policies. This allows organizations to have more control over how routes to and from other organizations are handled. In BGP, routers have neighbors and they are allowed to send route advertisements to those neighbors. In BGP, a route advertisement consists of a prefix along with a number of attributes such as next-hop, AS Path, local preference, multi-exit discriminator (MED), and communities. Each BGP speaker selects the best route advertisement it receives for each prefix and installs it into its RIB. The route selection is a standard procedure that depends on the attached route attributes. For example, the *AS path* attribute is a record of the ASes that the route has been propagated along, and shorter AS paths are preferred by the route selection procedure.

What makes BGP different from other protocols like OSPF is the ability for operators

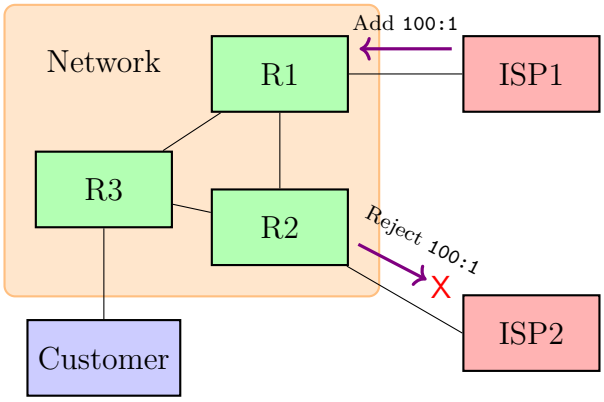


Figure 1.3: Using BGP to influence routing between ASes

to specify complex policies to influence route selection and route propagation[1]. Operators can specify *import policies* that are applied to each received route advertisement before a best route is selected, and they can specify *export policies* that can transform a route advertisement before it is sent to a neighbor. This is shown in figure 1.2. As defined in RFC4271[45], routers receive a route, apply the import policy, select the best route to be installed, and then apply the export policies before propagating the route to neighbors. Using these, operators can implement a variety of policies. For example, operators can specify policies to reject certain prefixes. They can influence route selection using the local preference or MED attributes. By assigning a higher local preference to routes from one neighbor and lower local preference to another, operators can specify that the former is preferred.

Figure 1.3 shows a more complex example of BGP policies that can be implemented. A standard policy that networks can implement is the no-transit policy, in which any routes learned from one ISP are not advertised to another while routes from customers can be advertised to either. This can be implemented in several ways. One possible way is to use the *community* attribute, which allows route advertisements to be tagged with 32-bit integer values, typically expressed as `A:B` where `A` is the upper 16 bits and `B` is the lower 16 bits.

[1]OSPF does allow operators to specify link costs and policies controlling redistributed routes, but these are typically not as complex as BGP policies.

The import and export policies can then be defined with the following behavior:

1. R1's import policy marks received routes from ISP1 with a BGP community `100:1`
2. R2's export policy filters routes tagged with `100:1` when advertising to ISP2, and
3. no other import or export policy strips community `100:1` from routes that it advertises.

In this case, the community `100:1` acts as a kind of state, signaling that this route should not be advertised to ISP2. Note that the above mechanism depends on policies at multiple locations in the network, and it only works assuming other routers in the network act correctly.

In summary, routing with BGP depends on the route advertisements that a router receives from its neighbors and the import and export policies that the operators specify. Together, these determine the routes that are installed into the routing table, which ultimately determines forwarding behavior. Operators can configure import and export policies to customize the way that they handle routes coming from other ASes. This gives BGP a lot of power and flexibility. However, this flexibility comes with a significant increase in the complexity of configurations. The fact that BGP is also used to share critical routing information across the Internet means that BGP configuration errors can have devastating consequences.

1.1.3 Router Configurations

The routing protocols that compute routes in the Internet are standard, but operators can influence the results of these protocols through configurations. Routers can be configured issuing text commands into the router's command-line interface or by loading a configuration from a file.

Figure 1.4 shows an excerpt of a configuration written in the Cisco IOS format. This example illustrates some of the flexibility and complexity involved in configuring BGP. In

this excerpt, lines 1-12 define the BGP policy. Line 2 sets the router ID which uniquely identifies this BGP speaker in the network. Lines 3-6 declare two neighbors `10.12.11.1` and `10.12.11.3` as part of the same `peer-group` with similar policy named `as1`. Lines 9-11 declare the policies that are applied to those neighbors for IPv4 routes. Specifically, the prefix `1.0.0.0/8` is announced, communities are sent to the neighbors, and the route map `POL` is the import filter applied on all routes received from the two neighbors.

The route map `POL` is defined in lines 22-27 and refers to other named structures defined in lines 16-20. It defines three clauses that are executed in order. The first rejects routes with prefixes matching the prefix list `NETS` (defined in lines 16-17). The second rejects routes with communities match the community list `COMMS` (defined in lines 19-20), and the last clause accepts all other routes after setting the local preference value to 30.

There are a few things to note about this example. First is the number of different pieces necessary to get a working configuration. Just focusing on the BGP policy, operators need to correctly configure a number of route maps, prefix lists, and other structures while applying them to the correct neighbors. This involves specifying many different constants for IP addresses, prefixes, and community values. Combining this with the configuration of static routes, other routing protocols, and other features results in configurations that are hundreds of lines long defining fairly complex behavior. Given that a large network may have hundreds of routers, it is no wonder that configuration mistakes will occur.

Another point to note is that while the Cisco configuration language cannot describe arbitrary computation the way that a general-purpose programming language can, configurations do follow a structure that is in some respects similar to programs. Prefix lists, community lists, and route maps can be thought of as functions that can be applied at multiple locations.

Lastly, different vendors offer different configuration languages. Figure 1.5 shows an example of BGP configuration for Juniper routers running JUNOS. It uses a noticeably different syntax from the Cisco configuration format, but still allows many of the same

```
1  router bgp 2
2   bgp router-id 2.1.1.1
3   neighbor as1 peer-group
4   neighbor as1 remote-as 1
5   neighbor 10.12.11.1 peer-group as1
6   neighbor 10.12.11.3 peer-group as1
7  !
8   address-family ipv4
9    network 1.0.0.0 mask 0.255.255.255
10   neighbor as1 send-community
11   neighbor as1 route-map POL in
12   neighbor 10.12.11.1 activate
13   neighbor 10.12.11.3 activate
14  exit-address-family
15 !
16 ip prefix-list NETS permit 10.9.0.0/16 le 32
17 ip prefix-list NETS permit 10.100.0.0/16 le 32
18 !
19 ip community-list standard COMM permit 10:10
20 ip community-list standard COMM permit 10:11
21 !
22 route-map POL deny 10
23  match ip address NETS
24 route-map POL deny 20
25  match community COMM
26 route-map POL permit 30
27  set local-preference 30
```

Figure 1.4: Cisco BGP configuration example

features specifying neighbor properties and applying import polices. There are features supported by some vendors or routers and not supported by others, and there can be subtle differences between the different configuration formats. For example, Cisco routers do not send communities unless configured to do so (line 10 in figure 1.4), whereas Juniper routers send communities by default. These differences can result in configuration errors as well.

1.2 Configuration Errors and Verification

1.2.1 Real-world Error

As with programming languages, it is easy for operators to make mistakes when configuring networks, and the mistakes can have fairly large impact, potentially taking down Internet

```
protocols {
    bgp {
        group AS1 {
            type external;
            peer-as 1;
            local-address 2.1.1.1;
            family inet {
                unicast;
            }
            import POL;
            local-as 2;
            neighbor 10.12.11.1;
            neighbor 10.12.11.3;
        }
    }
}
```

Figure 1.5: Juniper BGP configuration example

services. On June 21, 2022, Cloudflare suffered an outage that disrupted traffic in 19 of their data centers for more than half an hour. These 19 data centers were repsonsible for a significant portion of their global traffic. A number of other websites and services rely on Cloudflare, and many of these were impacted as well. This incident was detailed in a blog post [47]. Because of an error during a configuration change, much of their services were brought down, and it took over an hour to completely recover.

In their description of what occurred, they stated that they were modifying their BGP configurations to standardize the communities being attached to the routes that they advertise within their network. Unfortunately, when they were creating the change, they accidentally reordered some of the clauses in one of their policies. Just like how `POL` in the figure 1.4 contains multiple clauses that are executed in order, their policy contained multiple clauses that allow different sets of prefixes to get advertised with different attributes. When their clauses were reordered, this resulted in many routers losing connectivity to a number of locations in their network.

This is a fairly simple error, but it was not caught by their peer review system or any of the operation procedures that they followed. As a result, it managed to disrupt global

Internet traffic. This is not an isolated incident. Errors from BGP configuration were the cause of numerous other Internet outages. The goal of control plane verification is to be able to catch these types of errors before they occur using check on the configurations.

1.2.2 Control Plane Verification Challenges

There are many technical challenges in designing a practical control plane verifier. These include the modeling the complexity of the routing process, scaling to the large number of routers and policies used in real networks, and localizing errors to their location in the configurations.

Complexity of routing: A verifier has to model the unique behavior of routing protocols. As described in section 1.1, in BGP, a router considers the messages it receives from each of its neighbors and selects a best route based on the configured policies and the BGP route selection mechanism. This best route then gets propagated to all the router's neighbors. While individual import and export policies are simpler than traditional programs, the route propagation makes it difficult to reason about the order of computation, and the best route selection process makes it more difficult to guarantee that a particular route will reach its destination, since it is possible that the route was not chosen as a best route. The combination of these factors makes the behavior of routing in the whole network complex. In fact, it has been shown that BGP is Turing complete [14] in that it is possible to construct networks that can perform arbitrary computation.

Scaling to large networks: Another challenge is that verifiers must be able to scale to large networks. This is difficult because verifiers must reason about all inputs and the states of the network that can result from them, and the space of all inputs and network states is extraordinarily large. A single network may have several neighbors, each sending route advertisements with a number of different attributes. All of these may get propagated through the network in varying degrees, resulting in different resulting states at each router. Verifying the control plane requires searching through all of these inputs and resulting states

and finding ones that violate a particular specification. For larger networks with hundreds of routers, it is impossible to naively search through this state space, so a more intelligent approach is needed in order to scale control plane verification.

Localizing errors: In order to be usable to operators, a control plane verifier has to provide actionable results. The goal of verification is to find errors in configurations without running them on real routers. Thus, if a network does not meet its specification, it is not enough to indicate that that it does not work. Ideally, verifiers should be able to point the user in the right direction by indicating which router configuration and which component the error occurs in, for example, indicating the route maps, ACLs, or other configuration components that resulted in the erroneous behaviour. Pointing out the exact cause of every error is likely not possible, especially given the large number of components in the network, but narrowing down the potential locations and providing additional information about the error is certainly necessary for any practical verification system.

1.2.3 Previous Control Plane Verification Tools

Several previous tools have been created to verify the network control plane. Each of them tackles the previously mentioned challenges in different ways, but each of them makes compromises, limiting their usefulness. Some are limited to fixed environments or are limited in terms of the properties they can prove, and none can scale to large networks while verifying a variety of properties and localizing errors to configurations.

One approach to reasoning about the control plane was to use simulation. Batfish [19] simulates routing protocols for a fixed environment, that is, one particular set of incoming route announcements. Batfish produces forwarding tables which can then be used with data plane verification techniques. In a similar vein, Microsoft's Crystalnet [34] emulates networks using real device firmware, allowing for greater accuracy when simulating network changes before they occur. This approach can scale but it can only handle a fixed environment, so it cannot show correctness for arbitrary environments. Thus, it is a form of testing rather

than verification as it cannot handle all possible inputs.

Another approach is symbolic modeling. Existing tools that use this method can provide strong guarantees, checking specifications for all inputs and environments, but have difficulty scaling to large networks. Minesweeper [6] and Bagpipe [53] use SMT approaches to model the control plane, allowing them to consider different environments and reason in the presence of link failures. Other approaches [20, 42, 3, 7] build on these techniques or use alternative techniques to speed up computation. However these tools also have difficulty scaling since they model the network holistically, or they use optimizations that prevent them from reasoning about arbitrary environments (figure 3.1).

Moreover, these control plane analyzers typically do not produce easily usable results. Minesweeper, for example, has two key limitations. Since it uses an SMT-based approach, it only returns a single counterexample if there is an error. Second, the provided counterexample consists of a concrete packet whose forwarding exhibits a behavioral difference in the two configurations, leaving to the operator the difficult tasks of identifying the set of packets that is impacted and the specific configuration lines that caused the difference. In other words, Minesweeper cannot localize errors. It is even difficult to distinguish between network configuration errors and limitations in Minesweeper's modeling. Other tools similarly model the network as a whole and do not provide any additional capabilities for localizing errors, so they have the same problems.

In summary, no previous tool can check a variety of control plane properties while scaling and localizing errors to the configuration lines that cause them.

1.3 Thesis Statement and Contributions

In this work, I propose new techniques that allow for checking control plane properties while being scalable and allowing for localization of errors. My thesis is that modular checking, which takes advantage of the way operators structure and design networks, can be used to scale control plane verification and localize errors.

1.3.1 Structure within Networks

The key insight is that networks are configured in a structured manner. This is what allows networks to be manageable despite their size and complexity. Leveraging this structure allows for more efficient and usable verification. Some of the ways that configurations are organized include:

1. **Components within configurations**: Configurations are typically organized into components responsible for different aspects of the router's behavior. For example, BGP, OSPF, static routes, route maps, and prefix lists are all configured separately, though the behavior of some of these may depend on others. Most previous verifiers have to deal with this form of structure when they parse the configurations, but they often discard it after creating a model of the network.

2. **Roles within a network**: Routers fulfil specific roles in the network, and usually there are many routers that have the same or similar roles. For example, large networks have multiple border routers that connect to routers in other networks and filter incoming traffic, and they have multiple core routers with large routing tables that handle routing within the network, sending traffic in the correct direction. All of the routers that fulfil a specific role will have similar but not identical configuration.

3. **Modularity of policy**: Operators want to implement policies with their configurations, such as the no-transit policy shown in section 1.1. These policies depend on the interaction of multiple routers in different locations. To handle these, part of a policy has to be implemented in one location, and a separate part of the policy has to be implemented in another location. Thus, there is a relationship between the local policies at different locations in the network even when they have different roles and different local behavior.

In the rest of this thesis, I describe some of the techniques I developed for exploiting the

structure of network configurations.

1.3.2 Localizing Router Differences

In chapter 2, I focus on how to localize errors in the specific case of comparing two router configurations that are intended to be behaviorally equivalent. This task arises often in large networks. First, it is common for pairs of routers from different manufacturers to serve as backups for one another in case of failure. Whenever one router in the pair is updated, the other must be consistently updated, which is non-trivial if they use different configuration formats. A second important use case is *router replacement*. Routers are periodically upgraded from one manufacturer (e.g., Juniper) to one another (e.g., Arista) with better features, cost, or performance.

Previous tools for network control-plane verification, such as Minesweeper [6], can be used to verify behavioral equivalence of two router configurations. However, while these tools can detect equivalence violations, they provide very little help in debugging such errors. Existing tools have two key limitations that my work aims to address. First, they cannot identify the scope of the error since they can only produce a single counterexample. Second, they cannot identify the lines of the configuration that cause the error. When they provide a counterexample, it consists of a concrete packet whose forwarding exhibits a behavioral difference in the two configurations, leaving to the operator the difficult tasks of identifying the configuration lines that caused the difference. Handling these two limitations requires *localization* to relevant message headers and configuration texts.

To allow for better localization, I introduce the novel approach of comparing router behavior modularly at the level of configuration components. This was implemented in the tool Campion. Instead of modeling entire routers monolithically, corresponding router configuration components, such route maps, access control lists (ACLs), OSPF costs, are compared. This immediately improves localization since any difference found can immediately be traced to the relevant component.

However the localization and efficiency is improved further by considering the structures and behaviors of specific configuration components. Many configuration components have the property that any *structural* difference implies a possible behavioral difference. For example, two OSPF link costs are only guaranteed to be behaviorally equivalent, for all possible configurations, if they are identical. For these configuration components, I compare them with a simple *structural equivalence* check.

A few other configuration components, specifically ACLs and route maps, encode sophisticated policies, so there are many possible structures for the same behavior, especially when considering multiple vendors. For these configuration components, I compare them with a *semantic equivalence* check. To identify *all* differences, I model the two components C_1 and C_2 as small programs (e.g., an ACL is a program taking a packet and returning a boolean) and consider the *paths* in C_1 and C_2 that each input may take. If some input results in the execution paths p_1 in C_1 and p_2 in C_2 and these result in different behavior, then I can return a difference with the relevant paths that caused that difference. This algorithm is conceptually similar to prior approaches to checking equivalence in C functions [44] and network data planes [16]. To our knowledge ours is the first approach that can precisely check equivalence of network control-plane structures, notably route maps.

Campion was evaluated on the network configurations of a large cloud provider and a large university campus. The operators of the cloud provider were in the process of replacing 30 Cisco routers with Juniper routers due to a corporate policy decision. This required them to manually translate the original Cisco IOS configurations to JunOS. They used Campion to proactively check equivalence, identifying four configuration errors that they fixed before they could cause service disruption, including one error that would have been a severe outage. In the university network, there were a pair of core routers and a pair of border routers from different device vendors and intended to be backups of one another. Campion identified and localized configuration errors across these two pairs. These errors have been present in the configurations for nearly three years, and the operators said that they were "highly

unlikely" to detect them by "just eyeballing the configs." Campion only takes a few seconds to compare a pair of routers.

1.3.3 Modular Verification of Network-wide Control Plane Properties

While Campion could provide fairly precise localization, it does so for the specific property of router equivalence, which does not need to consider how policies in one router interact with policies in another. In the subsequent chapters, I develop an approach to proving network wide BGP properties using modular checks. I characterize two classes of BGP network properties, one class consisting of safety properties and another consisting of liveness properties. I then show how these can be verified using a combination of local checks on individual BGP policies. These techniques were implemented in the tool Lightyear.

Proving network-wide properties using local checks scales significantly better than previous approaches. The lack of scalability in previous tools is fundamentally caused by a shared limitation of earlier approaches: they model and reason about network behavior *monolithically*. As the size of the network grows, the number of possible network states grows exponentially, limiting their ability to scale. By contrast, verification has scaled to large systems in other domains, like software or hardware, through *modular* checking. In this style, subsystems (e.g., a software function or hardware module) are verified independently to meet *local* specifications (e.g., a precondition/postcondition pair) that together imply a desired global property [21, 26, 41]. Prior work has used modularity to scale data-plane analysis [23], but modularizing control-plane verification is more challenging due to complex routing protocols and policies.

Like prior verifiers, Lightyear takes as input a network's configuration and a *global* property to verify. To ensure the property, Lightyear additionally requires the user to provide local constraints that should hold on individual routers and edges. It then automatically produces a set of *local* checks on individual nodes and edges that 1. verify the user's local constraints and 2. ensure that these constraints imply the given end-to-end property.

To do so, it target two specific classes of properties: *safety* properties on individual routers, which intuitively ensure that "bad" routes never reach a particular node, and *liveness* properties, which intuitively ensure that a "good" route will eventually be accepted or forwarded at a particular location. The former includes common properties like filtering bogons, preventing transit between peers, and ensuring isolation. The latter includes many control-plane reachability queries: for example that a route received from one neighbor will be sent to another.

Lightyear's approach has the following benefits:

1. **Scalability:** Lightyear performs a *linear* number of checks in the network size (number of nodes and edges). Each check's runtime depends only on the complexity of an individual node's configured policy. Hence Lightyear scales roughly linearly with network size. Prior approaches that reason about the *joint* behavior of all nodes' policies scale at least *quadratically*, if not *exponentially*. Lightyear's local checks are also trivially parallelizable and enable incremental re-checking when configurations change.

2. **Strong Guarantees:** If all of Lightyears's local checks are satisfied, then the specified network property is guaranteed to hold for all possible external route announcements from neighbors. Further, for safety properties our guarantees will hold even in the presence of arbitrary node or link failures, though this is not true in general for liveness properties.

3. **Localization:** While prior approaches identify incorrect behavior, the resulting counterexample is a *global* snapshot, making it difficult to determine which router and policy is erroneous. By contrast, a local-check violation in Lightyear directly indicates the erroneous router and policy.

Lightyear's main tradeoff is that users must specify local constraints. However, this does not make the task of creating specifications significantly more difficult. Most networks are

designed in a modular and structured fashion, so many desired end-to-end properties can only require a few simple constraints. Network nodes are commonly partitioned into *roles*, such as border or core, each with its own responsibilities, and nodes in the same role will typically have similar local constraints.

Chapter 3 describes how safety properties are checked. To bridge between checks on individual routers and network properties, I introduce the idea of network invariants, which are similar to loop invariants used in program verification. The central idea is that BGP configurations establish and maintain an invariant for certain parts of the network. By specifying the invariants for the network, it is possible to determine the checks needed to prove safety properties in the network.

Chapter 4 describes how liveness properties are checked. These are more difficult to reason about than safety properties, since they require showing that routes will get propagated through the network. This requires reasoning about the best routes selected at each router. Building on the framework in chapter 3, I show the constraints that need to be provided and the checks that need to be performed.

Chapter 5 shows the results from evaluating Lightyear. Lightyear was used to verify multiple properties for BGP in a large cloud provider's wide-area network, which has hundreds of routers and tens of thousands of BGP peerings. To my knowledge no prior verification tool that reasons about all possible external route announcements has been demonstrated at this scale. I also ran tests on synthetic networks to show how well Lightyear scales.

1.4 Comments

The work in this thesis is a revised and extended presentation of research developed through collaborative work. Chapter 2 presents research on modular equivalence is based on prior work presented at SIGCOMM 2021 [48]. The work presented in chapters 3, 4, and 5 is based on work that has been accepted at SIGCOMM 2023.

CHAPTER 2

Checking Modular Equivalence of Routers

This chapter presents an approach for debugging two router configurations that are intended to be equivalent. This is a comparatively simple task, but it has common applications in large networks. First, this can be used to check correctness of backup routers which should have the same behavior but may be configured from different manufacturers. These have to be consistently updated which can be easy to difficult when they have different configuration formats. Second, routers are often upgraded or replaced from one vendor to another for features, cost, or performance. This requires manual rewriting of configurations, which is difficult and can result in configuration errors. Third, configurations are often updated, but updates are usually limited in scope. In this case, the behavior difference is usually minimal. All cases require a tool that can check equivalence and report differences between router behaviors.

Existing tools for network control-plane verification, such as Minesweeper [6], can be used to verify behavioral equivalence of two router configurations. However, while these tools can *detect* equivalence violations, they provide very little help in *debugging* such errors. In particular, existing tools have two key limitations that my work aims to address. First, they provide only a *single* counterexample and hence identify only a single behavioral difference between the two configurations. Second, the provided counterexample consists of a concrete packet whose forwarding exhibits a behavioral difference in the two configurations, leaving to the operator the difficult tasks of identifying the *set of packets* that is impacted and the specific *configuration lines* that caused the difference. I call the first challenge *header*

localization and the second *text localization*.

This chapter presents the tool Campion, which performs localization through a novel approach. Rather than representing the behavior of each router configuration *monolithically*, for example as a set of SMT constraints [6], Campion compares pairs of corresponding components between the two configurations (route maps, ACLs, OSPF costs, etc., see table 2.1) separately. Performing equivalence checks on a per-component basis immediately helps: every pair of components that are not behaviorally equivalent is reported, and each such violation is by construction localized to the relevant configuration components.

In the context of modular checking, two configuration components C_1 and C_2 are considered equivalent if *any* configuration containing C_1 could instead use C_2 without changing the configuration's behavior. How should each pair of components be checked for equivalence? Observe that there are two distinct types of configuration components from the point of view of modular checking.

Many configuration components have the property that any *structural* difference implies a possible behavioral difference. For example, two OSPF link costs are only guaranteed to be behaviorally equivalent, for all possible configurations, if they are identical. The same is true for static routes in two configurations. These configuration components can be compared with a simple *structural equivalence* check that I call StructuralDiff. This check is efficient, reports and localizes all behavioral differences — all structural mismatches — and makes it trivial for users to understand the error.

On the other hand, a few configuration components, specifically ACLs and route maps, encode complex behaviors. There are many possible ways to write an ACL or route map for the same behavior, especially when considering multiple vendors. For example, Juniper and Cisco route maps are structured in very different ways. For these configuration components, I compare them with a *semantic equivalence* check that I call SemanticDiff. To identify *all* differences semantically, I model the two components C_1 and C_2 as small programs. For example, an ACL can be thought of as a program that takes a packet header and returns a

boolean for whether the packet is dropped or not, and there is a form of control flow in the match statements, which branch depending on values in the header. Using static analysis to find behavioral differences, the tool considers each path p_1 through C_1 and p_2 through C_2 and checks whether there is some input that traverses along p_1 and p_2 through their respective components and exhibits a behavioral difference. If there is a difference, this result can be returned along with the relevant paths that produced these behaviors, allowing operators to see which lines of configuration affected the difference. This algorithm is conceptually similar to prior approaches to checking equivalence in C functions [44] and network data planes [16]. To my knowledge this is the first approach that can precisely check equivalence of network control-plane structures, notably route maps.

The SemanticDiff algorithm localizes each behavioral difference to a specific path through each component. To help users understand the difference, I also introduce a novel algorithm called HeaderLocalize that localizes each difference to the relevant space of inputs. In the implementation, SemanticDiff produces the impacted set of inputs I as a binary decision diagram (BDD). Given this BDD and the original configurations, HeaderLocalize produces a representation of all destination IP addresses in I in terms of the constants (prefixes or prefix ranges) that appear in the configurations, and does so in a minimal way.

Perhaps surprisingly, Campion is *protocol-free*: it does not need to model routing protocols like BGP and OSPF. my modular approach obviates the need for such reasoning, as equivalence of each corresponding pair of configuration components implies that those protocols will behave identically on the two routers. I formally prove this theorem, thereby justifying my approach. A potential downside of my modular approach is that it can produce false positives: it is possible for two configuration components to cause a behavioral difference for *some* configuration, and hence be flagged as erroneous by Campion, but still be behaviorally equivalent in the context of the two given router configurations. However, my experiments indicate that false positives are rare. Intuitively this makes sense because

Feature	Check Used
ACLs	SemanticDiff
Route Maps (BGP, Route Redistribution)	SemanticDiff
Static Routes	StructuralDiff
Connected Routes	StructuralDiff
Other BGP Properties	StructuralDiff
OSPF Properties (costs, areas, etc.)	StructuralDiff
Administrative Distances	StructuralDiff

Table 2.1: Components supported by Campion and the check used for each.

configurations are created and maintained in a modular fashion, with different aspects of the configuration responsible for different aspects of the behavior. Moreover, for management reasons, it may even be desirable to prefer clear correspondences between the components, so even a false positive may still be of interest to operators.

I evaluated Campion on the network configurations of a large cloud provider and a large university campus. First, the operators of the cloud provider were in the process of replacing 30 Cisco routers with Juniper routers due to a corporate policy decision. This required them to manually translate the original Cisco IOS configurations to JunOS. They used Campion to proactively check equivalence, identifying four configuration errors that they fixed before they could cause service disruption, including one error that would have been a severe outage. Second, the university network has a pair of core routers and a pair of border routers from different device vendors and intended to be backups of one another. Campion identified and localized configuration errors across these two pairs. These errors have been present in the configurations for nearly three years, and the operators said that they were "highly unlikely" to detect them by "just eyeballing the configs." Campion only takes a few seconds to compare a pair of routers.

To summarize, the contributions of this chapter are:

- A modular approach that identifies *all* behavioral differences between two configurations and localizes them to the relevant configuration lines (section 2.2). For each con-

figuration component, I determine whether a full semantic analysis (SemanticDiff) is needed or a simple structural equivalence check (StructuralDiff) suffices (see section 2.1). I also describe a novel algorithm for localizing the relevant inputs (HeaderLocalize).

- A theorem (section 2.2.4) that shows my modular approach to equivalence checking of configuration components suffices to ensure router behavioral equivalence, despite not reasoning about the network protocols.

- A tool, Campion (section 2.3), that localizes behavioral differences between router configurations. Campion supports all of the routing and forwarding components modeled by Minesweeper. Campion is available as open-source software.[1]

- An experimental evaluation of Campion on routers from a large cloud vendor and a university network.(section 2.4).

2.1 Example

This section shows two examples of Campion's output that identified behavioral differences in routers from a large university network. I present one case involving differences between BGP route maps, which Campion identified and localized using SemanticDiff and HeaderLocalize, and a second case involving differences in static routes, which Campion identified and localized using StructuralDiff. In both cases, I also demonstrate the advantages of Campion by comparing its output to that of Minesweeper [6], a state-of-the-art network configuration verification tool.

2.1.1 Route Map Diffs via Semantic Checks

Figure 2.1 shows simplified versions of route maps from two core routers in a large university network (see section 2.4.2). The two route maps are intended to be behaviorally identical,

[1] https://github.com/atang42/batfish/tree/rm-localize

with the first written for a Cisco router and the second for a Juniper router. Both configurations define a prefix list `NETS` to match a specific set of IP prefixes (lines 1-2 in figure 2.1a and 1-4 in figure 2.1b), as well as a community list `COMM` to match the community tags `10:10` and `10:11` (4-5 in figure 2.1a and 5 in figure 2.1b). The remainder of each snippet defines a route map `POL` for each router, which rejects route advertisements that match prefixes from `NETS` or are tagged with communities from `COMM` and accepts all other advertisements (7-12 in figure 2.1a and 6-21 in figure 2.1b).

Despite the superficial similarity of the two configurations, there are large behavioral differences. Campion uses SemanticDiff and HeaderLocalize to find and localize these differences. Table 2.2 shows Campion's output when given the two route maps in figure 2.1. The output has two results, each of which represents a distinct configuration error. For each error, Campion identifies *all* the route advertisement prefixes that are treated differently by the two route maps, namely route advertisements for prefixes that are in the set `Included Prefixes` but not the set `Excluded Prefixes`. I call the process of identifying and representing all problematic inputs *header localization*. Further, Campion also shows the action that each route map takes on these advertisements as well as the configuration lines responsible for that action. I call the process of identifying all relevant lines of the configuration *text localization*.

In the output shown in table 2.2a, the `Action` and `Text` rows indicate that advertisements for the relevant prefixes match the `NETS` prefix list in the Cisco route map and are therefore rejected, but these prefixes fall through to the last term in the Juniper route map and are accepted. Careful inspection reveals the problem: in the Cisco route map, `NETS` matches prefixes with lengths between 16 and 32, while in the Juniper route map it only matches prefixes with lengths of exactly 16. Thus, a prefix like `10.9.1.0/24` is matched by the Cisco route map but not by the Juniper route map.

The second result that Campion produces (table 2.2b) identifies a second, unrelated configuration difference. The `Included Prefixes` and `Excluded Prefixes` rows show that

	cisco_router	**juniper_router**
Included Prefixes	10.9.0.0/16 : 16-32 10.100.0.0/16 : 16-32	
Excluded Prefixes	10.9.0.0/16 : 16-16 10.100.0.0/16 : 16-16	
Policy Name	POL	POL
Action	REJECT	SET LOCAL PREF 30 ACCEPT

Text	route-map POL deny 10
	match ip address NETS
		rule3 {
		then {
		local-preference 30;
		accept;
		}
		}

(a) Difference 1

	cisco_router	**juniper_router**
Included Prefixes	0.0.0.0/0 : 0-32	
Excluded Prefixes	10.9.0.0/16 : 16-32 10.100.0.0/16 : 16-32	
Community	10:10	
Policy Name	POL	POL
Action	REJECT	SET LOCAL PREF 30 ACCEPT

Text	route-map POL deny 20
	match community COMM
		rule3 {
		then {
		local-preference 30;
		accept;
		}
		}

(b) Difference 2

Table 2.2: Campion result when checking equivalence of configurations in figure 2.1 using a Semantic Check

```
ip prefix-list NETS permit 10.9.0.0/16 le 32
ip prefix-list NETS permit 10.100.0.0/16 le 32
!
ip community-list standard COMM permit 10:10
ip community-list standard COMM permit 10:11
!
route-map POL deny 10
 match ip address NETS
route-map POL deny 20
 match community COMM
route-map POL permit 30
 set local-preference 30
```

(a) Excerpt from the Cisco route map

```
    prefix-list NETS {
        10.9.0.0/16;
        10.100.0.0/16;
    }
    community COMM members [10:10 10:11];
    policy-statement POL {
        term rule1 {
            from prefix-list NETS;
            then reject;
        }
        term rule2 {
            from community COMM;
            then reject;
        }
        term rule3 {
            then {
                local-preference 30;
                accept;
            }
        }
    }
```

(b) Excerpt from the Juniper route map

Figure 2.1: Cisco and Juniper route maps with subtle differences

this difference occurs for advertisements of all prefixes other than those in the ranges of the `NETS` prefix list. While Campion can find all differences and identify all relevant IP prefixes, for other fields of the route advertisement it currently provides a single example. In this case, the output indicates that this difference occurs when the route advertisement contains

Route received (Cisco)	Prefix: 10.9.0.0/17
Route received (Juniper)	Prefix: 10.9.0.0/17
Packet	dstIp: 10.9.0.0
Forwarding	Juniper router forwards (BGP) Cisco router does not forward

Table 2.3: Minesweeper result when checking equivalence of configurations from figure 2.1

only the community `10:10`. The `Action` and `Text` rows show that the Cisco route map matches the advertisement against the community list `COMM` and rejects it, while the Juniper route map again falls through to the last rule. This difference reveals a subtle error: `COMM` in the Cisco route map matches route advertisements containing *either* the community `10:10` or `10:11`, whereas `COMM` in the Juniper route map erroneously matches only advertisements tagged with *both* communities.

Campus network operators confirmed both of the above behavioral differences as configuration errors. Further, the errors are subtle and have existed since at least July 2017. The network operator commented, "your config-analysis tool is great. It's highly unlikely anyone would detect the functional discrepancies just by eyeballing the configs." As described in section 2.4.2, Campion found additional differences that have been removed here to keep the example simple.

Comparison with Minesweeper. Minesweeper [6] builds a logical representation of the network behavior, modeling the routing process and forwarding behavior. It then uses a satisfiability modulo theories (SMT) solver to answer verification queries. Minesweeper supports a behavioral equivalence check of individual routers, but it does so by checking that the logical representation of both routers' entire configurations are equivalent. A major drawback of this monolithic approach is the difficulty to diagnose the source of the error — any identified difference could be caused by BGP configuration, OSPF configuration, ACLs, or static routes.

In order to make the comparison more fair, I adapted Minesweeper to only check be-

havioral equivalence of two route maps. Specifically, Minesweeper checks that its logical representations of the two route maps are equivalent: whenever they receive the same set of route advertisements, they produce the same forwarding behavior for all packets. Table 2.3 shows the output of this modified version of Minesweeper on the above example. There is a single counterexample indicating that, when both routers receive a route advertisement with prefix `10.9.0.0/17`, they will produce different rules for forwarding packets with destination IP address `10.9.0.0`: the Juniper router will forward them, while the Cisco router will not.

Minesweeper's output identifies a behavioral difference between the two route maps that corresponds to Campion's output shown in table 2.2a. However, Minesweeper's output is lacking in several important ways. (1) It only provides information about a single behavioral difference. However, as explained earlier, there are actually two unrelated configuration differences between these route maps (table 2.2a and table 2.2b). (2) For the error that Minesweeper does identify, it only provides a single concrete example, with a specific route advertisement and destination IP prefix. To fully fix the problem of unintended differences between the two route maps, operators must understand the set of all route advertisements that produce this behavioral difference. Having this set explicitly also provides an indication of the scope of the problem. (3) Minesweeper provide no information about what parts of the route maps are responsible for the behavioral difference.

It is possible to modify Minesweeper again, this time to produce multiple concrete examples. This can be done by simply querying the SMT solver multiple times, each time including additional logical constraints that disallow previously generated counterexamples. This approach could potentially alleviate the first two issues described in the previous paragraph, but my experiments with this approach illustrate that it is not very effective. On the above example, running Minesweeper does provide counterexamples from both classes of differences from table 2.2 but it takes 7 counterexamples in order to have at least one for each prefix range that is relevant for Difference 1. Further, the approach is fragile: when I replaced the number 32 in the second line of the Cisco configuration (figure 2.1a) with 31,

	cisco_router	juniper_router
Prefix	10.1.1.2/31	
Next Hop	10.2.2.2	None
Admin. Distance	1	None
Text	ip route 10.1.1.2 255.255.255.254 10.2.2.2	None

Table 2.4: Campion result when checking equivalence of static routes using a Structural Check

it took 27 counterexamples for Minesweeper to provide a violation of Difference 1 instead of Difference 2.

2.1.2 Static Route Diffs via Structural Checks

Campion detects differences in configuration components such as static routes and OSPF costs using a structural equivalence check. For example, for static routes Campion simply considers the set of static routes in each router and identifies all structural differences: cases where a route is present in one set but not the other, or where a route is present in both but with different attributes such as the next hop and administrative distance. This technique illustrates another advantage of my modular approach. Because I are checking configuration components in isolation from the rest of the configurations, for many components a simple structural check is *as precise as* a behavioral check via a semantic representation, while providing better localization and understandability for users.

An example of an output produced by Campion when checking static routes is shown in table 2.4. This output shows that in the Cisco router, a static route exists that sends packets destined to `10.1.1.2/31` to `10.2.2.2`, but there is no such route in the Juniper router. Differences like this were found in both the university and cloud networks.

Table 2.5 shows the output that Minesweeper produces for the same example. Minesweeper can identify that the forwarding was caused by a static route, but it does not determine the prefix of the static route, the other relevant fields like the administrative distance, or the

Packet	dstIp: 10.1.1.2
Forwarding	Cisco router forwards (static) Juniper router does not forward

Table 2.5: Minesweeper result when checking equivalence of static routes

lines of the configuration. Hence operators have to search through a potentially large set of static routes and determine which one would affect the routing of packet to a `10.1.1.2`. Further, if there were multiple static-route differences, Minesweeper would only find one, while Campion would identify all.

2.2 Design and Algorithms

I describe Campion's design and core algorithms. Campion's overall algorithm for identifying and localizing behavioral differences between configurations C_1 and C_2 is as follows:

```
1  func ConfigDiff (C1, C2)
2      result ← [ ]
3      pairs ← MatchPolicies(C1, C2)
4      for (p1, p2) ∈ pairs do
5          differences ← Diff(p1, p2)
6          for d ∈ differences do
7              result ← result.append(Present(d, {C1, C2}))
8          end for
9      end for
10     return result
```

This algorithm consists of three main parts:

1. The corresponding components (ACLs or BGP route maps) for C_1 and C_2 are paired up by the MatchPolicies function. This can be done with heuristics such as matching components by name or matching components that relate to the same neighboring node, or this information can be provided by the user.
2. For each component pair, the Diff function invokes either SemanticDiff or StructuralDiff to produce a set of *differences*, each of which can include a set of inputs, the actions taken by each component, and the locations in the configurations.

3. The PRESENT function formats the results for output to the user, including invoking HEADERLOCALIZE on the results of SEMANTICDIFF in order to produce an understandable representation of the set of inputs.

I now describe SEMANTICDIFF, HEADERLOCALIZE, and STRUCTURALDIFF in more detail. I then discuss the general applicability of SEMANTICDIFF and STRUCTURALDIFF and show how my modular approach can find and localize behavioral differences across entire router configurations.

2.2.1 SemanticDiff

SEMANTICDIFF takes a pair of configuration components as input and returns a list of all behavioral differences. The same basic algorithm applies to both ACLs and route maps. Each difference is a quintuple of the form: (i, a_1, a_2, t_1, t_2). In this quintuple, i refers to a set of inputs to the components, represented as a logical formula over message headers. For dataplane ACLs the inputs are sets of packets, and for route maps they are route advertisements. a_1 and a_2 are the respective actions taken by the two components when given an input from i. The action for ACLs is either accept or reject, but for route maps the accept action can also set fields such as local preference. t_1 and t_2 are the respective lines of text from the two components that process inputs from i and result in a_1 and a_2.

The SEMANTICDIFF algorithm has two main steps. First, for each configuration component, the space of inputs is divided into equivalence classes, based on their *paths* through the component. Both ACLs and route maps can be viewed as a sequence of *if-then-else* statements, so two inputs are in the same equivalence class if and only if they take the same set of branches through these statements. Each equivalence class is represented symbolically as a logical predicate on the input (either a packet header or route advertisement). my implementation uses BDDs to represent these predicates. Each equivalence class is also associated with the text lines that are on the corresponding path as well as the action taken.

This step consequently produces two lists of triples:

$$L_1 = [(i_{1,1}, a_{1,1}, t_{1,1}),\ (i_{1,2}, a_{1,2}, t_{1,2}), \ldots,\ (i_{1,m}, a_{1,m}, t_{1,m})]$$

$$L_2 = [(i_{2,1}, a_{2,1}, t_{2,1}),\ (i_{2,2}, a_{2,2}, t_{2,2}), \ldots,\ (i_{2,m}, a_{2,m}, t_{2,m})]$$

Figure 2.2 shows the equivalence classes for the example route map from figure 2.1a. `NETS` and `COMM` correspond to the names of the attribute filters — `NETS` for prefix filters and `COMM` for communities. I use $[\![\texttt{NETS}]\!]$ to denote the set of accepted prefixes, and similarly $[\![\texttt{COMM}]\!]$ to denote the set of accepted communities. I also denote the complement of a set X as $\neg X$. There are three equivalence classes, one per clause in the route map — the first clause is associated with the space $[\![\texttt{NETS}]\!]$, the second clause is associated with $\neg [\![\texttt{NETS}]\!] \cap [\![\texttt{COMM}]\!]$, the space of routes matching $[\![\texttt{COMM}]\!]$ but not $[\![\texttt{NETS}]\!]$, and the third clause is for all remaining routes. Each equivalence class is also associated with whether it accepts or rejects routes and what fields are set.

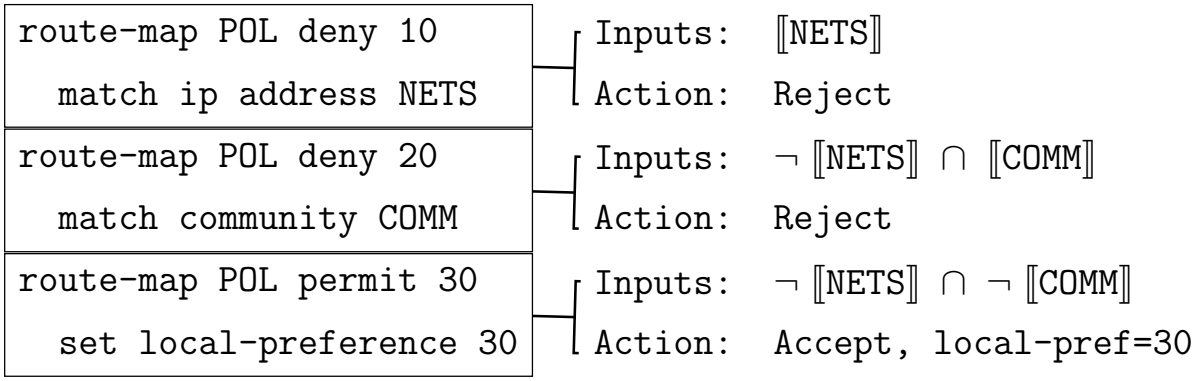


Figure 2.2: Partitioning the space of route advertisements based on route map definitions.

Once the inputs are partitioned into equivalence classes for both components, the SemanticDiff algorithm then performs a pairwise comparison to identify behavioral differences. For each pair of equivalence classes $(i_{1,i}, a_{1,i}, t_{1,i})$ and $(i_{2,j}, a_{2,j}, t_{2,j})$ from the two components, if $i_{1,i}$ and $i_{2,j}$ have a non-empty intersection and the actions $a_{1,i}$ and $a_{2,j}$ differ, then there is

a behavioral difference. In that case, I add

$$(i_{1,i} \cap i_{2,j},\ a_{1,i},\ a_{2,j},\ t_{1,i},\ t_{2,j})$$

to the list of differences returned by SemanticDiff.

2.2.2 HeaderLocalize

SemanticDiff produces the set of packets that exhibit behavioral differences as a logical predicate. The HeaderLocalize algorithm produces a more human-understandable representation in terms of the constants (e.g. IP prefixes) that appear in the configuration, handling the *header localization* problem. Specifically, HeaderLocalize produces a compact representation of the set of all destination IP addresses relevant to an ACL difference and the set of all IP prefix ranges relevant to a route map difference. For ease of presentation, I only describe finding prefix ranges relevant to route map differences, but the process for ACLs is analogous. In principle, HeaderLocalize can also be extended to other route fields such as communities, but I have not yet done so. Currently, instead of producing all communities relevant to a route map difference, Campion outputs a single example.

For route maps, sets of IP prefixes are represented by *prefix ranges*, each of which is a pair of a prefix and a range of lengths. For example, (`1.2.0.0/16, 16-32`) is a prefix range where the prefix is `1.2.0.0/16` and the length range is `16-32`. A prefix p is a member of a prefix range R if both of the following hold:

1. The IP address of p matches the prefix of R
2. The length of p is included inside the range of R

For example, `1.2.3.0/24` is a member of the prefix range (`1.2.0.0/16, 16-32`), (`0.0.0.0/0, 0-32`) is the set of all prefixes, and (`1.0.0.0/8, 24-24`) is the set of all prefixes with length 24 and 1 as the first octet. I say that a prefix range R_1 is contained in another prefix range R_2, denoted $R_1 \subset R_2$, if the members of R_1 are a subset of those of R_2.

The input to HeaderLocalize is a BDD S representing the set of messages affected by an identified policy difference, along with the original configurations C_1 and C_2. The output is a representation of S's prefix ranges in terms of the prefix ranges that are in the two configurations. First, all prefix ranges from the two configurations are extracted to get the set $\mathcal{R} = \{R_1, R_2, \ldots R_n\}$. If the set of all prefixes (`0.0.0.0/0, 0-32`), which I will call U, is not in $\mathcal{R}$, then it is added. Furthermore, $\mathcal{R}$ is extended so that it is closed under intersection. Since each line of a route map can allow or reject route advertisements based on prefix ranges in the configuration, it is always possible to represent the set S as a combination of complements, unions, and intersection of sets from $\mathcal{R}$. The goal of HeaderLocalize is to identify the minimal such representation.

To find this minimal representation, HeaderLocalize builds a directed acyclic graph (DAG) that relates the prefix ranges in $\mathcal{R}$ to one another. This data structure is analogous to the ddNF data structure previously used for packet header spaces [13], but here I associate each node with prefix ranges rather than tri-state bit vectors representing data-plane packets. HeaderLocalize's ddNF data structure consists of a set of nodes N, a set of edges $E \subseteq N \times N$, a labeling function l mapping nodes to prefix ranges, and a root node. It satisfies the following properties:

1. The root node is labeled with U, the set of all prefixes, and all other nodes are reachable from it.
2. Each node has a unique label (and thus in the following explanation, I will sometimes refer to a node by its prefix range or vice versa).
3. The set of prefix ranges used as labels contains $\mathcal{R}$ and is closed under intersection.
4. For any nodes $m, n \in N$, there is an edge $(m, n) \in E$ exactly when $l(n) \subset l(m)$ and there is no node m' such that $l(n) \subset l(m') \subset l(m)$.

An example DAG is shown in figure 2.3 for a set of seven prefix ranges. There is one node per prefix range, and each node's prefix range is a subset of those of its ancestors. For

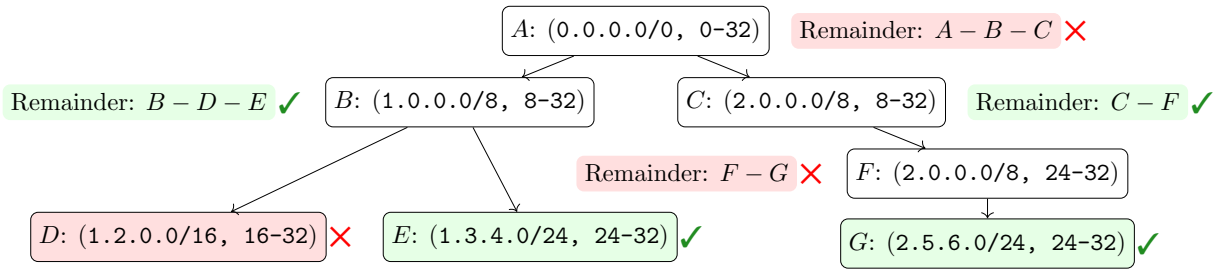


Figure 2.3: DAG created from prefix ranges. Green (✓) nodes represent leaves or remainders contained in a set S, and red (✗) nodes represent those that are not. S can be represented by the union of $B - D$, $C - F$, and G

example D is contained in B and A. The DAG is built by inserting one prefix range at a time, starting with U [13]. I also associate each internal node, with prefix range R and outgoing edges to nodes labeled $C_1, C_2, \ldots, C_k$, with the set of prefixes $R - C_1 - C_2 \cdots - C_k$. I call this set the *remainder* set, as it is the set of prefixes that remain in R after prefixes of the children nodes are removed. For example, the remainder set of node B in figure 2.3 is $B - D - E$. The remainder and leaf node sets are all disjoint from one another, and their union is U. Importantly, because the set S of interest was created through unions, intersections, and complements of the prefix ranges in $\mathcal{R}$, each remainder set and leaf prefix range has the property that either it is contained in S or disjoint from S.

Next HeaderLocalize uses the DAG to produce a representation of S in terms of the prefix ranges in $\mathcal{R}$. This is done by traversing the DAG with the recursive function GetMatch shown below. If the current node is a leaf, then its prefix range R is included in the result if that range is contained in S. If the current node is internal, then there are two cases. If the node's remainder is contained in S, then its prefix range R should be included in the result, after removing any of the node's child prefixes in the DAG that are not contained in S. This latter process is done through a recursive call to GetMatch with the complement set of S. if the node's remainder is not contained in S, then I simply recurse on the children and union the results.

The GetMatch algorithm produces a representation of S that is a union of terms of

```
1  func GetMatch (S, node)
2      C ← Children(node)
3      R ← PrefixRange(node)
4      if IsLeaf(node) then
5          if R ⊆ S then
6              return {R}                          ▶ node is a leaf, and R ⊆ S
7          else
8              return ∅                            ▶ node is a leaf, and R ∩ S = ∅
9          end if
10     end if
11     if Remainder(node, C) ⊆ S then              ▶ checks if R − C1 − C2 ... Ck ⊆ S
12         nonmatches ← ⋃_{k∈C} GetMatch(¬ S, k)
13         return {R − nonmatches}                 ▶ returns {R − X1 − X2 ... Xm}
14     else
15         return ⋃_{k∈C} GetMatch(S, k)           ▶ returns {X1, X2 ... Xn}
16     end if
```

the form $R - X_1 - X_2 - \ldots X_k$, where R is a prefix range, but each X_j is also in the form $R - X_1 - X_2 - \ldots X_k$. For example, running GetMatch on the DAG in figure 2.3 produces $\{B - D, C - (F - G)\}$, and the nodes in the figure are colored to illustrate the algorithm's process. As a final simplification step, I remove all *nested differences* from the result through a single pass over it. In my example, the result $C - (F - G)$ is transformed into $\{C - F, G\}$, so the final representation of the set S is $\{B - D, C - F, G\}$.

2.2.3 StructuralDiff

It would be possible to use a semantic approach like SemanticDiff to reason about all configuration components, just as I do for route maps and ACLs. However, I observe that other configuration components typically have a very stylized structure, as a single atomic value (e.g., integer or boolean) or a simple collection of such values. Hence, when considered modularly, the equivalence of two such components is tied to their structure.

That is, two components are behaviorally equivalent, for all possible configurations, if and only if their structural representation is identical. Thus I can use a simple structural check without incurring additional false positives versus a semantic approach. Since the structural

approach does not require logical modeling, it is more efficient. Further, localization is trivial since the structural check directly identifies the portions of the two components that differ.

my StructuralDiff function implements this approach. All components are represented as atomic values, tuples, or unordered sets. Atomic values are tested for equality. Tuples are compared by testing that the corresponding values are equal. Finally, sets are compared using set difference.

For example, to check two OSPF configurations are equivalent (excluding route redistribution which is handled by SemanticDiff), it suffices to check equivalence for all corresponding attributes on all corresponding links. That means both routers must have OSPF edges to the same peers, and the corresponding edges are configured with the same costs, areas, passive status, etc. I can think of the configuration of each OSPF link as a tuple of its configured attributes and check each corresponding attribute. The same approach works for BGP properties not implemented with route maps, such as which edges are to route reflector clients and whether communities are propagated.

Other components that affect routing include connected and static routes. Connected routes are formed by the set of subnets connected to the router's interfaces, and the difference between routers is the set of such subnets present in one router but not the other. Similarly, a single static route can be represented as a tuple consisting of a destination prefix, a next-hop, an administrative distance, and optional fields like tags; so the difference is the set of tuples present in one router but not the other. Administrative distances can also be compared as values configured per protocol.

As mentioned earlier, localization for these components is straightforward because the equivalence check is performed directly on the components' structures. Further, unlike route maps and ACLs, these components have no explicit notion of input. Hence there is no need for, or analogue to, HeaderLocalize for such differences.

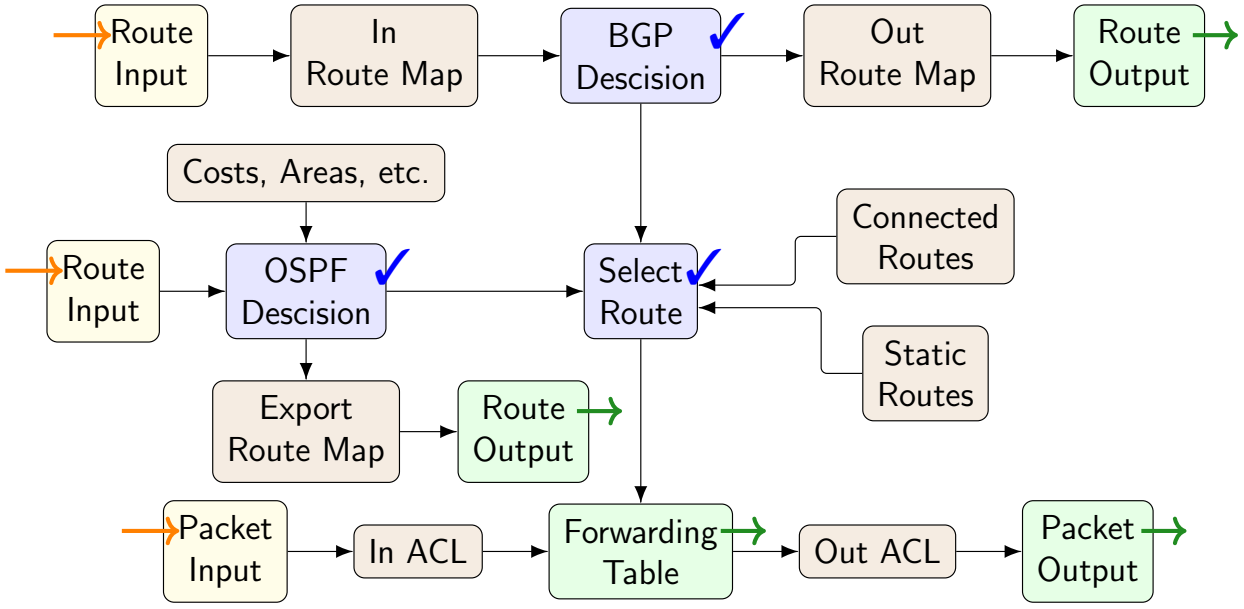


Figure 2.4: Basic features of routing and forwarding. Blue nodes(✓) represent fixed processes. Yellow nodes (incoming →) are inputs and green nodes (outgoing →) are outputs. Unmarked (brown) nodes represent configurable entities.

2.2.4 Debugging an Entire Router

I now formalize my approach to checking full router equivalence. I observe that many crucial parts of routing, such as the route selection process, are fixed. They are implemented according to a standard and depend only on the provided inputs and configurations. All of the various processes in figure 2.4 need to be modeled to fully simulate a router or network, but only the configured aspects (shown in brown) need to be modeled to find behavioral differences.

Figure 2.4 provides a flow diagram illustrating the processes supported by Campion. For routing, there is both a BGP process (top of figure) and an OSPF process (middle of figure), as these are the most common inter-domain and intra-domain routing protocols; other protocols could be added similarly. The bottom of the figure shows the router's process for forwarding routes. The brown (unmarked) nodes represent parts of the router configuration, while the other components are fixed processes like routing protocols (in blue (✓)), or input

routes and packets (in yellow (incoming →)), or outputs and byproducts like selected routes and forwarded packets (in green (outgoing →)).

Assuming that these are the only routing components used in the configurations being compared, then Campion is a sound verifier for router configuration equivalence: If Campion identifies no differences, then the two router configurations are behaviorally equivalent. I formalize the fact that behavioral equivalence can be verified without reasoning about the routing protocols as follows (my formalization considers behavioral equivalence of entire networks, but it therefore also applies to the special case of individual routers).

Definition 1. *A network* $\mathcal{N} = (T, \mathcal{R}, \mathcal{C}_\mathcal{P}, \mathcal{F}_\mathcal{P}, \preceq_\mathcal{P})$ *is a topology* $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ *of vertices and edges, a set of routes* $\mathcal{R}$*, a family of configuration functions* $\mathcal{C}_\mathcal{P} : \mathcal{E} \to \Omega$ *that maps each edge in the topology to a configuration* Ω*, a family of transfer functions* $\mathcal{F}_\mathcal{P} : \Omega \times \mathcal{E} \times \mathcal{R} \to \mathcal{R}$ *that transforms a route along an edge for a protocol, and a protocol preference relation* $\preceq_\mathcal{P}$*:* $\mathcal{R} \times \mathcal{R}$ *that compares two routes for a protocol.*

Definition 2. *For two networks* $\mathcal{N} = (T, \mathcal{R}, \mathcal{C}_\mathcal{P}, \mathcal{F}_\mathcal{P}, \preceq_\mathcal{P})$ *and* $\mathcal{N}^* = (T^*, \mathcal{R}, \mathcal{C}^*_\mathcal{P}, \mathcal{F}^*_\mathcal{P}, \preceq_\mathcal{P})$ *and an isomorphism* $\mathcal{I}$ *between* $\mathcal{T}$ *and* $\mathcal{T}^*$*, I say that the two networks are locally equivalent if for all protocols* $p \in \mathcal{P}$*, edges* $e \in \mathcal{E}$*, and routes* $r \in \mathcal{R}$ *then* $\mathcal{F}_p(\mathcal{C}_p(e), e, r) = \mathcal{F}^*_p(\mathcal{C}^*_p(I(e)), I(e), r)$.

[Soundness] If networks $\mathcal{N}$ and $\mathcal{N}^*$ are locally equivalent for isomorphism $\mathcal{I}$, then they have the same set of routing solutions.

Proof. The proof is by a reduction to the stable routing problem [7] and is described in the appendix. □

2.3 Implementation and Limitations

Campion operates on a vendor-independent representation produced by Batfish [19]. Real routers support an enormous number of features. For Campion, I have focused on the most common components used for routing and forwarding. Campion currently supports all of the configuration components and features that are supported by Minesweeper (table 2.1).

This includes common features of BGP route maps, like communities, local preference, and MEDs, as well as other configurable aspects of BGP like route redistribution. It also includes configurable OSPF attributes like link cost and areas, static routes, and ACLs. Sets of packets and route advertisements are represented by BDDs that are handled with the JavaBDD library, extending code from Bonsai [7] used to encode import filters, output filters, and ACLs.

As mentioned in the previous section, it is sometimes necessary to match up corresponding components between two routers. I used a few simple heuristics instead of manually specifying matching components. For BGP properties and route maps, I match up connections with the same neighbor id, and I report the neighbors that occur in one router but not the other. I match ACLs with the same name. For OSPF attributes, I match interfaces using a combination of their interface names, Batfish's inferred topology, and their IP address masks. This is necessary since interfaces in backup routers usually have different IP addresses. While these heuristics are not perfect, they allow Campion to be run quickly and easily.

Campion can identify differences and perform header localization for any vendor format that Batfish supports. However, currently Campion can only output exact text lines for configurations in Cisco IOS and Juniper JunOS formats, since I must write *unparsers* to convert Batfish's representation back to the original configuration text. For other formats, Campion does not produce exact text lines, but it still provides substantial localization information, including the component name, affected headers, and actions. Similarly, for some formats I do not show the exact text lines for StructuralDiff results, for example OSPF costs. But in these cases the localization information that Campion provides typically allows operators to find the relevant lines with simple text searches.

HeaderLocalize for route maps currently only provides exhaustive information for IP prefix ranges. For other relevant parts of a route advertisement such as community tags, Campion provides a single example. It is possible to extend HeaderLocalize to

provide exhaustive information across multiple parts of a route advertisement, but doing so increases the complexity both of the algorithm and of its output. The current approach has been sufficient for operators to understand Campion's results and localize the errors.

2.4 Evaluation

I applied Campion to debug router configuration differences from a large cloud provider and the campus network of a large university, both of which employ a diversity of hardware router vendors. my experiments demonstrate Campion's ability to identify cross-vendor configuration differences and to provide actionable localization information to operators.

2.4.1 Differencing in a large Data Center

Network A is from a global cloud vendor that uses routers from different manufacturers. I tested Campion on a data center network from vendor A that employs a Clos topology with hundreds of routers and thousands of servers. All routers are either Juniper or Cisco, whose configuration languages are supported by Campion. The data center network uses eBGP, iBGP, OSPF, static routes, ACLs, and route redistribution for the layer-3 routing topology. It carries business traffic for multiple global services. Each router configuration is thousands of lines.

Scenarios. I asked the network operators to employ Campion on three frequent, real and challenging tasks:

Scenario 1: Debugging redundant routers. Some routers (e.g., Top-of-Rack) are configured to be *backups* of one another with equivalent *modular* policies handling BGP, OSPF and static routes. For diversity, the operators deploy redundant routers from different vendors (e.g., Juniper, Cisco). Because network A took months to build, its current configuration comprises fragments written by different operators for diverse purposes, making hidden inconsistencies likely. It is important to not only ensure equivalence of multi-vendor, redundant routers, but also to *quickly* localize the root causes of any errors. Network A is constantly

being reconfigured as more policies are added for upcoming production traffic. Campion allows greater agility by allowing new policies to be more quickly deployed in diverse backup routers. The operators used Campion to compare all pairs of backup routers.

Scenario 2: Router replacement. Network A has an important update called *router replacement*, where operators replace a router from one vendor with one from a different vendor. Such replacements occur several times a month to take advantage of the price, performance, and newer features. For example, the operators of network A might replace lower-version Cisco routers with higher-version Juniper routers in order to avoid a Cisco bug. Router replacement is one of the riskiest update operations in network A, since operators must manually rewrite the old configurations to the new format; many critical errors have occurred as a result. The operators used Campion to check for differences between old and new configurations before performing a scheduled replacement, in order to *proactively* detect errors.

Scenario 3: Access control in gateway routers. In network A, many ACL rules are applied in gateway routers for traffic control. All of network A's gateway routers should have identical access-control policies, but it is difficult for network A's operators to guarantee this since: (1) the number of ACL rules is very large, and (2) the use of nested ACL rules makes their logic complex. The operators used Campion to check the equivalence of ACL rules in the gateway routers of the data center network.

Output evaluation. Note that network A's operators used Campion and its user interface *without any feedback or help from us in interpreting results.* The operators gave us very positive feedback on the practicality and usability of Campion. By using Campion, they found several risky, hidden configuration errors, as summarized in table 2.6. All differences that Campion found were unintentional and considered to be errors by the operators. The network configurations had recently undergone a standardization process to replace ambiguous and "uncommonly-used" configuration commands with unambiguous and standard ones. Hence any differences found by Campion were likely to be erroneous, and indeed this was

Scenario	Component	Structural or Semantic	Differences
Scenario 1	BGP	Semantic	5
	Static Routes	Structural	2
Scenario 2	BGP	Semantic	4
Scenario 3	ACLs	Semantic	3

Table 2.6: Data Center Network Results

borne out by the lack of any false positives.

Scenario 1: Debugging redundant routers. Campion detected seven configuration bugs across all of the redundant router pairs that it analyzed. Five of the bugs represent missing fragments of BGP policy, and two of them were incorrect next hops in static routes. For four BGP bugs, Campion was able to accurately localize the difference. For example, Campion pointed out that a prefix for an import filter was missing in the primary router but present in the backup one. Why were these bugs not detected by customers or real-time monitoring systems? This was because the missing prefixes had not been used for production traffic yet, but would have been in the near future. Once a service using this prefix is enabled, a service problem would have occurred. Thus, Campion proactively prevented a future service disruption.

The fifth BGP error that Campion detected used a version of the Cisco IOS format which Campion does not fully support yet. Campion still detected the error and produced useful localization information, such as the relevant input space and the actions taken by each router, but the output configuration text was inaccurate. Due to this inaccuracy, the operator reported the need to spend more time to understand the precise bug location, but they still said that it was easy to spot the deviant configuration lines from Campion's output.

The two static route errors Campion detected were misconfigured next hops. Backup routers in network A should forward the same prefix to the same next hop, but Campion detected that they were configured to forward a particular prefix p to different next hops.

This is very dangerous: a cascading failure would have triggered when the production traffic corresponding to p is turned on in the near future. Campion accurately pointed out non-equivalent next hops of this kind in *two* pairs of backup routers.

Scenario 2: Router replacement. I used Campion to test more than 30 router replacements. Campion successfully detected four bugs: one was an incorrect community number and three were incorrect local preferences. One local preference bug was for the replacement of a reflector device for iBGP. If this bug were not detected, the proposed replacement would have caused a severe outage.

Further, network A's operators also tested Campion on a synthetic case based on a static route replacement which resulted in a significant outage one year ago. The tags of two static routes were configured differently due to a misunderstanding of the semantics of the two vendors. Campion accurately pointed out the difference between the static routes. In other words, a significant outage could have been avoided if Campion had been used a year ago.

Scenario 3: Access control in gateway routers. Campion successfully detected three ACL differences between gateway routers from Cisco and Juniper. Table 2.7 shows Campion's output for one of these differences.[2] Campion's text localization identified the exact line in the Cisco ACL where traffic was rejected. The Juniper ACL equivalent is divided into terms, and Campion's text localization was able to locate which term accepted the traffic. Further, Campion's header localization also identified header information like the relevant source IP prefix.

Running Time. For each of the above three scenarios, although the configuration files of each device in network A contains thousands of lines, Campion finished its localization task within five seconds for each pair of routers.

Comparing Campion with an existing tool. While provider A has its own home-grown verification system that has been used for 1.5 years, this system can only tell whether

[2]The IP addresses and ACL name in this figure have been anonymized for confidentiality reasons.

	Router 1 (current)	Router 2 (reference)
Included Packets	srcIP: 9.140.0.3/32 dstIP: 0.0.0.0/0	
Excluded Packets	srcIP: 9.140.0.3/32 dstIP: 0.0.0.0/0 protocol: ICMP +28 more	
ACL Name	VM_FILTER_1	VM_FILTER_1
Action	REJECT	ACCEPT
Text	2299 deny ipv4 9.140.0.0 0.0.1.255 any	set firewall family inet filter VM_FILTER term permit_whitelist

Table 2.7: An example for ACL rules debugging. Router 1 and Router 2 are Cisco and Juniper routers, respectively.

the network configuration meets operator intent, but does not provide any error localization capability. Thus, network A's operators spend considerable time localizing bugs even when the existing tool identifies bugs in the network. Campion therefore provides a new capability that can potentially reduce debugging time considerably for network A's operators.

Localization efficiency. For the configurations checked, all localization results were less than five lines of configuration code. The configuration files tested vary in size from 300 lines to more than 1000 lines. Of these, the number of lines that are part of an ACL or route map definition is typically more than 100. Campion thus drastically reduces the amount of configuration that operators must search through to debug a difference.

2.4.2 Differencing in a University Network

The university network consists of approximately 1400 devices, including border routers that connect to external ISPs, backbone core routers and building routers.

I ran Campion to compare the policies for a pair of core routers and a pair of border routers. In each pair, one used Cisco configuration format and the other used Juniper format. I chose these two pairs because they are the only Cisco-Juniper backup pairs with routing

policy. The Cisco configurations and the Juniper core router configuration contain about 1800 lines of text. The Juniper border router configuration contains about 3500 lines of text. The results are shown in table 2.8.

I match route maps that are applied to the same BGP neighbor. In total, there were five pairs of operator-defined export route maps, and one pair of operator-defined import route maps. The differences that Campion found are summarized in table 2.8a.

The prefix ranges, communities, and text lines produced by Campion made it straightforward to identify these discrepancies. The list of issues that I sent to the operators does not exactly correspond to the raw output of my tool. For example, since Campion divides sets of advertisements based on which lines process them, it is possible that a single underlying difference in the configuration results in multiple lines of outputted differences. In table 2.8a, the Outputted Difference column reports the number of raw outputs produced by Campion, whereas the Differences Reported column reports how many distinct issues I reported to the operators. I categorize a reported difference as Confirmed if the operator indicated that the identified difference was both an actual difference and unintentional. The last column indicates the number of reported differences whose status is unknown at this time.

As shown in the table, the operators confirmed that most of the differences Campion identified were in fact errors. Based on earlier snapshots, the differences have been present since at least July 2017.

The route maps shown earlier in figure 2.1 illustrate two issues from a pair of core-router route maps (labeled Export 1 in table 2.8a). These were differences in the definitions of a prefix list and a community set and were confirmed as unintentional discrepancies. For the difference in the prefix lists, the operator agreed it was a misconfiguration, but was not sure whether the Cisco or Juniper router was correct. For the community difference, the operator wrote: "The community group is an obvious mistake on my part. The Juniper config is wrong. I followed the wrong Juniper doc when configuring the community group."

Router Pair	Route Map	Outputted Differences	Differences Reported	Confirmed	Pending
Core Routers	Export 1	5	5	4	1
	Export 2	1	1	1	0
Border Routers	Export 3	1	1	1	0
	Export 4	1	1	1	0
	Export 5	2	1	1	0
	Import	0	-	-	-

(a) SemanticDiff results on route maps

Router Pair	Component	Classes of Errors	Differences Reported	Confirmed	Pending
Core Routers	Static Routes	2	1	0	0
	BGP Properties	1	1	0	0

(b) StructuralDiff results

Table 2.8: University Network Results

In addition to the differences shown in figure 2.1, the actual route maps contained different definitions for their third clause, with the Juniper router performing a match on communities that was not done in the Cisco router. They also have different redistribution behavior for certain addresses. Further, the two routers have different fall-through behaviors (accept vs. deny) when handling advertisements that fail to match any clause, which causes two additional behavioral differences. Operators confirmed all but the last of these issues, which is still pending. When asked about the difference between the third clauses of each route map, the operator replied: "The Juniper config is correct and the intent is obvious because of the English-language syntax. The Cisco config we're not sure what change should be made, if any." This demonstrates the challenge for operators when dealing with multi-vendor backups, and the need for a tool like Campion to ensure consistency and localize errors.

Export 2, the other core router policy, also had the difference in prefix lists mentioned previously for Export 1 but did not have any other issues. The differences in the border router policies similarly affected the matched prefixes and communities but were of a different

nature: there were differences in two regular expressions used to match communities for Export 3 and Export 4. Campion reported that advertisements with a certain community were accepted in the Cisco router but not the Juniper router. For Export 5, there was one prefix that was absent in a prefix list in the Juniper router but present in the Cisco router list. These were also confirmed as errors by the operators.

When comparing other properties of the core routers using Campion's StructuralDiff, I found differences in the static route configuration and the BGP configuration. In the static routes I found two classes of differences. The first included many static routes that applied to the same prefix but had different next hops and different administrative distances. I deemed these as intentional differences, since the next hops had similar addresses, suggesting that their next hop routers were of the same role, and the administrative distances did not affect the relative priority of routes. The second class of static route differences included two static routes that were present in one router but not the other, as demonstrated in section 2.1. These were reported to the operators, and they said that these were intentionally added as a workaround for a specific BGP routing issue. The BGP configuration difference was that certain iBGP neighbors of the Cisco router were missing a `neighbor send-community` command to propagates communities, while Juniper routers send communities by default. The operators indicated that this configuration difference does not cause a behavioral difference because the core routers do not set communities on routes.

2.4.3 False Positives

I distinguish between two types of false positives that Campion may produce, both of which were exhibited in the results for the university network. First, there can be intentional differences between routers. This was the situation for the static routes that were added in one configuration as a workaround for a specific BGP routing issue, as well as for the static routes that had differing next hops. Second, there can be spurious differences due to Campion's modular approach. Specifically, any *potential* behavioral difference between

corresponding components is reported by Campion, but these differences may not cause an *actual* behavioral difference in the current network, for example because the differences are shadowed or accounted for by other parts of the configuration. This was the situation for the iBGP neighbors of one router which were not configured to send communities.

However, I argue that it is still worthwhile to report both kinds of false positives. Reporting intentional differences allows the operator to ensure that all and only expected differences exist between the two routers. In the case of static routes added as a workaround, the operator commented, "I just need to find another way to resolve this," indicating that this difference is intentional but still not optimal. Reporting spurious differences is valuable because they represent *latent* errors that can potentially be "activated" by a change elsewhere in the network configuration. In the case of the spurious difference for sending communities, if the core routers later start to set communities on routes then this difference will cause an important behavioral difference. Indeed, the operator commented that these kinds of spurious differences would likely be examined and addressed when the routers are next replaced.

2.4.4 Scalability

For each of the data center scenarios, Campion finished its localization task within five seconds for each pair of routers. For the university core and border pairs, the total runtime to compare the core and border pairs was 3 seconds. When combined with the parsing of the configurations, the total time was under 10 seconds, with configuration parsing taking a majority of the time. I additionally tested the scalability of SemanticDiff for ACLs. I used Capirca[3] to randomly generate nearly equivalent ACLs for Cisco and Juniper configurations. I introduced 10 differences between the two ACLs and compared them. When the ACLs were generated with 1000 rules, SemanticDiff took less than a second. When the ACLs were generated with 10,000 rules, SemanticDiff took 15 seconds. These tests were done

[3] https://github.com/google/capirca

with a 2.2 GHz CPU. Moreover, Batfish's parsing time for the 10,000 case is 13 seconds, which is comparable to the runtime of SemanticDiff.

CHAPTER 3

Modular Verification of BGP Safety Properties

A key problem in verifying network control plane properties with formal methods is scaling these techniques to large networks. While previous tools attempt to scale through various means, they are not efficient enough to be used today on large real-world networks such as the wide-area networks of hyperscalers.

This lack of scalability is fundamental to the *monolithic* approach to modeling that is used by earlier tools. They analyze the network configuration and routing processes as a whole, exhaustively exploring all possible control-plane behaviors induced by the complex interactions among all configuration directives and protocols. Networks can contain hundreds of routers and links, and as the size of the network grows, the number of possible network states grows exponentially, making it very difficult to scale verification techniques. In other domains, like software or hardware, the way to handle this is through *modular* checking. In this style, subsystems (e.g., a software function or hardware module) are verified independently to meet *local* specifications (e.g., a precondition/postcondition pair) that together imply a desired global property. Prior work has used modularity to scale data-plane analysis [23], but modularizing control-plane verification is more challenging due to complex routing protocols and policies.

Control plane behavior depends on the interaction of complex configurations with BGP, a distributed message-passing protocol. A classical way to reason modularly about protocols is through invariants indexed by time [50], and/or employ temporal logic [33]. This requires significant effort and expertise. Instead, I demonstrate that in practice a wide range of

Tool Feature	Minesweeper [6]	BagPipe [53]	Plankton [42] Tiramisu [3]	ARC [20] Hoyan [55]	Lightyear
Analyzes all peer BGP routes	●	●	○	○	●
Analyzes failures	●	○	●	●	●
Checks safety and liveness properties	●	◐	●	●	●
Verification is fully automatic	●	●	●	●	◐
Near linear scaling with network size	○	○	○	○	●
Localizes bugs in configurations	○	○	○	○	●

Table 3.1: Comparison of prior verification tools with Lightyear.

desired properties can be modularly verified without making time explicit.

This chapter, along with chapter 4, presents a modular approach to network control plane verification that is implemented in the tool Lightyear. Like prior verifiers, it takes as input a network's configuration and a *global* property to verify. However it additionally requires the user to provide local constraints that should hold on individual routers and edges. Given a set of *local* constraints on individual nodes and edges, Lightyear can produce a set of checks that are applied to route maps in the router configurations. If all the checks for the policies pass, then that implies that the global property holds.

A comparison between Lightyear and previous tools is shown in table 3.1. Since Lightyear can prove global properties using only local checks, and there is a limit to the size of local checks, Lightyear can scale nearly linearly in the size of the network. It is the first tool that can scale in such a fashion. Additionally, since checks are local to specific route maps, if any check fails, the user can immediately narrow down errors to the relevant router configuration.

This chapter focuses on *safety* properties on individual routers, which intuitively ensure that "bad" routes never reach a particular node. This includes common properties like filtering bogons, preventing transit between peers, and ensuring isolation. Chapter 4 focuses on *liveness* properties, which intuitively ensure that a "good" route will eventually be accepted or forwarded at a particular location. This includes many control-plane reachability queries: for example that a route received from one neighbor will be sent to another. In the following sections, I first show an overview of checking a safety property with an example, and then I

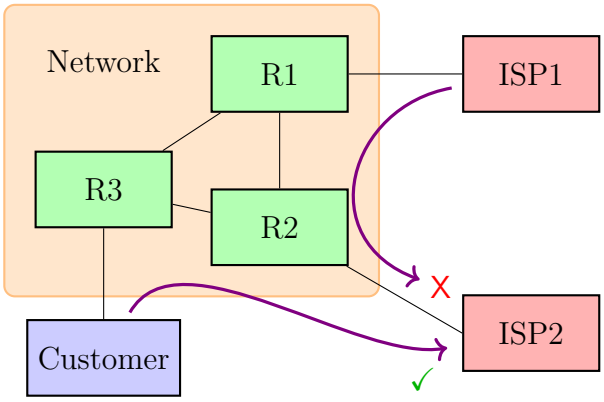


Figure 3.1: Example network with safety and liveness properties. Routes from ISP1 should not be sent to ISP2 (safety). Routes from Customer should reach ISP2 (liveness). Policies are implemented by tagging and checking communities.

present the general formulation and proof.

3.1 Overview and Example

In this section I show how Lightyear checks safety properties with the example in Figure 3.1. In the example network, each edge represents a connection between BGP speakers. The network contains three BGP routers: R1, R2, and R3. R1 and R2 each have an ISP as an external neighbor. R3 is connected to an external neighbor that is a customer. The network satisfies two properties. First, it satisfies the standard no-transit property that routes originating from ISP1 should not be advertised to ISP2, and second, it satisfies the property that routes from Customer, with appropriate prefixes, should eventually be sent to ISP2. The former is a *safety property*, holding when a certain event (advertising a route from ISP1 to ISP2) never occurs. The latter is a *liveness property*, which is explained in chapter 4. Both are network-wide policies in that they depend on the interaction of multiple routers to achieve the correct result.

Existing control-plane verifiers [6, 53, 20, 3, 42] would verify these properties by creating a representation of the possible data planes that can result from the *entire network's configuration* and then searching this representation for counterexamples. This joint representation

of all network node behaviors has inherent scalability limitations.

However, the scalability limitations can be overcome by using knowledge about the structure of the network configurations. Network configurations are highly modular by design. Each router has route maps, which define an import and export policy on each peering session. These determine the router's responsibilities, and the different local policies act in combination to ensure desired global properties. In our example network, one way to ensure the no-transit property is to use a common approach based on communities:

- R1's import policy marks received routes from ISP1 with a BGP community (a simple 32-bit tag) with value `100:1`
- R2's export policy filters routes tagged with `100:1` when advertising to ISP2, and
- no other import or export policy strips community `100:1` from routes that it advertises.

Note that each of the above behaviors is *node-local* and pertains to an individual BGP route map. Unlike Lightyear, prior control plane verification tools are not aware of this modular structure and so cannot leverage it. Alternatively, one could make a tool that can simply perform these kinds of local checks. However, in that case there is no guarantee that together they imply the desired end-to-end property. Even in this simple example, the fact that it is necessary to check the third condition above is subtle and easily missed. The rest of this section shows the necessary inputs and generated checks need to modularly verify this property for the network in Figure 3.1. The inputs consist of an end-to-end property and a network invariant. These are used to generate a set of local checks. If the checks hold, then the network property is verified.

End-to-end Property: For safety properties, the end-to-end property of interest is specified as a pair of a particular location in the network and a predicate on the routes reaching that location. Many network policies fall into this class of properties, for example bogon filtering; ensuring that a network only advertises routes to its own destinations; and

Type	Location(s)	Logical Formula	Description
End-to-end Property	R2 → ISP2	$\neg\texttt{FromISP1}(r)$	*No routes sent to ISP2 come from ISP1*
Network Invariants	ISP1 → R1	`True`	*ISP1 can send our network any route*
	R2 → ISP2	$\neg\texttt{FromISP1}(r)$	*No routes sent to ISP2 come from ISP1*
	Nodes and other edges in network	$\texttt{FromISP1}(r) \Rightarrow \texttt{100:1} \in \text{Comm}(r)$	*Routes from ISP1 are tagged with community 100:1*
Generated Checks	ISP1 → R1	$(\texttt{True} \wedge r' = \texttt{Import}(\text{ISP1} \to \text{R1}, r)) \Rightarrow (\texttt{FromISP1}(r') \Rightarrow 100{:}1 \in \text{Comm}(r'))$	
	R2 → ISP2	$((\texttt{FromISP1}(r) \Rightarrow 100{:}1 \in \text{Comm}(r)) \wedge r' = \texttt{Export}(\text{R2} \to \text{ISP2}, r)) \Rightarrow \neg\texttt{FromISP1}(r')$	
	Other Edge E	$((\texttt{FromISP1}(r) \Rightarrow 100{:}1 \in \text{Comm}(r)) \wedge r' = \texttt{Export}(E, r)) \Rightarrow (\texttt{FromISP1}(r) \Rightarrow 100{:}1 \in \text{Comm}(r))$ $((\texttt{FromISP1}(r) \Rightarrow 100{:}1 \in \text{Comm}(r)) \wedge r' = \texttt{Import}(E, r)) \Rightarrow (\texttt{FromISP1}(r) \Rightarrow 100{:}1 \in \text{Comm}(r))$	

Table 3.2: Using Lightyear to prove the no-transit property from figure 3.1. The user-provided global property and local invariants are show in blue. Lightyear-generated local verification checks are shown in yellow.

forms of isolation between nodes or groups of nodes. Such properties can also express complex constraints among BGP attributes, for example that prefixes in a specific range always have a particular local preference or MED value.

As shown in the first line of Table 3.2, the no-transit property specifies that no route transmitted over the edge from R2 to ISP2 should originate at ISP1. To enable the expression of rich properties, Lightyear allows users to define *ghost attributes* that conceptually update message headers with additional fields. This is a common technique in software verification, where additional variables are introduced that do not affect the computation but allow for easier property specification [18]. In the table, $\texttt{FromISP1}(r)$ is a boolean ghost variable that is defined by the user to be false in all originated routes, set to true by the import filter on R1 from ISP1, and left unchanged by all other filters.

Network Invariants: Users must also specify invariants. While in principle the user could specify a different invariant for each network location, many locations play the same role in the network and have the same behavior with respect to the desired end-to-end property. In our example, there are only three network invariants, shown in Table 3.2, which correspond exactly to the three node-local behaviors described earlier that ensure the no-

transit property. First, no assumption is made about the routes coming from ISP1 to R1, so the associated predicate is True. Second, routes coming from R2 to ISP2 should not come from ISP1. Note that this invariant is identical to the end-to-end property, which is common but need not be the case. Third, all other locations in the network should satisfy the key correctness invariant: routes from ISP1 must be tagged with the community `100:1`. Notably, this three-part decomposition is analogous to the modular verification of software [21], which typically involves a *precondition* that is assumed to hold initially, a *postcondition* to be proven, and one or more *inductive invariants* that hold throughout each execution and are sufficient to imply the postcondition.

Generated Checks: Given this information from the user, Lightyear automatically generates local checks to validate the given network invariants. Importantly, each local check pertains to a single BGP filter on a single network router, applied to messages from a specific neighbor. Together these checks implement a form of *assume-guarantee reasoning* [26, 41]: each location's network invariant is proven under the assumption that the local invariant of its directly connected locations hold. As I show later, together these checks imply that all local invariants in the network are respected.

Table 3.2 shows the local checks that Lightyear automatically generates for our running example. The first check ensures that the import filter at R1 on the edge from ISP1 to R1 establishes the key invariant $\texttt{FromISP1}(r) \Rightarrow 100{:}1 \in \text{Comm}(r)$. Since that filter tags all routes with community `100:1`, the check is easily provable by an SMT solver. The second check ensures that the key invariant is sufficient to ensure that routes from ISP1 are not exported on the edge from R2 to ISP2. Since the export filter at R2 on that edge drops all routes that are tagged with `100:1`, the check passes. The third set of checks ensure that the key invariant is preserved by all other import and export filters in the network. Since these filters never strip community `100:1` from a route, the checks pass.[1] Lastly (not shown in the table), Lightyear must check that the invariant on the edge from R2 to ISP2 implies the

[1]There are also some analogous checks for originated routes, but they are omitted here for simplicity.

end-to-end property. This check is trivial since the two properties are identical.

In summary, Lightyear's approach to control-plane verification leverages the modular structure that is already present in the network configurations. By requiring the user to make this structure explicit through a set of local invariants at each location, Lightyear soundly reduces checking an end-to-end network property to a set of checks that each pertain to a single BGP import or export filter.

This approach has numerous benefits over the prior, monolithic approaches. First, our approach is highly scalable, since the number of checks is linear in the number of edges in the BGP network graph. Second, Lightyear's modular checks provide a very strong guarantee. For both safety and liveness properties, the approach handles all possible external route announcements from neighbors. For safety properties, it additionally provides resilience to arbitrary failures "for free," since it proves that "bad" routes are not received without making any assumptions about the paths on which they might flow. Third, the modular approach naturally supports incremental verification when a node is updated: only the local checks pertaining to that node must be re-checked. Finally, modularity has large benefits for error localization and understanding: the failure of a local check directly pinpoints the erroneous import or export filter and the local invariant that it fails to satisfy.

3.2 BGP Model

This section describes our model of BGP. I model the semantics of BGP as a set of allowed *traces*. Our semantics is a variant of that from the Bagpipe tool [53].

A trace is a sequence of *events*. There are three types of events: `recv`, `slct`, and `frwd`. For $r \in$ Routes, R and $N \in$ Routers, and $R \to N \in$ Edges:

1. `recv`$(N \to R, r)$ occurs when R receives route r from neighbor N
2. `slct`(R, r) occurs when R selects r as the best route for a destination and installs it
3. `frwd`$(R \to N, r)$ occurs when R forwards route r to the neighbor N

Using the terms from RFC4271 [45], one can also think of $\texttt{recv}(N \to R, r)$, $\texttt{slct}(R, r)$, and $\texttt{frwd}(R \to N, r)$ as corresponding to installing a route into the Adj-RIB-In, Loc-RIB, and Adj-RIB-Out, respectively. Thus, traces can be thought of as the sequence of updates to network's BGP state. I denote the set of all traces as Traces.

A *valid* trace is one that could occur for a given topology and policies, according to the BGP semantics. I formalize the notion of trace validity as a set Valid $\subseteq$ Traces of traces that satisfy specific properties. I consider a trace $A_1, A_2, \ldots, A_n$ to be valid, and hence part of the set Valid, if it satisfies a set of *safety axioms*.

The safety axioms consist of the following properties, for all $1 \leq k \leq n$:

1. If $A_k = \texttt{recv}(N \to R, r)$, then either:
   (a) $N \in$ Externals, or
   (b) there exists $j < k$ such that $A_j = \texttt{frwd}(N \to R, r)$
2. If $A_k = \texttt{slct}(R, r)$, then there exists $j < k$, $r' \in$ Routes, and $N \in$ Routers $\cup$ Externals such that $A_j = \texttt{recv}(N \to R, r')$ and $r = \texttt{Import}(N \to R, r')$
3. If $A_k = \texttt{frwd}(R \to N, r)$, then either:
   (a) $r \in \texttt{Originate}(R \to N)$, or
   (b) there exists $j < k$ and $r' \in$ Routes such that $A_j = \texttt{slct}(R, r')$ and $r = \texttt{Export}(R \to N, r')$

3.3 Safety Checks

Lightyear requires three inputs from the user in order to check safety properties. The first input, the network configurations, is standard. As described previously, the configurations are used to build the BGP topology as well as the policy functions.

The second input is the network safety property, which requires that all route announcements that can reach a particular location satisfy certain constraints. Formally, a network safety property is a pair (ℓ, P) where:

$$(\ell, P) \in (\textsc{Routers} \cup \textsc{Edges}) \times \mathbb{P}(\textsc{Routes})$$

ℓ is a location, either a router or an edge, and P is a set of routes matching a particular constraint. In practice, users directly specify a logical constraint on route attributes that represents P.

Each safety property (ℓ, P) corresponds to a property of all possible valid traces, as defined in the previous section — all routes that can reach location ℓ must satisfy P. Formally, a network satisfies a property (ℓ, P) if for all $T \in \textsc{Valid}, r \in \textsc{Routes}, R, N \in \textsc{Routers}$

- if $\ell = R$ and $\texttt{slct}(R, r) \in T$, then $r \in P$
- if $\ell = R \to N$ and $\texttt{frwd}(R \to N, r) \in T \vee \texttt{recv}(R \to N, r) \in T$, then $r \in P$

For example, the location $(R1 \to R2)$ and constraint 1:1 $\in \text{Comm}(r)$ together specify the property that if $\texttt{frwd}(R1 \to R2, r)$ or $\texttt{recv}(R1 \to R2, r)$ are in a valid trace, then r should always have the community 1:1.

Finally, Lightyear's third input is a set of network invariants, one per location in the given network. Formally, the network invariants are modeled as a set of pairs denoted I:

$$I \subseteq (\textsc{Routers} \cup \textsc{Edges}) \times \mathbb{P}(\textsc{Routes})$$

Each element of the set has the form (ℓ, P), where ℓ is a location and P is a set of routes, as in the network property defined above. The semantics of each pair is a property of traces, analogous to the semantics of network properties shown above.

I require that there exist exactly one pair in I per location in the given network, and I use

the notation I_ℓ to denote the set P of routes associated with location ℓ in I. I also require that $I_{R \to N} = \textsc{Routes}$ for each edge $R \to N$ where $R \in \textsc{Externals}$. In other words, I make no assumption about routes coming from external neighbors but rather assume that any route may be advertised.

Given the network configuration, network property (ℓ, P), and network invariants I, Lightyear generates the following local checks for each edge $A \to B$ in the network topology, which validate each location's network invariant using assume-guarantee reasoning:

1. **Import:** For all $r, r' \in \textsc{Routes}$, if $r = \texttt{Import}(A \to B, r')$ and $r' \in I_{A \to B}$, then $r \in I_B$.

2. **Export:** For all $r, r' \in \textsc{Routes}$, if $r = \texttt{Export}(A \to B, r')$ and $r' \in I_A$, then $r \in I_{A \to B}$.

3. **Originate:** For all $r \in \textsc{Routes}$, if $r \in \texttt{Originate}(A \to B)$, then $r \in I_{A \to B}$.

For example, the first check verifies that the import route map at B on the edge $A \to B$ satisfies I_B, assuming that $A \to B$ satisfies its local invariant. If the router B is external then the import check is not performed, and similarly if the router A is external then the export and originate checks are not performed. In our implementation of Lightyear, the local checks are performed by modeling import and export filters using SMT constraints and invoking an SMT solver to validate each check or provide a counterexample.

Finally, Lightyear checks that the network invariants I imply the network property (ℓ, P). This is done simply by requiring that $I_\ell \subseteq P$, i.e. that the network invariant for ℓ implies the network property P. Again this check is performed with an SMT solver.

3.4 Proof of Correctness

In this section I prove that Lightyear's modular approach to control-plane verification is correct.

First I state and prove the key lemma, which says that the local checks are sufficient to ensure that the network invariants I hold, for all valid traces.

Lemma: Given a BGP topology and policy as well as network invariants I, let C be the set of Import, Export, and Originate checks that Lightyear generates. If all checks in C pass, then for all $T \in \textsc{Valid}, r \in \textsc{Routes}, R, N \in \textsc{Routers}$:

- if $\texttt{slct}(R, r) \in T$, then $r \in I_R$
- if $\texttt{frwd}(R \to N, r) \in T \vee \texttt{recv}(R \to N, r) \in T$, then $r \in I_{R \to N}$

Proof: The proof is by induction on the length of the (partial) trace T.

Base case: For a partial trace of length 0, there are no events, so the statement is vacuously true.

Inductive case: Suppose $T = A_1, A_2, \ldots, A_{k+1}$. I assume by induction that the statement is true for $A_1, A_2, \ldots, A_k$. I do a case analysis on the event A_{k+1}:

Case $A_{k+1} = \texttt{recv}(N \to R, r)$, so I have to show that $r \in I_{N \to R}$. By the trace validity axioms, either:

1. $N \in \textsc{Externals}$. In this case I know that $I_{N \to R} = \textsc{Routes}$, so $r \in I_{N \to R}$.
2. There exists $j < k + 1$ such that $A_j = \texttt{frwd}(N \to R, r)$. Then by the inductive hypothesis I have that $r \in I_{N \to R}$.

Case $A_{k+1} = \texttt{slct}(R, r)$, so I have to show that $r \in I_R$. From the trace validity axioms, I know that there exists $j < k+1, r' \in \textsc{Routes}$, and $N \in \textsc{Routers} \cup \textsc{Externals}$ such that $A_j = \texttt{recv}(N \to R, r')$ and $r = \texttt{Import}(N \to R, r')$. From the inductive hypothesis, I know that $r' \in I_{N \to R}$. Therefore by the Import check in C for $N \to R$, I can conclude that $r \in I_R$.

Case $A_{k+1} = \texttt{frwd}(R \to N, r)$, so I have to show that $r \in I_R$. By the trace validity axioms, either:

1. $r \in \texttt{Originate}(R \to N)$. Then from the Originate check in C for $R \to N$ I have that $r \in I_{R \to N}$.

2. There exists $j < k+1$ and $r' \in$ Routes such that $A_j = \texttt{slct}(R, r')$ and $r = \texttt{Export}(R \to N, r')$. From the inductive hypothesis, I have that $r' \in I_R$. Then from the Export check in C for $R \to N$, I can conclude that $r \in I_{R \to N}$.

Now I prove the correctness theorem for Lightyear, which says that Lightyear's checks are sufficient to ensure that the given network property holds, for all valid traces.

Theorem: Given a BGP topology and policy, a network property (ℓ, P), and network invariants I, let C be the set of Import, Export, and Originate checks that Lightyear generates. If all checks in C pass and $I_\ell \subseteq P$, then for all $T \in$ Valid, $r \in$ Routes, $R, N \in$ Routers:

- if $\ell = R$ and $\texttt{slct}(R, r) \in T$, then $r \in P$
- if $\ell = R \to N$ and $\texttt{frwd}(R \to N, r) \in T \vee \texttt{recv}(R \to N, r) \in T$, then $r \in P$

Proof: There are two cases:

1. $\ell = R$ and $\texttt{slct}(R, r) \in T$. From the earlier lemma I have that $r \in I_\ell$, and since $I_\ell \subseteq P$ it follows that $r \in P$.

2. $\ell = R \to N$ and $\texttt{frwd}(R \to N, r) \in T \vee \texttt{recv}(R \to N, r) \in T$. Again from the earlier lemma I have that $r \in I_\ell$, and since $I_\ell \subseteq P$ it follows that $r \in P$.

3.5 Extensions and Discussion

3.5.1 Ghost Attributes

To increase Lightyear's expressiveness, users can define *ghost attributes*, which conceptually extend each route with additional fields. For example, the $\texttt{FromISP1}(r)$ ghost attribute from Section 3.1 is used to indicate whether r originated from ISP1. A ghost attribute is defined

by specifying the set of values that the attribute can take, along with updates to the `Import`, `Export`, and `Originate` functions that make up the given network's policy (Section 3.2).

In the case of `FromISP1`(r) from Figure 3.1, it can be defined as a boolean attribute with the following behavior:

- the import filter on ISP1 $\rightarrow$ R1 sets `FromISP1` to true
- the import filter on ISP2 $\rightarrow$ R2 sets `FromISP1` to false
- other filters leave `FromISP1` unchanged
- all originated routes have `FromISP1` set to false

Other natural network properties can be expressed using ghost attributes. A `WaypointR` attribute that is true only for routes processed by a particular router R can be defined by specifying that filters on R set `WaypointR` to true, origination as well as import filters from external neighbors at other routers set `WaypointR` to false, and filters between the routers in the network leave `WaypointR` unchanged.

Ghost attributes do not affect the description of Lightyear or proof of its correctness above, as they do not depend on the specific set of attributes that are in a route.

3.5.2 Fault Tolerance

A significant benefit of Lightyear's approach to control-plane verification of safety properties is that it supports reasoning about failures "for free." That is, if all of Lightyear's checks pass, then the given network property is guaranteed to hold not only in the failure-free case but also in the presence of *arbitrary* node and link failures.

Lightyear soundly reasons about failures because of our over-approximate notion of trace validity (Section 3.2. Specifically, any trace that is feasible according to the given BGP topology and passes the import and export filters along the corresponding path is considered

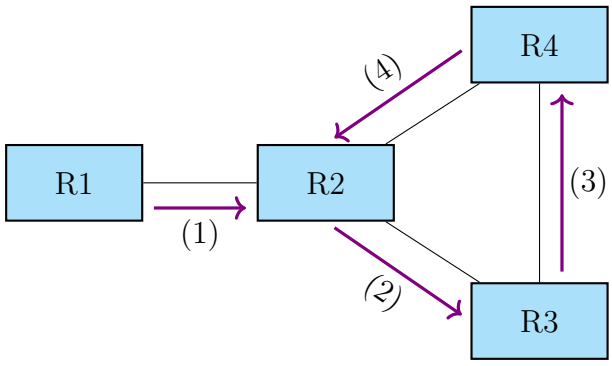


Figure 3.2: A network where a safety property cannot be prove

valid. Hence, every trace that can occur under any failure scenario is already considered valid. By our correctness theorem, all of these traces satisfy the property (ℓ, P).

3.5.3 Incompleteness

The method for verifying safety properties is always sound but it may not be complete for pathological networks. Incompleteness in this case means that there is a combination of a network and a property such that the property holds on all valid traces in the network, but it is not possible to prove the property by stating an invariant and checking the local contracts generated by the invariant. Specifically, my approach may not work for networks where the import and export policies do not prohibit loops, and the correctness of the property depends on the BGP best route selection mechanism.

Consider the example in figure 3.2. In this network, there are four routers and the numbered arrows show the path taken by a route propagated through the network. Suppose it has the following policies configured:

- At R2, the import policy on routes from R1 adds community sets local preference 200.
- At R2, the import policy on routes from R4 adds community `10:1` and sets local preference 50.
- All other policies accept all routes sent between R1, R2, R3, R4 without altering them

(even those that would normally be dropped because of AS paths or iBGP fields).

- Routes from any neighbors not pictured are rejected
- Only R1 originates routes.

Now, suppose the user wants to prove the property that for all routes r and traces T, $\texttt{slct}(R2, r) \in T \implies \texttt{10:1} \notin \text{Comm}(r)$. This can be expressed as a safety property. The property is true for the following reason: the only possible way for R2 to receive and import a route with community `10:1` is if the route from R1 looped around the network and came back to R1 via R4, as in the path shown in figure 3.2. The route will have local preference 50, but in order for this to occur, R2 must already have a route obtained from R1 with local preference 200. Therefore R2 will never select and install a route with community `10:1`.

However, even though the property is true, it is not possible to state an invariant for this that is established with checks on import and export filters. If there were an invariant I, it must be that $I_{R2} \implies \texttt{10:1} \notin \text{Comm}(r)$. It is not possible to establish this invariant because it is possible for R2 to successfully import a route r with $\texttt{10:1} \notin \text{Comm}(r)$, namely the case where the route comes from along the path in the figure above. Because of this, my safety checking framework is not complete in all cases.

The fundamental reason why the above example cannot be checked is that it contains a looping path and depends on the BGP best route selection mechanism to ensure safety. The looping path ensures that the "bad" route is present only when a "good" route is already present in a particular routing table, and relying on the best route selection mechanism means that the property is not checkable with checks on import and export filters. It is worth noting that in a typical BGP network, the AS path attribute and iBGP route reflector attributes prevent signaling loops from occurring in the network, so cases like the above will never occur. My conjecture is that for networks where import and export policies (which includes the default AS path and iBGP rules) prevent loops, all properties should be checkable. That is, for all true properties, it is possible to state an invariant that implies the property with

generated checks that will pass.

CHAPTER 4

Modular Verification of BGP Liveness Properties

The previous chapter shows how safety properties can be checked using modular checks. This chapter extends that to another class of network properties. These are liveness properties in that intuitively they model that some "good" route will eventually be accepted or forwarded in the network given some initial conditions. This includes many control-plane reachability queries: for example that a route received from one neighbor will be sent to another.

Reasoning modularly about liveness properties is particularly challenging; it requires that the modular checks together imply an end-to-end path through the network. I show a natural approach to do this using two kinds of constraints: *path constraints* that ensure the feasibility of a "good" path, and a related *no-interference invariant* that ensure good paths cannot be prevented. In the following sections, I define the class of liveness properties, show the checks necessary to prove such a property.

4.1 Overview and Example

Consider the example in figure 3.1. This network should satisfy the property that routes from Customer, with appropriate prefixes, will eventually be sent to ISP2. This is a liveness property because I want to show that a "good" route (a route from Customer), will eventually be sent to ISP2, given the initial condition that the network receives such route from Customer.

The rest of this section shows the inputs that the user would need to provide and the checks that would need to be performed in order to verify the property in this example. A

more general formulation is shown in the following sections. The user needs to provide the end-to-end property, a path that a route can travel along, and a set of constraints for the route at each step in the path. This will be used to generate two things: a set of propagation checks, and a no-interference property.

End-to-end Property: For liveness properties, the end-to-end property of interest is also a pair of a particular location in the network and predicate. However, here the predicate indicates that a route satisfying the property will eventually reach that location. The property in Table 4.1 shows that a route with a customer prefix will eventually be sent from R2 to ISP2. If the routes of interest come from a neighbor, as in this case, then the property will only be provable under the assumption that the neighbor advertises such a route. Users can optionally specify such an assumption, as shown in the table.

Path and Constraints: As with safety properties, users need to provide a set of local constraints on individual network locations, but they take a different form for liveness properties. Users must provide a *path* through the network that the desired route can take to reach the destination from the source, along with local constraints for each edge and node along the path. The path does not need to be unique. Intuitively, each local constraint indicates the properties of the "good" routes that will reach that particular location, and together they constitute a witness that a "good" route will eventually reach its intended destination. As shown in Table 4.1, our example has two path constraints: at locations R3, R2, and R3 $\rightarrow$ $R2$ there will eventually be a route with the customer prefix that does not have the community `100:1`, and at R2 $\rightarrow$ ISP2 there will eventually be route with the customer prefix. It is important that routes from Customer do not have the community `100:1`, or else they will be dropped at R2, as I saw for the no-transit property.

Propagation Checks: In order to prove the liveness property two types of checks need to be performed. First, there are local checks that together imply that a route will in fact traverse the given path, in the absence of interference from other possible paths. These checks are analogous to the generated checks for safety properties shown earlier. In our

example in Table 4.1, the checks again follow a three-part structure, with $Customer \rightarrow R3$ establishing an invariant, $R3 \rightarrow R2$ maintaining the invariant, and $R3 \rightarrow R2$ using the invariant.

No-interference: Finally, liveness properties require an additional set of checks. Since BGP only selects the best route available from all of a router's neighbors, it is not enough to show that filters do not reject "good" routes along our path. It is also necessary to show that other routes in the network can never interfere, at any node along the path. To do this, I need to check that any route accepted on the path with the same prefix must satisfy the corresponding path constraint. For this example, at R3 and R2, routes with a customer prefix are checked to never have the community `100:1`. This constraint ensures that if routes for customer prefixes arrive along other paths and are preferred to those arriving on our path, those routes will still satisfy the desired property (i.e., they will be sent from R2 to ISP2). Note that this means that our approach does not require that the specified path be unique in the network, so I can verify liveness properties even in some scenarios where there is routing redundancy. The no-interference constraint is itself a set of safety properties, and so in general it must be proven using the machinery shown in the previous subsection, with its own set of local invariants and checks.

4.2 Checks

I now describe the general framework for verifying liveness properties modularly. Proving liveness properties modularly is more difficult than proving safety properties, since it requires showing both that "good" routes are allowed and that interfering routes are not.

4.2.1 Inputs for Liveness Checks

The inputs for a liveness check consist of the following:

1. The network configurations
2. A liveness property $(\ell, P) \in \{\textsc{Routers} \cup \textsc{Edges}\} \times \mathbb{P}(\textsc{Routes})$

Type	Location(s)	Logical Formula	Description
End-to-end Property	R2 → ISP2	$\texttt{HasCustPrefix}(r)$	*Customer prefixes are advertised to ISP2*
Assumption	Customer → R3	$\texttt{HasCustPrefix}(r)$	*Assume customer routes are advertised to R3*
Path Constraints	R3, R2, R3 → R2	$\texttt{HasCustPrefix}(r) \wedge \neg$ 100:1 $\in \text{Comm}(r)$	*Routes from customer are accepted/forwarded and not tagged with community 100:1*
	R2 → ISP2	$\texttt{HasCustPrefix}(r)$	*Routes are forwarded to ISP2*
Propagation Checks	Customer → R3	$(\texttt{HasCustPrefix}(r) \wedge r' = \texttt{Import}(\text{Customer} \to \text{R3}, r)) \Rightarrow (\texttt{HasCustPrefix}(r') \wedge \neg 100{:}1 \in \text{Comm}(r'))$	
	R3 → R2	$((\texttt{HasCustPrefix}(r) \wedge \neg 100{:}1 \in \text{Comm}(r)) \wedge r' = \texttt{Export}(\text{R3} \to \text{R2}, r)) \Rightarrow (\texttt{HasCustPrefix}(r') \wedge \neg 100{:}1 \in \text{Comm}(r'))$	
		$((\texttt{HasCustPrefix}(r) \wedge \neg 100{:}1 \in \text{Comm}(r)) \wedge r' = \texttt{Import}(\text{R3} \to \text{R2}, r)) \Rightarrow (\texttt{HasCustPrefix}(r') \wedge \neg 100{:}1 \in \text{Comm}(r'))$	
	R2 → ISP2	$((\texttt{HasCustPrefix}(r) \wedge \neg 100{:}1 \in \text{Comm}(r)) \wedge r' = \texttt{Export}(\text{R2} \to \text{ISP2}, r)) \Rightarrow \texttt{HasCustPrefix}(r')$	
No-interference Checks (Safety Properties)	R3, R2	$\texttt{HasCustPrefix}(r) \Rightarrow \neg 100{:}1 \in \text{Comm}(r)$	*Routes accepted at R3 and R2 with a customer prefix must not have community 100:1*

Table 4.1: Using Lightyear to prove the liveness property from figure 3.1. The user-provided global property, and path constraints are show in blue. The propagation checks are shown in yellow for the path is Customer → R3 → R2 → ISP2. The no-interference checks are safety properties proven using their own invariants (not shown).

3. A path $(\ell_1, \ldots, \ell_n = \ell)$ where $\ell_i \in \{\textsc{Routers} \cup \textsc{Edges}\}$

4. A constraint $C_1 \ldots C_n$ for each location in the path, where $C_i \in \mathbb{P}(\textsc{Routes})$

The property (ℓ, P) represents a liveness property of all valid traces, namely that there will eventually be a route at ℓ that satisfies P. Formally, this means for all $T \in \textsc{Valid}$, either:

- $\ell \in \textsc{Routers}$ and there exists r' such that $\texttt{slct}(\ell, r') \in T$ and $P(r')$ holds, or
- $\ell \in \textsc{Edges}$ and there exists r' such that $\texttt{frwd}(\ell, r') \in T$ and $P(r')$ holds

The path $(\ell_1, \ell_2, \ldots, \ell_{n-1}, \ell_n = \ell)$ is a sequence of routers and edges that I expect the route to travel across. I require that it represents an actual topological path in the network: if $\ell_i = R \in \textsc{Routers}$ then for some N, $\ell_{i+1} = R \to N$, and if $\ell_i = R \to N$, then $\ell_{i+1} = N$. For example, $ISP1 \to R1, R1, R1 \to R3, R3, R3 \to Customer$ is a path in the network from Figure 3.1. The last location ℓ_n must be the location ℓ of the end-to-end property that I are verifying.

The constraints $C_1 \ldots C_n$ are properties that represent the set of "good" routes that reach each ℓ_i along the path. They play a role analogous to the local invariants I_{ℓ_i} for proving safety properties, described earlier. The property C_1 for the first location in the path is simply assumed to hold; in practice it is usually an edge coming from an external router, in which case it is not possible to prove whether it sends a route. Rather, the best I can do is prove that if that router sends a "good" route, then it will eventually reach its intended destination in the network.

4.2.2 Local Checks

The checks for liveness can be broken up into two parts: checks that prove propagation along the given path, and checks that prove there is no interference from outside routes.

Propagation along a path: These checks are analogous to the Import and Export checks performed for safety verification, but they are only checked along the given path. Together they ensure that the import and export filters along the path $(\ell_1, \ldots, \ell_n)$ do not drop "good" routes. Specifically, for all valid traces T and $i < n$:

If $\ell_i = R \in \textsc{Routers}$, then:

$$C_i(r) \wedge r' = \texttt{Export}(R \to N, r) \implies r' \neq \textsc{Reject} \wedge C_{i+1}(r')$$

and if $\ell_i = R \to N \in \textsc{Edges}$, then:

$$C_i(r) \wedge r' = \texttt{Import}(N \to R, r) \implies r' \neq \textsc{Reject} \wedge C_{i+1}(r')$$

No interference: Next, I need to verify that it is not possible for a router along the path to select a "bad" route with the same prefix as a "good" route. Let $\texttt{Prefix}(C_i)$ refer the set of prefixes with at least one route in C_i:

$$\{p \mid p = \texttt{Prefix}(r) \wedge r \in C_i\}$$

Then at each router ℓ_i along the path I must prove the following safety property:

$$(\ell_i, \texttt{Prefix}(r) \in \texttt{Prefix}(C_i) \implies C_i(r))$$

These properties can be proven using our existing approach for proving safety properties (chapter 3, given appropriate local invariants.

Implying the network property: The above checks ensure that all of the local C_i constraints in fact hold. Finally, Lightyear generates a local check that $C_n \subseteq P$, similar to the analogous check for safety properties, to ensure that the local constraints imply the desired end-to-end liveness property.

4.3 Proof of Correctness

In this section, I prove the correctness of the modular checks for liveness properties.

Theorem: Given the following:

- The network configurations
- A liveness property (ℓ, P)
- A path $S = (\ell_1, \ell_2, \ldots, \ell_{n-1}, \ell_n = \ell)$
- A constraint for each location $C_1 \ldots C_n$

For all valid traces T, if all of the following are true:

1. all checks (propagation, no interference) pass
2. there exists r such that $\texttt{recv}(\ell_1, r) \in T \wedge C_1(r)$
3. for all r, $C_n(r) \implies P(r)$
4. there are no link failures along the path

then there exists r' such that either:

- $\ell \in$ ROUTERS and there exists r' such that $\texttt{slct}(\ell, r') \in T$ and $P(r')$ holds, or
- $\ell \in$ EDGES and there exists r' such that $\texttt{frwd}(\ell, r') \in T$ and $P(r')$ holds

Proof: Consider a valid trace T. By the assumption, there exists r_1 such that $\texttt{recv}(\ell_1, r_1) \in T$ and $C_1(r_1)$

There must exists at least one router $R = \ell_j$ and a route r_j such that $\texttt{slct}(R, r_j)$ is in the trace and $\texttt{Prefix}(r_j) = \texttt{Prefix}(r_1)$. If there are no routers outside the path that have their routes accepted then $r_2 = \texttt{Import}(\ell_1, r_1)$ is the most preferred route at ℓ_2, so $\texttt{slct}(\ell_2, r_2)$ will be in the trace. If there are routers outside the path that have their routes accepted, then by the no interference check, it must be that the router accepted at ℓ_j will satisfy $C_j(r_j)$.

Consider the last router that accepts a route from a neighbor outside the path. I will use induction to show that all locations ℓ_i between it and the end will have a route satisfying C_i:

Base case: Take the last router $R = \ell_j$, where there exists r_j, C_j such that $\texttt{slct}(\ell_j, r_j) \in T$ and $C_1(r_j)$. I have shown above that there must be one.

Inductive step: If $\ell_i = R \in$ ROUTERS, then I know that $\texttt{slct}(\ell_i, r_i) \in T$ and $C_i(r_i)$ from the inductive hypothesis. I want to show that there exists r_{i+1} such that $\texttt{frwd}(\ell_{i+1}, r_{i+1}) \in T$ and $C_{i+1}(r_{i+1})$. This is true because:

- let $r' = \texttt{Export}(\ell_{i+1}, r_i)$
- $\texttt{slct}(\ell_i, r_i) \in T$ and $C_i(r_i)$ (from the inductive hypothesis)
- $r' \neq$ REJECT and $C_{i+1}(r_{i+1})$ (from the propagation check)
- $\texttt{frwd}(\ell_{i+1}, r_{i+1}) \in T$ (from the liveness axiom)

If $l_i = N \to R \in$ EDGES, then I know that $\texttt{frwd}(\ell_i, r_i) \in T$ and $C_i(r_i)$, and I want to show that there exists r_{i+1} such that $\texttt{slct}(\ell_{i+1}, r_{i+1}) \in T$ and $C_{i+1}(r_{i+1})$. This holds because:

- let $r_{i+1} = \texttt{Import}(\ell_i, r_i)$
- $\texttt{recv}(\ell_i, r_i) \in T$ (from liveness axiom given no link failures)
- $r_{i+1} \neq \textsc{Reject}$ and $C_{i+1}(r_{i+1})$ (from the propagation check)
- I know that R and any router after R in the path did not accept any routes from any neighbors not in the path, so $N \to R$, so I know $\texttt{slct}(\ell_{i+1}, r_{i+1}) \in T$ and $C_{i+1}(r')$

From this, I know that at ℓ_n, there exists a route r_n such that $C_n(r_n)$ and either $\texttt{frwd}(\ell_n, r_n) \in T$ or $\texttt{slct}(\ell_n, r_n) \in T$. $C_n(r_n) \implies P(r_n)$, which is what I wanted to prove.

4.4 iBGP Full Mesh

The technique described in the previous sections is not complete, in the sense that there are networks where a liveness property holds but it is not possible to prove it. One special case is the iBGP full mesh. An iBGP full mesh has two differences from other BGP topologies.

1. Each router in an AS is connected to every other router in the AS via an iBGP session
2. When a route is learned from an iBGP session, it is not advertised to any other iBGP neighbor. It is only advertised to eBGP neighbors[1].

This mechanism allows routes learned from outside the AS to propagate within the AS without looping or using additional fields like AS path.

To see why the liveness checking approach from the previous sections does not work for an iBGP full mesh, consider the network in figure 4.1. Suppose I want to check the property that if DataCenter1 sends a route to Core1, then eventually Core2 accepts a route.

[1]Normally, a route learned from an iBGP session is also advertised to route reflector clients, but a full mesh should not have any client links

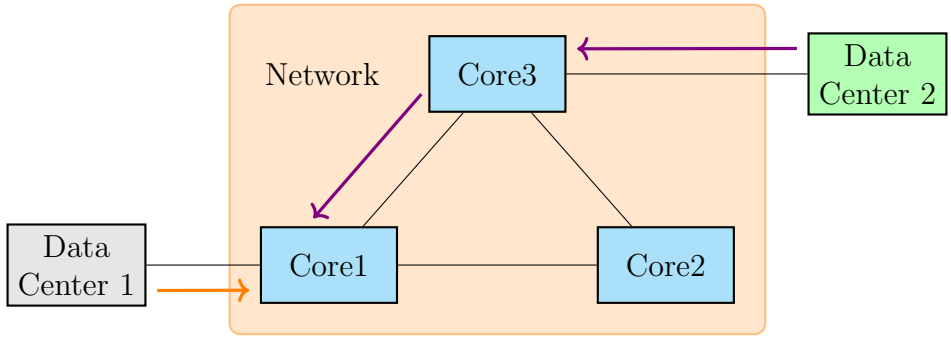


Figure 4.1: BGP topology with an iBGP full mesh and interfering BGP advertisements.

To model the semantics of iBGP, I can consider each route to have an *EdgeType* that is either `eBGP`, `non-client (iBGP)` , or `client (iBGP)`. This field represents the type of edge from which that the router received the route. This can be modeled as an additional attribute that is set by each import policy in the network. The rule for iBGP is that a route r with $\texttt{EdgeType}(r) = \texttt{non-client}$ is dropped when exported to another non-client iBGP neighbor. This can be modeled as part of the export policy.

One can try to use the earlier liveness checking approach by specifying the path as $DataCenter1 \rightarrow Core1 \rightarrow Core2$. In order for a route r at Core1 to be exported to Core2, it must be that $\texttt{EdgeType}(r) \neq$ non-client. This is part of the constraint that must be specified by the user, and it is not possible to specify a constraint where this does not hold.

Now, consider the case where Core3 selects a route s' after importing a route s from DataCenter2. This will get sent to Core1. If this route is higher in priority than the route received from DataCenter1, it will get selected. However, this does not satisfy the constraints because that edge has $\texttt{EdgeType}(r) =$ non-client. However, since Core3 also sends a route to Core2, it is possible that the property still holds. Thus, I get a case where the property holds because Core2 does accept a route, but I cannot prove it using the earlier method.

Intuitively, this is a case where routes coming from Core3 is interfering with routes coming from Data Center 1. However, this is expected in an iBGP full mesh, where most routes are rejected, and the correctness depends on the special topology in which all pairs of routers

are connected. This is unlike other BGP networks where routes are typically accepted by default, and attributes like AS path and the policies affecting them play a larger role in the correctness. In general, it is likely not possible to verify policies without checking all policies among the iBGP routers, which the approach to checking liveness in the previous sections does not do.

4.4.1 Alternative Checks for iBGP full mesh

However, it is possible to check the correctness of an iBGP full mesh though it requires different checks and some additional assumptions about the network. Below I present an alternative that works for an iBGP full mesh. The main difference is that liveness has to be checked in every direction between each pair of iBGP neighbors for eBGP or iBGP client routes. If this hold, then the global property should hold even if iBGP non-client routes are dropped.

If all of the following are true for routers $R_1 \dots R_n$ and constraints P:

1. Routers $R_1 \dots R_n$ are connected in a full mesh of iBGP connections and there are no other non-client iBGP connections on those routers.

2. For all pairs of routers R_i, R_j, and all routes r, r', if $P(r) \wedge r' = \texttt{Export}(R_i \rightarrow R_j, r) \wedge \texttt{EdgeType}(r) \neq$ non-client then $r' \neq \textsc{Reject} \wedge P(r')$

3. For all pairs of routers R_i, R_j, and all routes r, r', if $P(r) \wedge r' = \texttt{Import}(R_i \rightarrow R_j, r)$ then $r' \neq \textsc{Reject} \wedge P(r')$

4. A safety property for all R_i that if they select or originate a route r then either:

   (a) $P(r)$

   (b) the prefix of r does not equal that of any route satisfying P

Then if a router R_i receives and imports a route r with $P(r)$ from an eBGP or iBGP client neighbor, then all routers $R_1 \dots R_n$ will select a route r' with $P(r')$

4.4.2 Proof

Statement: For all valid traces T, if there exists i, N, r, r' and the following are true:

- R_i is in the full mesh
- $R \to N$ is an eBGP or iBGP client connection
- $\texttt{recv}(N \to R_i, r) \in T$
- $P(r')$ where $r' = \texttt{Import}(N \to R_i, r)$
- all checks pass

Then for all $k \in [1, n]$, there exists r_k such that $\texttt{prefix}(r_k) = \texttt{prefix}(r) \wedge \texttt{slct}(R_k, r_k) \in T \wedge P(r_k)$

Proof: If $\texttt{recv}(N \to R_i, r) \in T$, then there exists r_i such that $\texttt{slct}(R_i, r_i) \in T$ with the same prefix and $P(r_i)$ holds (because of the safety check). There are two cases:

If $\texttt{EdgeType}(r_i) \neq$ non-client, then for all $k \neq i$,

1. for $r_{i \to k} = \texttt{Export}(R_i, R_k, r_i)$, $r_{i \to k} \neq \textsc{Reject} \wedge P(r_{i \to k})$ (because of local check on export filters)
2. $\texttt{frwd}(R_i \to R_k, r_{i \to k}) \in T$ and $\texttt{recv}(R_i \to R_k, r_{i \to k}) \in T$ (liveness axioms and assumption of no link failures)
3. there exists a route r_k such that $\texttt{slct}(R_k, r_k) \in T$ (because there must be at least one received route with higher priority than all others)
4. $P(r_k)$ holds (because of safety check)

If $\texttt{EdgeType}(r_i) =$ non-client, then:

1. there exists R_j and $r_{j\to i}$ such that $\texttt{recv}(R_j \to R_i, r_{j\to i}) \in T$ and $\texttt{frwd}(R_j \to R_i, r_{j\to i}) \in T$ (because of safety axioms)

2. there exists r_j such that $\texttt{slct}(R_j, r_j) \in T$ (because of safety axioms)

3. $P(r_j)$ holds (because of safety checks)

4. for all $k \in [1, n] \setminus \{i, j\}$, and for $r_{j\to k} = \texttt{Export}(R_j, R_k, r_j)$, $r_{j\to k} \neq \textsc{Reject} \wedge P(r_{j\to k})$ (because of local check on export filters)

5. $\texttt{frwd}(R_j \to R_k, r_{j\to k}) \in T$ and $\texttt{recv}(R_j \to R_k, r_{j\to k}) \in T$ (liveness axioms and assumption of no link failures)

6. there exists a route r_k such that $\texttt{slct}(R_k, r_k) \in T$ (because there must be at least one received route with higher priority than all others)

7. $P(r_k)$ holds (because of safety check)

CHAPTER 5

Lightyear: Results of BGP Modular Checks

This chapter shows the results from running Lightyear on configurations from several networks. Section 5.1 describes using modular checks on the wide area network of a major cloud provider. Section 5.2 compares the scaling of Lightyear and Minesweeper for synthetic network configurations, and section 5.3 describes the experiences from running Lightyear on configurations from a university network.

5.1 Cloud Provider WAN

I used Lightyear to modularly verify properties of the wide-area network (WAN) of a major cloud provider, containing hundreds of routers and several thousands of peering sessions. In doing so, I show that: (1) important behavioral properties in real-world networks can be expressed in Lightyear; (2) these properties can be proven through a combination of modular checks; (3) this approach scales, allowing properties to be verified quickly; and (4) if a local check does not succeed, it produces actionable information, indicating a bug in either a specific route map or a specific local invariant. To our knowledge no prior tool that verifies properties of all possible external announcements from neighbors has been demonstrated to scale to such a size.

I used Lightyear to verify two classes of properties that the wide-area network must satisfy. In all cases I determined the intended network behavior by inspecting the configurations and talking with the network operators, and the local constraints were written based on that intent. This process was typically iterative. That is, I would write an initial

Type	Locations (l)	Logical Formula (I_l)	Description
End-to-end Property	Any R in network	$FromPeer(r) \implies$ $\textsc{Prefix}(r) \notin \textsc{Bogons}$	*Bogon prefixes from peers should not be accepted*
Network Invariants	$R \in \textsc{Routers}$	$FromPeer(r) \implies$ $\textsc{Prefix}(r) \notin \textsc{Bogons}$	*Bogon prefixes from peers should not be accepted at routers*
	Internal edges R1 $\to$ R2	$FromPeer(r) \implies$ $\textsc{Prefix}(r) \notin \textsc{Bogons}$	*Bogon prefixes from peers should not be sent along edges*
	Other	$True$	*Edges to and from external peers are unconstrained*

(a) End-to-end property and network invariants needed to verify that the network does not accept bogons from external peers.

Type	Locations (l)	Logical Formula (I_l)	Description
End-to-end Property	$R \notin \textsc{Region}$	$FromRegion(r) \implies$ $\textsc{Prefix}(r) \notin \textsc{ReusedIPs}$	*Routers outside a region should not accept routes with reused addresses from that region*
Network Invariants	$R \in \textsc{Region}$	$FromRegion(r) \wedge$ $\textsc{Prefix}(r) \in \textsc{ReusedIPs} \implies$ $\textsc{RegionalComms} \cap \text{Comm}(r) = \{C\}$	*Routes with reused addresses are tagged with a community for that region and no other region*
	$R \notin \textsc{Region}$	$FromRegion(r) \implies$ $\textsc{Prefix}(r) \notin \textsc{ReusedIPs}$	*Routers outside a region should not accept routes with reused addresses from that region*
	R1 $\to$ R2	I_{R1}	*Edges have same invariant as sending router*
	E $\to$ R	$\text{Comm}(r) = \emptyset$	*Edges from external peers have no communities*

(b) End-to-end property and network invariants needed to verify that reused addresses are not accepted by any router outside the region.

Type	Locations (l)	Logical Formula (I_l)	Description
End-to-end Property	$R_2 \in \textsc{Region}$	$FromRegion(r) \wedge$ $\textsc{Prefix}(r) \in \textsc{ReusedIPs}$	*R_2 inside a region eventually accept a route with reused addresses from that region*
Assumption	Edge from data center $D \to R_1$	$FromRegion(r) \wedge$ $\textsc{Prefix}(r) \in \textsc{ReusedIPs}$	*Assume there is a route from the data center to R_1 with a reused prefix*
Path Constraints	$R_1, R_2, R_1 \to R_2$	$FromRegion(r) \wedge$ $\textsc{Prefix}(r) \in \textsc{ReusedIPs} \wedge$ $\textsc{RegionalComms} \cap \text{Comm}(r) = \{C\}$	*R_1 and R_2 eventually select a route with reused prefixes and the regional community*

(c) End-to-end property and path constraints needed to verify that reused addresses are eventually selected by each WAN router in that region. I assume that the route flows from the data center along the path $D \to R_1 \to R_2$

Table 5.1: End-to-end properties and network invariants for three use cases in the WAN.

property specification and its set of local invariants based on our current understanding of how the network operates. If Lightyear reported violations of local checks, I would inspect the counterexamples and discuss with operators, either determining that the bugs are real errors or identifying special cases that led to refined local invariants and (sometimes) refined end-to-end property specifications.

Implementation: I implemented Lightyear as a tool in C#. The tool parses and extracts the BGP policy along with import and export route maps from each configuration, while supporting common attributes of BGP routes such as communities, AS paths, MEDs,

local preference, along with most route map features, like matching on or setting attributes. The tool allows users to provide local invariants written as a C# function using the Zen constraint solving library [58], and to specify the routers and policies of interest. The Zen library translates the functions into SMT formulas that are solved by Z3 [15]. If there are any violations of the property, the tool returns counterexamples, showing a specific route map, and a combination of attributes that leads to a violation.

Internet Peering Policies: I implemented local invariants to verify 11 properties in Lightyear to ensure that different kinds of "bad" routes are never accepted from peers. Each of these properties can be expressed as a safety property on each node R in the network of the following form:

$$(R, \{r \mid FromPeer(r) \implies Q(r)\})$$

with different properties $Q(r)$. These include properties like not accepting bogons or routes with invalid AS paths. An example of the invariants for the no-bogons property is shown in Table 5.1a. The network has a set of Internet *edge routers*, that peer with Internet service providers, other cloud providers, and customers, and so act as gateways between the cloud provider and the Internet. The wide-area network ensures that "bad" routes are not admitted by filtering them at all of the Internet edge routers.

As mentioned earlier, running Lightyear to check these properties is an iterative process, which involves refining the local constraints based on operator feedback. In the end, through this process Lightyear identified 11 actual configuration errors. These included cases where a route map denied more traffic than intended, and inconsistencies between the filters of edge routers that are intended to have similar behavior. All of the findings were latent bugs that did not have an immediate impact, but could become impactful on additional failure. Further, because Lightyear is sound the operators can be sure that these are the *only* violations of the desired end-to-end properties. As of this writing, all the bugs are prioritized for fixing by network engineers.

Verification with Lightyear is highly scalable. The maximum time that it took Lightyear to sequentially run all of the local checks for any single property was 15 minutes, across all devices in the network. As another data point, an automation that sequentially ran the local checks for four of the properties across all of the hundreds of edge routes took a total of 16 minutes. Given that each of these checks can be run independently on each device configuration, it would also be easy to parallelize these checks in the future in order to scale horizontally for large number of devices.

While using Lightyear to verify these 11 properties, I also learned best practices for writing properties. Initially, I combined multiple properties into a single property for Lightyear to check. However, I found that writing multiple simpler properties, with associated simpler local constraints, was not only easier to write and debug but also was usually faster to run, since the constraints are simpler for the underlying SMT solver to process.

Proper IP Reuse: In the second use case, Lightyear was used to verify proper usage of reused IPs within the network. The cloud network is partitioned into dozens of *regions*, and some private IPv4 addresses are reused in different regions. There is a safety property that traffic sent to these private addresses must stay within the region, and also a liveness property that routes to reused addresses are advertised to other WAN routers in the same region. I verified both of these properties for all regions in the network.

The safety property to verify is as follows, for each router R that is *not* part of the region of interest:

$$(R, \{r \mid FromRegion(r) \implies \textsc{Prefix}(r) \notin \textsc{ReusedIPs}\})$$

Here $FromRegion(r)$ is a ghost variable that is set to true only on routes coming from external routers in the particular region, and ReusedIPs is the set of prefixes that are reused. The liveness property requires that in each region, a route with a reused prefix from the data center routers can reach all other routers in that region, possibly going through one intermediate router. That is, for every pair of WAN routers R_1 and R_2 in the same region,

if R_1 is connected to a data center router D, then routes with a reused prefix can travel $D \to R_1 \to R_2$.

The WAN enforces these properties by tagging routes for reused IP addresses with a region-specific community C when they are received from data centers. Routers in the same region then accept routes tagged with that community, while routers in other regions reject them. The local constraints I used to verify the safety and liveness properties are shown in Table 5.1b and 5.1c respectively. One subtlety is that routes to reused IP addresses in the region of interest must not only have the community C, but they also must not be tagged with any other region's community. Otherwise, these routes could be accidentally accepted by other regions. The local constraints validate this property, and the WAN enforces it by deleting all communities on routes coming from the data centers, before adding the community C.

The communities used in each region were documented in a metadata file, which made it easy for me to write the local constraints for each region. In one case, Lightyear found a violation where a region used a community that was not present in the metadata file. The operators acknowledged that this was a bug that could cause some traffic to be redirected. In every other case, Lightyear was able to verify both the safety and the liveness properties.

5.2 Scaling Experiments

To illustrate the scaling benefits of modular checking, I compared Lightyear with Minesweeper [6] on synthetic test cases. For a fair comparison, I created a new implementation of Lightyear that is built on top of the same parser and constraint generation system as Minesweeper. I use a BGP full mesh where each router is connected to one external neighbor through eBGP and all other routers through iBGP. This leads to a total of N^2 edges in a network of size N. The network's configuration is relatively simple, with each eBGP connection using only prefix and community filters. I checked a no-transit safety property, similar to the example in Figure 3.1.

Figure 5.1 provides details on these results by comparing the number of SMT variables and constraints generated by each tool, as well as the amount of time used to solve the SMT constraints compared to the total computation time. I used 10 different communities for 10 routers and 20 different communities when testing 20 or more routers. As the network size increases, Minesweeper requires several orders of magnitude more SMT variables and assertions than the maximum number required by Lightyear for any local check. As a result, SMT solving time dominates the run time of Minesweeper and is the limiting factor on its ability to scale, while for Lightyear the solving time is a relatively small portion of the total time. Minesweeper does not terminate within two hours when run on a network of size 40, while Lightyear verifies a network of size 100 in 5.5 minutes.

5.3 University Network Results

Our experiences running Lightyear on the campus network of a large university show how the local counterexamples produced can aid in refining the network invariants and localizing errors.

For this case, I use the implementation of Lightyear built using the technology of Minesweeper. Parsing network configurations and determining the topology and policy is done using Batfish [19]. Since Batfish converts the configurations into a vendor-independent format, this allows Lightyear to handle configurations from numerous different vendors, such as Cisco and Juniper.

Properties and invariants are specified as Python functions mapping routers and edges to BGP route constraints. Our implementation allows users to define one boolean ghost variable, with Python functions to specify how route import, export, and origination should affect the variable's value. Currently I allow ghost variables to be transformed based on the location in the network, but not based on other route attributes. For example, users can specify that an import filter changes the value of a variable to true, but users cannot specify that the filter transforms the variable to true if the prefix is 10.0.0.0/8. This is not

a fundamental limitation but simplifies our implementation and has been sufficient for our use cases.

The code for generating local checks from the user-provided invariants is written in Python. The local checks rely on the symbolic modeling of BGP policies [6]. Specifically, BGP route announcements are modeled as symbolic variables, and BGP route maps in the configurations are modeled as SMT constraints on these symbolic variables, and the Z3 SMT solver [15] is used to perform the local checks. Minesweeper's symbolic analysis supports many common features of BGP route maps, like communities, local preference, and MEDs, as well as other configurable aspects of BGP like route origination. However, it does not support reasoning about AS-path filtering, which is necessary for one of our use cases. For that, I use an alternate backend for symbolic route analysis from Batfish that is based on binary decision diagrams (BDDs) [5].

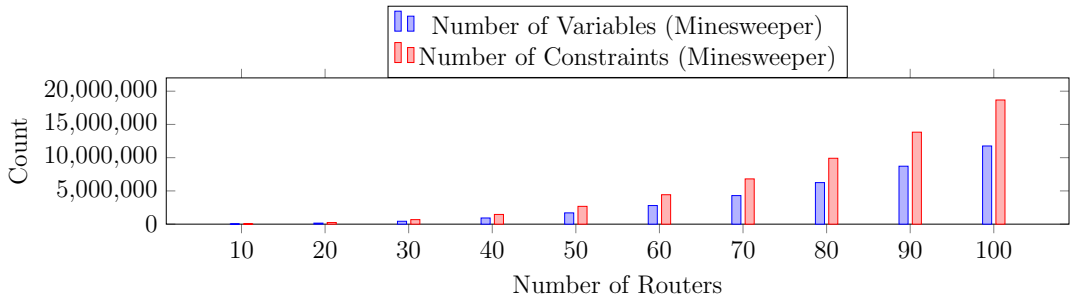
I used Lightyear to verify the property that only university-owned address blocks can be advertised to the ISPs. This property is shown at the top of Table 5.2. Our initial assumption was that this property only depends on prefix filtering done by the outbound filters on the border routers, so I provided the network invariants shown in the middle of the table. However, Lightyear produced counterexamples for the local checks on the edges from the border routers to the ISPs, since the outbound filters allow advertisements for other prefixes as long as they are tagged with the well-known blackhole community [31], which tells the ISPs to blackhole that traffic. However, by inspection I found that this mechanism was not being used currently — the blackhole community is not added to any announcements. Hence I were able to still prove the desired end-to-end property, using the refined set of invariants shown at the bottom of Table 5.2, where SCOMMS represents the set of blackhole communities.

To demonstrate our approach's utility in error localization, I used Lightyear again after injecting three errors into the configurations: (1) adding a policy that attaches the blackhole community (2) changing a policy from removing communities to propagating them, and

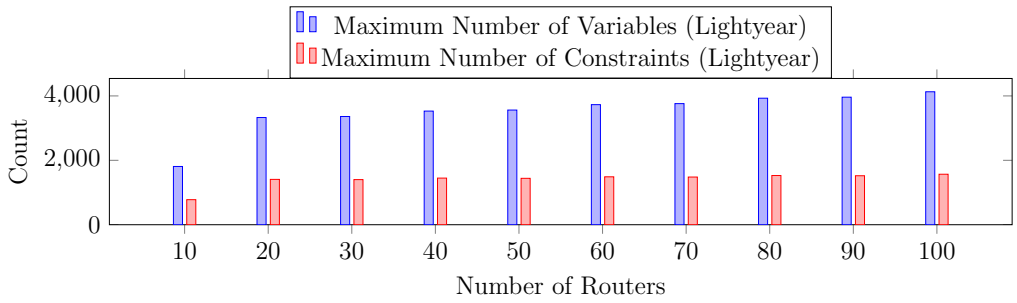
Type	**Locations (l)**	**Logical Formula (I_l)**	**Description**
End-to-end Property	Edges to ISP R → P	$\textsc{Prefix}(r) \in \textsc{UniPrefixes}$	*University routers only send prefixes for its owned address blocks*
Initial Network Invariants	Edges to ISP R → P	$\textsc{Prefix}(r) \in \textsc{UniPrefixes}$	*University routers only send prefixes for its owned address blocks*
	Other	True	*Other locations are unconstrained*
Refined Network Invariants	Edges to ISP R → P	$\textsc{Prefix}(r) \in \textsc{UniPrefixes}$	*University routers only send prefixes for its owned address blocks*
	Edges from external $P \to R$	True	*Routes from external neighbors are unconstrained*
	Other	$\text{Comm}(r) \cap \textsc{SComms} = \emptyset$	*Routes at other locations inside network are not tagged with specific communities*

Table 5.2: The end-to-end property and network invariants needed to verify that the university only advertises its own aggregated networks. Both the initial guess and the refined invariants are shown.

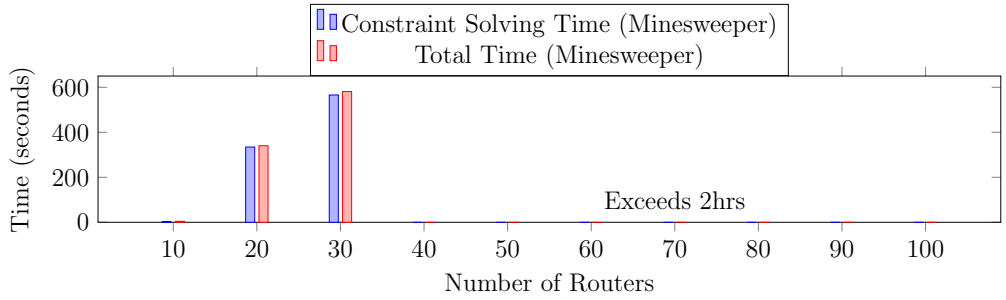
(3) changing the filter at the border router to allow a non-university address block. Each of these errors causes the local checks to fail for the edges using the erroneous routing policy. Further, the counterexample provided by Lightyear for each error provides the relevant information to understand the error, such as the prefix being advertised and whether communities are being added or propagated. For example, the first error causes the local check to fail because I cannot prove that $\text{Comm}(r) \cap \textsc{SComms} = \emptyset$ for all routes r that pass the filter, and the counterexample route includes the blackhole community.



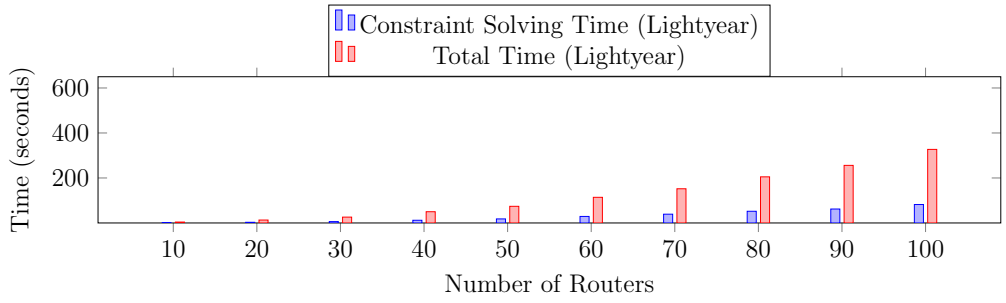
(a) Number of variables and constraints generated by Minesweeper for synthetic networks.



(b) The maximum number of variables and constraints in any single local check generated by Lightyear.



(c) Time used by Minesweeper to verify a property of synthetic networks. Runtime for networks with 40 routers or more exceeds two hours, not including time to parse configurations.



(d) Time used by Lightyear to verify a property in synthetic networks, not including time to parse configurations.

Figure 5.1: Comparing Lightyear and Minesweeper on synthetic networks of various sizes.

CHAPTER 6

Related Work

This chapter compares my work on modular verification, described in previous chapters, to other network verification tools that have been developed. Previous tools for verifying network properties roughly fall into three categories: Tools checking specific properties on individual configurations, tools that model the data plane but not the control plane, and tools that model the control plane. The tools that model more complex behaviors typically struggle to scale or provide usable results, while tools that can scale or provide usable results can only check more modest properties.

Earlier works in network verification checked specific properties in individual router configurations. rcc [17] detects faults in BGP configuration using a high-level specification. It considers many common causes of error in BGP and checks constraints that typically imply good behavior. rcc does not have any problem scaling as it runs on one router as a time, but it can only check simple configuration properties and is neither sound nor complete. FIREMAN [57] checks firewall properties by modeling firewall actions as binary decision diagrams (BDDs). It checks for inconsistencies, inefficiencies, and certain policy violations (e.g. allowing bogons). These, and other tools [37] had many drawbacks. They could only reason about one router at a time, they did not check more complex features of router configuration like prefix lists and router maps, and they did not reason about network state. Thus, they could not answer reachability queries like "Could a packet reach from point A to point B in the network?"

To answer such queries, many approaches applied formal method techniques to reason

about data plane behavior. These approaches model the network as a web of switches. Each switch maintains a forwarding table that matches the packet entering the switch with an output port. Data plane verification techniques tried to answer queries about whether packets can or cannot reach from node A to node B or whether packets will loop. Unlike simulation, they attempt to answer these queries for all packets, and much of the challenge comes from figuring out how to deal with the large space of packets and packet transformations. One approach was Anteater [36] which models data plane behavior as a SAT formula. Another approach was Header Space Analysis (HSA) [29] which used ternary simulation for better scaling. It models packet headers as ternary bit vectors with each bit as either 1, 0, or unknown, and simulates the effect of the network on these headers. Later approaches build on these ideas with alternative approaches to speed up computation [54, 39], or allow for incremental verification [30, 28, 22].

These data plane analysis techniques have been successful in discovering errors in networks. They are successful in that they can scale verification to forwarding rules across the network, with relatively simple specifications. However, this type of verification requires known forwarding tables. Forwarding tables are a result of dynamic routing processes in the networks, so to use these tools, network operators need to supply a consistent snapshot of the network forwarding state in addition to the router configurations. This means that data plane checking has to be done retrospectively. They can detect errors present in networks, but in general, they cannot determine how a configuration change would affect network behavior. Much of the challenge in real-world networks is change validation, that is, guaranteeing the correctness of a change to the network configuration before it enters the production system. Data plane verification alone cannot solve this problem. In addition, forwarding rules are a product of the configuration, but if there is an error in a forwarding rule, it is not obvious what configuration caused that error.

The next step was verification of the network control planes that generate the forwarding tables used in the data plane. Checking correctness of the control plane would allow errors

to be caught before a configuration change is made. In traditional networks, routers run distributed routing protocols like BGP and OSPF to compute the forwarding tables. Operators configure attributes (e.g. link costs) and policies (e.g. route filters) to control the operation of the protocols and influence the result, but the final routing table generated also depends on the BGP messages from neighbors and the link failures within the network.

Batfish [19] was one attempt at checking control plane properties. Batfish takes router configurations and simulates routing protocols for a fixed environment, producing forwarding tables that can then be used with data plane verification techniques. In a similar vein, Microsoft's Crystalnet [34] emulates networks using real device firmware, allowing for greater accuracy when simulating network changes before they occur. This approach to checking configurations by simulating the routing process can scale but it is limited to a single environment. It cannot handle arbitrary environments, so it cannot verify the correctness of configurations in all cases.

In order to verify configurations for all environments, control plane verifiers were developed using symbolic modeling. Minesweeper [6] and Bagpipe [53] model the control plane, using SMT approaches, allowing them to consider different environments and reason in the presence of link failures. Other approaches [20, 42, 3, 7] build use alternative techniques to speed up computation. My work falls into this category of control plane verification, with an additional focus on the scalability and localization.

6.1 Comparisons to Campion

At a high level, my work on Campion differs from prior work in network verification in two ways. First, we target verifying behavioral equivalence of two router configurations, while prior work typically targets network-wide reachability properties. Second, we localize identified errors to both relevant headers and configuration lines even for the control plane; most prior work in control plane verification simply provides individual concrete counterexamples.

Data Plane Verification Tools: Many tools verify reachability properties of a net-

work's data plane, including its ACLs and forwarding tables [36, 29, 30, 54, 25, 4, 35]. Several tools focus on ACLs [37, 51] and localize errors to ACL lines [25, 29, 51, 24]. The closest to Campion is `netdiff` [16], a tool for checking data plane equivalence in networks. It uses a similar symbolic execution approach, but it focuses on the data plane. Campion extends these capabilities to perform configuration localization for the control plane. HeaderLocalize and StructuralDiff have no analogue in `netdiff`.

Control Plane Verification: Control plane verifiers can be adapted to perform router equivalence checking, as we showed for Minesweeper [6] in section 2.1. However, when verification fails, these tools only provide individual, concrete counterexamples, while Campion localizes to both headers and configuration text. As we have seen by the experiment in section 2.1, even if we extend Minesweeper to produce multiple counterexamples it is still not able to quickly find all errors. Further, this still leaves the question as to which parts of the text caused each error. There exist a tool extending Minesweeper to localize errors by leveraging an SMT solver's ability to provide *unsatisfiable cores* when verification fails [46]. The approach localizes errors to specific SMT constraints, but not to configuration lines or headers. Campion leverages the BDD encoding of ACLs and route maps from Bonsai [7], which uses BDDs to perform network abstraction, not router differencing or debugging. Campion's structural checks are reminiscent of `rcc` [17], but Campion's checks are designed to ensure behavioral equivalence and to do so without incurring additional false positives over a modular semantic check.

Outlier Detection: Benson *et al.* [11, 10] infer data-plane reachability specifications from a network's forwarding tables and use these specifications in part to identify outliers. However, they only consider the data plane and cannot localize back to the original configurations. SelfStarter [27] infers parameterized configuration templates for ACLs and route maps and uses them for outlier detection. This approach uses sequence alignment and so requires router configurations to be structurally similar. Further, SelfStarter localizes configuration text but cannot localize headers.

Equivalence Checking: Equivalence checking is an old idea beyond networks, and the SemanticDiff algorithm is similar in spirit to prior work. For example, Ramos *et al.* [44] perform equivalence checking of two C functions via pairwise comparisons of execution paths. Because network ACLs and route maps are loop-free, Campion is exhaustive, finding *all* differences and localizing to *all* IP prefixes; equivalence checking of software is undecidable in general.

6.2 Comparisons to Lightyear

Control Plane Verification: State-of-the-art approaches to network control-plane verification were summarized in Table 3.1. Unlike Lightyear, these approaches are all *monolithic* — they require joint analysis of the configurations of all nodes — which dramatically limits scalability. Compared to Lightyear, Minesweeper's worst case complexity is exponential in the network size. Other improvements not only reduce generality but are at least quadratic in the network size even when using specialized algorithms. Most approaches make tradeoffs in expressiveness, for example giving up the ability to reason about all possible BGP announcements from neighbors [20, 42, 3, 56]. In contrast, Lightyear's modular approach only requires reasoning about individual BGP route maps in isolation and so is highly scalable. Lightyear also provides guarantees across all possible external announcements and, for safety properties, arbitrary failures.

`rcc` [17], validates important properties of BGP configurations, largely through local checks on individual configuration. However, `rcc` is limited to specific "best practice" policies, and there is no guarantee that the local checks together ensure the desired end-to-end properties.

Closest to my work are recent techniques for modular control-plane verification, Kirigami [49] and Timepiece [50] that use assume-guarantee reasoning for the control plane using local invariants. However, both make a different set of tradeoffs than Lightyear. Kirigami's local invariants require the exact routes that will arrive on a particular edge.

Because these invariants are fully concrete, Kirigami cannot reason about arbitrary route announcements from neighbors or give guarantees in the presence of failures.

Timepiece allows more expressive local invariants than Kirigami, using an explicit notion of time. In Timepiece, routing protocols have discrete, synchronized time steps, and in each step, each router computes the best route among those it receives. This allows it to specify and check temporal-logic properties, but require users to provide complex local invariants for each node explicitly indexed by time. I, however, assume that routes can be sent and arrive in arbitrary orders, and we demonstrate how to specify and check common safety and liveness properties without explicit time.

Another line of work has improved scalability of control-plane verification through forms of *abstraction* [7, 8]: the full network is analyzed monolithically, but irrelevant or redundant configuration information is abstracted away to simplify the analysis. My work is orthogonal to this line of work; the two approaches could be combined.

Data Plane Verification: Data plane approaches generally require joint reasoning about the entire network. A recent exception is RCDC [23], which modularly verifies global reachability contracts in a data center via local checks. However, RCDC is specific to the data center design and does not provide a general framework for decomposing global property checks into local checks. Another approach[40] exploits abstraction, such as symmetries, to scale data-plane verification.

Modular Verification: Assume-guarantee reasoning [26, 41] enables modular verification in other domains. A global property is modularized by providing each system component with local invariants that it must satisfy, assuming other components satisfy their invariants. Lightyear applies this methodology to networks to generate the local checks that each BGP policy must satisfy.

Verification often requires identifying *inductive invariants*, properties that hold over some unbounded space of system states, such as the iterations of a loop [21]. Such invariants arise

naturally in networks and enable many locations to use the same local invariant. Typically, a small set of nodes establishes an inductive invariant (e.g., by attaching a community), and this invariant holds through the network as long as other nodes "do no harm" (e.g., never remove communities).

CHAPTER 7

Conclusion

In this thesis, I have presented multiple approaches to improving network verification using modular techniques. Most existing network verification tools, especially for the control plane, check properties holistically, modeling the network as a whole in order to reason about network-wide properties. Instead, I am proposing that many of these checks can be done while inspecting smaller modules of the network. This leads to better scalability and improved localization, allowing for network verification to become a practical tool in real world networks.

In chapter 2, I presented Campion, a tool for debugging router configurations intended to be behaviorally equivalent. Unlike prior work, Campion uses modular structural or semantic checks to localize errors to the affected message headers and relevant configuration lines. Our experience with a cloud provider and a university indicates that Campion satisfies a real need by localizing crucial errors. Campion exploits the modular structure of configurations to break up complex checks of whole router behavior into smaller per-component checks. This "bottom up" style eases localization, sidesteps reasoning about the routing protocols, and allows simple structural checks to often be used without additional loss of precision.

In chapters 3, 4, and 5, I presented Lightyear, which can verify network-wide reachability tools using local checks. I show the necessary inputs and checks needed to verify safety and liveness properties. To my knowledge it is the first network verification tool that can scale to large networks. It has been used at a major cloud vendor and discovered several bugs. Further, Lightyear finesses the need to reason about time to prove safety and liveness,

offering a sweet spot between expressiveness and complexity that has worked well for many desired properties in our network.

None of these capabilities would be possible without exploiting modularity. As in other forms of verification, I believe focusing on modularity will be critical to making real-world network verification and debugging effective. Because of design patterns in the way configurations are written and maintained in well engineered networks, it is possible to reason about policies that depend on many different pieces of the network. There is plenty of structure in the configurations, and that structure has not been adequately leveraged in past works.

7.1 Possible Extensions

The work in thesis has made it possible to modularly check network properties. However, there is more work that can be done building on these ideas. One extension is to learn the invariants and constraints that were used in chapter 3 and chapter 4, so that users do not need to specify them manually. Another possibility is synthesize provably correct configurations using the modular techniques introduced.

7.1.1 Learning Invariants

Chapter 3 and chapter 4 developed the idea of checking network properties using modular checks. These modular checks relied on user-provided invariants and constraints in order to generate the checks on BGP route maps. This is analogous to the idea of inferring invariants in software programs. In software, loop invariants have to be specified in order to check the behavior of programs with loops. However, there has been work to automatically infer these invariants. If these invariants and constraints can be automatically inferred, then the burden on the users who are using the tool would be reduced. Applying this idea to the realm of network router configurations would allow network properties to be proven with less work from the user.

One possible way to do this is to generate candidate constraints using the given network

property and the route map definitions in the network. These can then be refined if the candidate constraints do not generate checks that hold in the network. For example, if the tool wants to generate an invariant for a no-transit safety property where routes received from one neighbor are not sent to the other, it can try to generate an invariant where routes from the first neighbor have either a particular community or a particular ASN in its AS path that is dropped on the export filter to the second neighbor.

One challenge is determining what the relevant BGP route attributes. A large network may use hundreds of different policies using several different attributes, so there would need to be a way to figure out which combinations of prefixes, communities, AS paths, and other attributes are used in implementing this particular property. A network satisfies several properties at the same time, so many attributes and policies are likely unrelated to any single property. A second challenge is determining which locations introduce or preserve a route property. Assuming most properties are implemented so that some route maps establish invariants and some preserve invariants, there needs to be graph algorithms or heuristics that can quickly determine which route maps are used. In the case of liveness properties, this would involve finding a valid path. A third challenge is handling cases when there is an error in the configurations, and the property does not hold. In this case, it is impossible to verify the property, so the tool should help to localize the error in some way.

7.1.2 Configuration Synthesis

Routers in a network have to be configured correctly in order for Internet services to work properly. This configuration is usually done manually by operators or through templates and simple scripts. However, these methods do not ensure the correctness of the configuration. One option is to use verification, which was the subject of this thesis. Another option is automatic synthesis based on operator intents. Users would provide some specification of what the network behavior and a tool would generate the configurations to meet the specification. This can additionally help by making it easier to switch between different

router vendors that use different configuration languages. Previous tools for configuration synthesis have been developed such as Propane/AT [1] or AED [2], but these have not seen wide adoption.

As with network verification, the major challenge is to create a system that is powerful enough to synthesize the policies that are used in real network while still generating configurations that are understandable to the operators. Large networks may have hundreds of routers each with hundreds or thousands of lines of configuration. An incremental synthesis tool would have to be able to determine where to add or remove configuration lines within all of that text, while ensuring that the change satisfies the intended network properties. Ideally, the end result should also be understandable and sensible to a human operator, as the network operators are unlikely to adopt a tool to perform critical change unless they can easily inspect the result to see if it matches their intention. This is hard to do with an automated system since the same high-level policy may be implemented in different ways, and some implementations are more idiomatic than others.

Building the ideas of modularity introduced in this thesis, it may be possible to create a configuration synthesis tool that can handle these challenges. By specifying the desired global property, the tool can make local changes so that the global property can hold. To reduce difficulty on the part of the operators, the tool would have to determine where to place the changes in order to create the most understandable result. To do this, it may have to rely on the insight that routers in the network typically fall into one of several roles, and routers in the same roll typically have similar policies. Significant work would have to be done in order to create a complete and usable system.

APPENDIX A

Proof of Soundness for Modular Equivalence

[Soundness] If networks $\mathcal{N}$ and $\mathcal{N}^*$ are locally equivalent for isomorphism $\mathcal{I}$, then they have the same set of routing solutions.

Proof. The proof is by a reduction to the stable routing problem [7]. First, I show that each protocol $p \in \mathcal{P}$ forms a stable routing problem (SRP). In particular for any given destination router $d \in \mathcal{V}$ advertising initial route d_r, $I(d) \in \mathcal{V}^*$ must also advertise d_r since the protocol-specific advertisement configurations must be the same. Given this, I can construct the SRP $(\mathcal{T}, \mathcal{R}, d_r, \preceq_p, \text{trans})$ for $\mathcal{N}$ and $(\mathcal{T}^*, \mathcal{R}, d_r, \preceq_p, \text{trans}^*)$ for $\mathcal{N}^*$, where:

$$\begin{aligned} \text{trans}(e, r) &= \mathcal{F}_p(\mathcal{C}_p(e), e, r) \\ \text{trans}^*(e, r) &= \mathcal{F}^*_p(\mathcal{C}^*_p(e), e, r) \end{aligned}$$

I further relate the two SRPs with the abstraction (f, h) where $f(e) = \mathcal{I}(e)$ and $h(r) = r$.

The main theorem for abstract SRPs is that of equivalent routing solutions when the abstractions are sound [7]. Thus, I must simply prove that this is a sound abstraction. To do so, I prove each of the sufficient conditions in [7]:

Dest-equivalence. I have $f(d) = \mathcal{I}(d)$ which is the destination router for $\mathcal{N}^*$ and $f(x) \neq I(d)$ for any $x \neq d$ by virtue of $\mathcal{I}$ being an isomorphism.

Orig-equivalence. I have $h(d_r) = d_r$ since h is the identify function, which by construction

is the route used at $\mathcal{N}^*$.

Drop-equivalence. I have $h(r) = r$ since h is the identity function, which trivially satisfies the drop-equivalence requirement that $h(r) = \bot \iff r = \bot$.

Rank-equivalence. By definition, I have $r_1 \preceq_p r_2 \iff h(r_1) \preceq_p h(r_2)$ since h is the identity function.

Trans-equivalence. From the fact that $\mathcal{N}$ and $\mathcal{N}^*$ are equivalent for $\mathcal{I}$, it follows that $\mathcal{F}_p(\mathcal{C}_p(e), e, r) = \mathcal{F}_p^*(\mathcal{C}_p^*(I(e)), I(e), r)$. This means that I have $\text{trans}(e, r) = \text{trans}^*(I(e), r)$ by definition. Substituting the definition of f and h, this gives us the equivalence: $h(\text{trans}(e, r)) = \text{trans}^*(f(e), h(r))$, which is the desired result.

Topology-abstraction. Finally, the topology requirements from [7] are trivially satisfied since $\mathcal{I}$ is a homomorphism.

This result demonstrates that each protocol will compute the same set of routing solutions. Thus the composition of the protocols will also compute and select the same set of routes.

□

REFERENCES

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Aed: Incrementally synthesizing policy-compliant and manageable configurations. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 482–495, New York, NY, USA, 2020. Association for Computing Machinery.

[2] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Aed: Incrementally synthesizing policy-compliant and manageable configurations. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 482–495, New York, NY, USA, 2020. Association for Computing Machinery.

[3] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 201–219, Santa Clara, CA, February 2020. USENIX Association.

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 113–126, New York, NY, USA, 2014. Association for Computing Machinery.

[5] Batfish. https://github.com/batfish/batfish.

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 155–168, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 476–489, New York, NY, USA, 2018. Association for Computing Machinery.

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Abstract interpretation of distributed network control planes. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[9] Ann Bednarz. Global microsoft cloud-service outage traced to rapid bgp router updates, Jan 2023.

[10] Theophilus Benson, Aditya Akella, and David Maltz. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA, 2009. USENIX Association.

[11] Theophilus Benson, Aditya Akella, and David A. Maltz. Mining policies from enterprise network configuration. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 136–142, New York, NY, USA, 2009. ACM.

[12] TODD Bishop. Xbox live outage caused by network configuration problem, 2013.

[13] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A. Seshia, and George Varghese. ddnf: An efficient data structure for header spaces. In Roderick Bloem and Eli Arbel, editors, *Hardware and Software: Verification and Testing*, pages 49–64, Cham, 2016. Springer International Publishing.

[14] Marco Chiesa, Luca Cittadini, Giuseppe Di Battista, Laurent Vanbever, and Stefano Vissicchio. Using routers to build logic circuits: How powerful is bgp? In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, 2013.

[15] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[16] Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Dataplane equivalence and its applications. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 683–698, Boston, MA, February 2019. USENIX Association.

[17] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 43–56, Berkeley, CA, USA, 2005. USENIX Association.

[18] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, 2016.

[19] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, Oakland, CA, May 2015. USENIX Association.

[20] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016*

ACM SIGCOMM Conference, SIGCOMM '16, pages 300–313, New York, NY, USA, 2016. ACM.

[21] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[22] Alex Horn, Ali Kheradmand, and Mukul R. Prasad. Delta-net: Real-time network verification using atoms. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 735–749, USA, 2017. USENIX Association.

[23] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 200–213, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 200–213, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. Automated analysis and debugging of network connectivity policies. Technical Report MSR-TR-2014-102, Microsoft, July 2014.

[26] C. B. Jones. Specification and design of (parallel) programs. In R. E. A. Manson, editor, *Proceedings of IFIP '83*, pages 321–332. IFIP, North-Holland, 1983.

[27] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. Finding network misconfigurations by automatic template inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 999–1013, Santa Clara, CA, February 2020. USENIX Association.

[28] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–111, Lombard, IL, April 2013. USENIX Association.

[29] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked*

Systems Design and Implementation, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[30] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. *SIGCOMM Comput. Commun. Rev.*, 42(4):467–472, September 2012.

[31] Thomas King, Christoph Dietzel, Job Snijders, Gert Döring, and Greg Hankins. BLACKHOLE Community. RFC 7999, October 2016.

[32] TOM Krazit. Networking issues take down google cloud in parts of the u.s. and europe, youtube and snapchat also affected, 2019.

[33] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[34] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 499–512, Oakland, CA, May 2015. USENIX Association.

[36] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review*, 41(4):290–301, 2011.

[37] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *Proceedings of the 24th International Conference on Large Installation System Administration*, LISA'10, page 1–8, USA, 2010. USENIX Association.

[38] Networkworld. What was wrong with united's router?, 2015.

[39] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. Scaling network verification using symmetry and surgery. *SIGPLAN Not.*, 51(1):69–83, January 2016.

[40] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. Scaling network verification using symmetry and surgery. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 69–83. ACM, 2016.

[41] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 123–144. Springer, 1984.

[42] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[43] Steve Ragan. Bgp errors are to blame for monday's twitter outage, not ddos attacks, 2016.

[44] David A. Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 669–685, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[45] Yakov Rekhter, Susan Hares, and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.

[46] Ruchit Shrestha, Xiaolin Sun, and Aaron Gember-Jacobson. Localizing router configuration errors using unsatisfiable cores. 2020.

[47] Tom Strickx and Jeremy Hartmann. Cloudflare outage on june 21, 2022, Jul 2022.

[48] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. Campion: Debugging router configuration differences. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 748–761, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Tim Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. Kirigami, the verifiable art of network cutting, 2022.

[50] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. Modular control plane verification via temporal invariants, 2022.

[51] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 214–226, New York, NY, USA, 2019. Association for Computing Machinery.

[52] DYLAN TWENEY. 5-minute outage costs google $545,000 in revenue, 2013.

[53] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an smt solver. *SIGPLAN Not.*, 51(10):765–780, October 2016.

[54] Hongkun Yang and Simon S Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, 2015.

[55] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. Accuracy, scalability, coverage: A practical configuration verifier on a global wan. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 599–614. Association for Computing Machinery, 2020.

[56] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. Accuracy, scalability, coverage: A practical configuration verifier on a global wan. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 599–614, New York, NY, USA, 2020. Association for Computing Machinery.

[57] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and P. Mohapatra. Fireman: a toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy (S P'06)*, pages 15 pp.–213, 2006.

[58] Zen. https://github.com/microsoft/Zen.