Title
Understanding Border Gateway Protocol Configurations and Policies

Author
Bui, Thomas Binh-Huy

Publication Date
2018

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Understanding Border Gateway Protocol Configurations and Policies

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Computer Science

by

Thomas Binh-Huy Bui

2018

ABSTRACT OF THE THESIS

Understanding Border Gateway Protocol Configurations and Policies

by

Thomas Binh-Huy Bui
Master of Science in Computer Science
University of California, Los Angeles, 2018
Professor George Varghese, Chair

We present an object-oriented representation of routing policies to encode BGP-related configuration information. We demonstrate how router policies can be represented as pseudocode functions, making it easier to understand a router's configured policy. We discuss the C-representation of router policies that allow for the use of symbolic execution to explore all paths through a single router and produce test announcements. We generate router configurations compatible for Quagga to emulate the network slice, send test announcements, and observe the resulting routing table. We describe a method to guarantee equivalence given any two routers' BGP configurations with functional equivalence.

The thesis of Thomas Binh-Huy Bui is approved.

Yuval Tamir

Todd D. Millstein

George Varghese, Committee Chair

University of California, Los Angeles

2018

TABLE OF CONTENTS

1 Introduction **1**

2 Background and Motivation **4**

2.1 Networking Information 4

2.2 Example Configuration 5

2.2.1 Interfaces 5

2.2.2 OSPF 6

2.2.3 BGP 7

2.2.4 BGP IPv4 Address Family 8

2.2.5 Filter Lists 11

2.2.6 Static Routes 12

2.2.7 Route Maps 13

2.3 Network Emulation 14

2.4 Related Work 15

2.5 Design Overview 17

3 Internal Representation **19**

3.1 Classes 19

3.1.1 Session 19

3.1.2 Policy/Route Map 22

3.1.3 BGP Filters 27

3.1.4 Router 29

3.2 Python Dictionaries 32

4 IR Output **35**

4.1 Pseudocode Generation 35

4.2 Test Generation 37

4.3 Router Equivalence 40

4.4 Quagga Configuration 43

5 Results **45**

5.1 Pseudocode Example 45

5.2 Router Equivalence Results 48

5.3 Sending Test Announcements 50

6 Conclusion **54**

References **56**

LIST OF FIGURES

2.1 Router BGP Diagram 10

2.2 Design Model 18

3.1 BGP Peer Export Policy Example 21

3.2 Route-Map `as1_to_as3`: Statement 1 24

3.3 Route-Map `as1_to_as3`: Statement 2 25

3.4 Community list `as1_community` 29

3.5 Prefix list `as4-prefixes` 29

3.6 redistribute static route-map STATIC-TO-BGP 31

3.7 network 1.0.1.0 mask 255.255.255.0 32

3.8 ip route 169.232.90.0 255.255.255.0 169.232.110.7 32

5.1 Example topology from Batfish 51

5.2 Routing Information Base of as1border2 before route injection 52

5.3 ExaBGP working snippet 52

5.4 Routing Information Base of as1border2 after the first announcement 53

LIST OF TABLES

5.1 Router Equivalence Results 48

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor George Varghese for believing in me and granting this wonderful opportunity. He has challenged me to think creatively and taught the valuable nature of interdisciplinary thinking. I can't thank him enough for his suggestions to make this paper a complete project.

Thank you to committee members Todd Millstein and Yuval Tamir for their useful insight during our Network Verification Group meetings. Without their direction, we would have never been able to accomplish such a great amount of progress. They also provided constructive feedback on this thesis.

I would also like to thank fellow members of the Network Verification Group: Siva Kesava, Alan Tang, and Lana Ramjit. I couldn't have done it without all your ideas and code contributions. I will have lasting memories of eating Panda Express and looking for tables at the Bomb Shelter.

Finally, I would like to thank my family for their unwavering support.

CHAPTER 1

Introduction

Network configuration is both error-prone and difficult to debug [FB05] [MWA02]. Since configuration is decentralized across all devices in a network, problems may propagate across the network and manifest themselves in a device far from the source of the configuration error. Furthermore, many configuration errors are latent and only triggered under a certain set of conditions.

As the state of the network changes or policy is updated, errors can occur quite easily. These failures can be caused by problems with hardware, software, or configuration. As many companies shift to providing online services, problems with the network become more costly. In order to triage configuration errors, network operators must use ad hoc methods and tools like *traceroute* and *ping* to pinpoint an error in thousands of lines of configuration spread over multiple devices.

Router configuration is expressed in a fairly low-level configuration language, lacking high-level primitives. Combined with the flexibility of the Border Gateway Protocol (BGP) to implement different policy, this allows for misconfigurations to take place. Also, there is no unified router configuration language, so network operators may need to coordinate routers from different vendors, each with their own configuration language. The complexity of configuration languages can lead to mismatches between the router's behavior and the network operator's intent, resulting in faulty router configurations. Examples of routing faults include routing loops, blackholes, improper route filtering, route leakage, route instability, and route hijacking.

Recent work has attempted to provide operators with more sophisticated tools. For example, automated data plane testing finds the minimal set of test packets needed to check

desired properties in a network. Data planes are a result of a set of configurations and an environment [ZKV14]. Thus, data plane testing is reactive; in the face of changes to a configuration or an environment, it must wait for the produced data plane to manifest itself before it can detect an error, at which point the error is already live in the network. By contrast, control plane testing has the advantage of finding latent bugs in a configuration before they are activated as well as pinning errors to a specific set of announcements.

There have been existing attempts to verify correct behavior of the control plane from configurations. *rcc* reads from configurations and checks for common desirable properties [FB05]. Batfish verifies configuration by modeling the data planes produced from the configurations [FFP15]. While these projects propose tools to verify properties, they can only verify control plane properties for environments provided during verification. They cannot test the correct operation of router hardware and software. Although automatic test generation has been done in the world of software, it has not been done for network routing policy. There are no automated testing tools that can monitor routing behavior under arbitrary live environments.

This thesis makes several contributions. First, we present a mechanism to extract BGP-related information from router configurations and display the policies in a easy to understand, human-readable format. Second, we present a first attempt at automated testing for a routing protocol, BGP. We use an imperative language data model coupled with symbolic execution to generate a high coverage set of test announcements which can be injected into an emulated network to capture routing behavior. Third, we develop a mechanism to perform router equivalence checks between two routers, where we define routers to be equivalent if they have equivalent routing policies with respect to some neighbor mapping.

The rest of the thesis is structured as follows:

- Chapter 2 describes relevant background information, technologies used, and a walk through of an example Cisco IOS router configuration.
- Chapter 3 discusses the implementation of the internal representation used to express router configurations.

- Chapter 4 provides the outputs generated as a result of reading the configurations and constructing our internal representation.
- Chapter 5 presents use cases of our internal representation. These include the pseudocode representation of policies, the injection of control plane packets into a network, and router equivalence results for a campus network of a large university.
- Chapter 6 concludes.

CHAPTER 2

Background and Motivation

2.1 Networking Information

A network consists of routers and switches, and these devices are used to send packets from one location to another. These data packets must follow the rules of the network as defined by the network's data plane. The network data plane determines the forwarding of traffic according to a packet's header by following the rules of a forwarding table. These forwarding rules determine the outgoing port on which to send a packet with a particular header. The network control plane builds the data plane depending on the link failures in the topology, route announcements received from neighboring nodes, and individual node configurations.

Modifying these router configurations is the current method used by network operators to steer traffic. These decisions are typically financially motivated. For example, given two possible routes through two different Internet service providers (ISPs) to the wide area network (WAN), a network operator would configure routers to prefer a route to the ISP which charges a cheaper price given equivalent bandwidths. This manual configuration of routers is possible by modifying attributes of a routing protocol. An autonomous system (AS) denotes a group of routers under a common network operator. Some examples of autonomous systems include an ISP, a university or a large company.

Two examples of well-known routing protocols include Open Shortest Path First (OSPF) and Border Gateway Protocol (BGP). OSPF is a link state routing protocol which computes best route to a destination by shortest path cost. OSPF typically uses link speed to determine cost, but this value can be overwritten to manually configure link costs. OSPF is an example of an interior gateway protocol (IGP), which are used to exchange routing information within

an autonomous system.

BGP is the most commonly used exterior gateway routing protocol in networking today. It is often used to transmit routing information among autonomous systems. BGP's configuration is much more flexible than OSPF, having many modifiable attributes (e.g. local preference, AS_Path, MED, etc.) This added flexibility comes at the cost of increased complexity to select the best route.

For example, we examine the local preference and AS_Path attributes of two route announcements for the same prefix. The route announcement with the larger local preference will be installed in the forwarding table. If the local preferences of the two route announcements are equivalent, the announcement with the shorter AS_Path length will be preferred. Cisco describes their full BGP best path selection algorithm [bgp].

2.2 Example Configuration

In this section, we examine various sections of a router configuration to describe their effect on router behavior. Assume we are given a simple configuration written for Cisco IOS which has been taken from the Batfish repository [bat].

This is a relatively small example of a router configuration, whereas real router configurations often contain thousands of lines of esoteric directives only known by Cisco CLI experts. Also, one needs to consult the documentation to understand the semantics of commands based on their context as well as learn the default settings that these configurations employ. Since our focus is on the BGP route announcements within a network, some of these lines contain unnecessary information for our model of the network.

We have divided the example configuration into the following sections.

2.2.1 Interfaces

This first section declares a router's interfaces as well as enters into interface configuration mode to specify attributes for each interface. Some examples of interface configuration

include the assignment of an IP address and subnet, setting up duplex operation, or assigning a speed to an interface. This router sets up three Gigabit Ethernet interfaces (LAN interface) and one Loopback interface (logical, or virtual interface).

```
interface Loopback0
 ip address 1.2.2.2 255.255.255.255
interface GigabitEthernet0/0
 ip address 10.13.22.1 255.255.255.0
 media-type gbic
 speed 1000
 duplex full
 negotiation auto
interface GigabitEthernet1/0
 ip address 1.0.2.1 255.255.255.0
 negotiation auto
interface GigabitEthernet2/0
 ip address 10.14.22.1 255.255.255.0
 negotiation auto
```

2.2.2 OSPF

This section configures the router's OSPF process. The **1** in `router ospf 1` defines the unique process-id parameter for OSPF processes on a router. This router specifies a router-id 1.2.2.2, which uniquely identifies the router within an OSPF routing domain. The `network area` command defines the interfaces on which OSPF runs as well as the area ID for those interfaces. The **redistribute connected subnets** command allows for the redistribution of routes into OSPF, using the **connected** keyword to refer to routes that are automatically established when IP is enabled on an interface. Another example usage of the redistribute command is **redistribute static subnets** to redistribute a router's IP static routes into an OSPF routing domain [cisa].

```
router ospf 1
 router-id 1.2.2.2
 redistribute connected subnets
 network 1.0.0.0 0.255.255.255 area 1
```

2.2.3 BGP

This next section configures Border Gateway Protocol (BGP), allowing a router to exchange network reachability information with other BGP speakers. The first line **router bgp 1** specifies the AS number (1) of the router and enables a BGP routing process for the router. The **router-id** command configures a fixed router ID as an IP address. This router ID is used as a tie-breaker in the BGP path selection procedure, with preference for the lowest IP address.

This router creates BGP peer groups by using the **neighbor peer-group** command while in BGP configuration mode. Neighbors with the same BGP policies can be added to the same peer group to simplify configuration, so that these peer groups can be referenced in the address family mode of BGP configuration instead of repeating commands for each neighbor in a peer group. This router's configuration defines a peer group **as1** with AS number 1. A neighbor with IP address 1.10.1.1 will have all the policies of peer group **as1** applied to this neighbor.

The **neighbor update-source** command specifies to use the router's Loopback interface, instead of assigning the closest interface, as the source interface for the TCP connection to BGP neighbor 1.10.1.1. The Loopback interface is typically used as the source interface for iBGP sessions because the BGP session will stay up as long as IGP can determine a path to the neighboring router. This allows BGP sessions to be robust to physical interface failures if there are multiple paths to the neighbor. This router and neighboring router 1.10.1.1 are both in AS1, which is why this router updates the source interface to the Loopback interface for that particular BGP session. [cisb].

```
router bgp 1
```

```
bgp router-id 1.2.2.2
bgp log-neighbor-changes
neighbor as1 peer-group
neighbor as1 remote-as 1
neighbor as2 peer-group
neighbor as2 remote-as 2
neighbor as3 peer-group
neighbor as3 remote-as 3
neighbor as4 peer-group
neighbor as4 remote-as 4
neighbor 1.10.1.1 peer-group as1
neighbor 1.10.1.1 update-source Loopback0
neighbor 10.13.22.3 peer-group as3
neighbor 10.14.22.4 peer-group as4
```

2.2.4 BGP IPv4 Address Family

This section configures BGP routing sessions that use IP addresses in the IPv4 unicast address family. The **additional-paths** commands specify that this router can both send and receive additional paths and selects all possible candidate paths with unique next hops for advertisement. Typically, a router would only advertise the best path from its routing information base (RIB). There are three components that make up the RIB for a BGP speaker [RLH06]:

- Adj-RIBs-In: Stores routes from BGP neighbors' route announcement messages
- Loc-RIB: Applies local policies to Adj-RIBs-In, narrowing down received routes to the set of best routes that this router will use.
- Adj-RIBs-Out: Stores routes that will be advertised to this router's BGP neighbors

The **neighbor advertise additional-paths** command is necessary for the router to ad-

vertise routes from the set of additional paths to a specified neighbor or peer group. This particular router example advertises additional paths only to neighbors in peer group **as1** [cisd].

The **network** command specifies networks that will be advertised by this BGP routing process. This router advertises 1.0.1.0/24 and 1.0.2.0/24 subnets. The **neighbor send-community** command specifies that community attributes should be sent to the particular BGP neighbor, since the default setting does not send community attributes to any neighbors.

The **neighbor route-map { in | out }** commands are used to apply a route map to incoming or outgoing routes for the specified neighbor or peer group. A route map applied with keyword **in** to a router's neighbor will filter route announcements received from that BGP neighbor. More specifically, these route maps are part of the router's local policies which narrow the list of routes in Adj-Ribs-In to the subset of routes used by the router in Loc-RIB. Similarly, a route map applied with keyword **out** to a router's neighbor will filter routes as they move from Loc-RIB to the router's Adj-RIB-out. This part of the RIB typically consists of a a router's best routes unless the additional-paths feature was configured [RLH06].

The **neighbor activate** command enables the exchange of routing information with a BGP neighbor. This is enabled by default for IPv4, but is disabled for all other address families [cisa]. Therefore, the configuration directives in this example are redundant.

```
address-family ipv4
 bgp additional-paths select all
 bgp additional-paths send receive
 network 1.0.1.0 mask 255.255.255.0
 network 1.0.2.0 mask 255.255.255.0
 neighbor as1 send-community
 neighbor as1 advertise additional-paths all
 neighbor as2 send-community
 neighbor as2 route-map as2_to_as1 in
```

```
 neighbor as2 route-map as1_to_as2 out
 neighbor as3 send-community
 neighbor as3 route-map as3_to_as1 in
 neighbor as3 route-map as1_to_as3 out
 neighbor as4 route-map as4_to_as1 in
 neighbor as4 route-map as1_to_as4 out
 neighbor 1.10.1.1 activate
 neighbor 10.13.22.3 activate
 neighbor 10.14.22.4 activate
 maximum-paths 5
exit-address-family
```

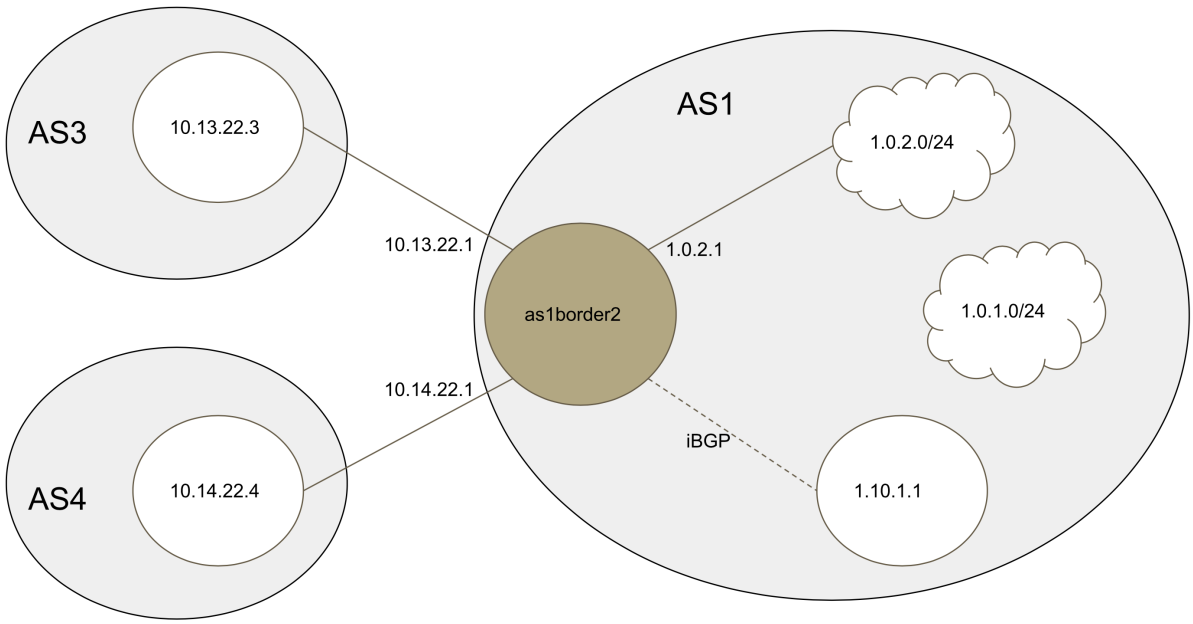

Figure 2.1: Router BGP Diagram

Figure 2.1 describes the BGP topology given router `as1border2`'s configuration, showing the physical interfaces used to peer with EBGP neighbors in AS3 and AS4. The dotted line represents the IBGP link connecting `as1border2` to 1.10.1.1, as this path is determined by an IGP such as OSPF. Also, since `as1border2` does not have a directly connected interface

to subnet 1.0.1.0/24, another router in this autonomous system may be connected to this subnet and will advertise the prefix to `as1border2` through OSPF. Then, `as1border2` can learn where to direct the traffic upon receiving data packets from EBGP neighbors.

2.2.5 Filter Lists

Upon returning to global configuration mode, this router configures various filter lists, and these lists will later be applied to route maps. The **ip community-list** command creates BGP community lists. A standard community list specifies well-known communities and community numbers, whereas an expanded community list filters communities using a regular expression. The new-format referenced in **ip bgp-community new-format** refers to the usage of two 2-byte numbers separated by a colon to view the autonomous system number and network number (AA:NN) as the community value string format; otherwise, the community value is displayed as its numerical, 32-bit value [cisa]. AS path filter lists uses the **ip as-path access-list** command to filter routes based on the AS path value matching a given regular expression, making it analogous to expanded community lists. Both community filters and AS path filters can have multiple statements per filter entry, creating a logical OR between the statements of each filter list. If a single statement has multiple values to match, a logical AND is created.

The **ip prefix-list** command adds statements to a prefix list entry. Each statement can permit or deny route announcements if they match the network/mask_length prefix as well as the optional constraints on the prefix length with **ge** and **le** [cisa]. The **access-list** command defines IP access lists. Extended ACLs have access list numbers within the range (100-199), therefore the access lists in this example are extended ACLs. One can view the extended ACL as an alternate implementaton of a prefix list when it is applied to a route map's match statement. For example, access-list 101 permits prefixes 1.0.1.0/24 and 1.0.2.0/24. The first IP address in the access-list corresponds to the prefix for the route to match with, and the second IP address corresponds to the mask length for the prefix match. The keyword **host** is a replacement for wildcard 0.0.0.0, which means to take all bits from the IP address.

Extended ACLs only match the exact prefix length, losing some flexiblity in comparison to prefix lists [cisc].

```
ip bgp-community new-format
ip community-list expanded as1_community permit _1:
ip community-list expanded as2_community permit _2:
ip community-list expanded as3_community permit _3:
ip community-list expanded as4_community permit _4:
ip as-path access-list 1 permit ^1$
ip prefix-list as4-prefixes seq 1 permit 4.0.0.0/8 le 32
ip prefix-list inbound_route_filter seq 5 deny 1.0.0.0/8 le 32
ip prefix-list inbound_route_filter seq 10 permit 0.0.0.0/0 le 32
access-list 101 permit ip host 1.0.1.0 host 255.255.255.0
access-list 101 permit ip host 1.0.2.0 host 255.255.255.0
access-list 102 permit ip host 2.0.0.0 host 255.0.0.0
access-list 102 permit ip host 2.128.0.0 host 255.255.0.0
access-list 103 permit ip host 3.0.1.0 host 255.255.255.0
```

2.2.6 Static Routes

In global configuration mode, the router also configures its static routes. The static route command `ip route` specifies the network prefix and the next hop IP address of the given route. The directive can include additional information, such as administrative cost, about the static route.

```
ip route 169.232.90.0 255.255.255.0 169.232.110.7
ip route 169.232.93.0 255.255.255.0 169.232.110.7
```

2.2.7 Route Maps

This final section defines the route maps which had been previously declared in the BGP configuration section. A route map is identified by a name, such as `as1_to_as2`, and consists of an ordered sequence of individual statements/clauses. Each route map clause has **match** commands and/or **set** commands. With a **permit** route map, a route which matches the conditions of the match statement(s) will then perform the actions of the set statement(s). The conditions of a match statement can match ACLs, prefix lists, community lists, an AS path, or a metric (MED) value. A set statement can update BGP attributes such as local preference, MED, a next hop address, a community attribute, or the AS path.

```
route-map as1_to_as2 permit 1
 match ip address 101
 set metric 50
 set community 1:2 additive
route-map as1_to_as2 permit 3
 match ip address 103
 set metric 50
 set community 1:2 additive
route-map as2_to_as1 permit 100
 match community as2_community
 set local-preference 350
route-map as1_to_as3 permit 1
 match ip address 101
 set metric 50
 set community 1:3 additive
route-map as1_to_as3 permit 2
 match ip address 102
 set metric 50
 set community 1:3 additive
```

```
route-map as3_to_as1 permit 100
 match community as3_community
 set local-preference 350
route-map as1_to_as4 permit 2
 set metric 50
 set community 1:4 additive
route-map as4_to_as1 permit 100
 match ip address prefix-list as4-prefixes
 match community as4_community
 set local-preference 350
```

2.3 Network Emulation

To emulate the router device such that it behaves as if it were in a real environment running the protocols, we use Mininet [LHM10] with the MiniNExT [SZC14] extension layer. Mininet can emulate a complete network of hosts, links, and switches on a single machine, but Mininet runs everything in the `root` process [sin]. Since we require isolation (separate process spaces with Linux containers) from other entities, we use **MiniNExT** (Mininet ExTended) [SZC14], an extension layer that makes it easier to build complex networks in Mininet. The only trade-off is that MiniNExT only supports Ubuntu 14 running Linux Kernel version 3.

To implement routing protocols, we use **Quagga** [Qua], a routing software suite, which provides implementations of OSPFv2, OSPFv3, RIP v1 and v2, RIPng and BGP-4. For each router, we have a configuration file which is used as an input to Quagga. The file provides configuration for the protcols supported by the router. Quagga configuration files typically specify the IP addresses assigned to the router's interfaces, parameters for routing protocols like OSPF and BGP, and their desired policies with its neighbors. The Quagga configuration language follows a syntax similar to the Cisco IOS syntax described in §2.2.

In order to test the routers, we need the ability to send arbitrary announcements. Since

Quagga works in the way specified by the configuration file alone and cannot generate arbitrary route announcements, we require a route-injector which can send announcements that we specify to a router being tested. **ExaBGP** [Man] facilitates route injection by plugging scripts into BGP. Those scripts can then receive and advertise routes. ExaBGP manages BGP while the scripts read routes from standard input or advertise them on standard output. Unlike Quagga, ExaBGP does not perform any FIB manipulation.

2.4 Related Work

Previous work has explored control plane verification and data plane testing in networks.

Tools such as rcc [FB05], Batfish [FFP15], ARC [GVA16], ERA [FSF16], and Minesweeper [BGM17] are control plane analysis and verification systems used to make claims about network properties such as reachability and consistency.

rcc (router configuration checker) uses static analysis to check configuration constraints in order to identify BGP configuration faults based on a high-level correctness specification. Their normalized representation for BGP configurations uses mySQL database. Unlike the other control plane analysis tools, rcc does not create a model of the network.

Batfish uses a variant of Datalog to encode a logical model of the control plane, representing the configuration information and control plane semantics as logical facts. From the logical model and a given environment (i.e. link status and route announcements from neighbors represented as logical facts), Batfish can analyze the resulting data plane. Since Batfish can only reason about a single data plane at a time and given the large number of potential environments, it is intractable to generate and analyze all data planes for a given control plane.

ARC develops an abstract representation for control planes, translating network configurations into a collection of weighted digraphs that represent the true forwarding behavior of the network. Reasoning about the control plane is accomplished by computing graph characteristics of ARC. However, ARC represents only a limited set of control plane protocols

and features.

ERA models a control plane message as a 128-bit vector to encode attributes of the route announcement, using binary decision diagrams (BDDs) to represent sets of route announcements to enable fast set operations. The control plane is modeled as a function which takes routes as inputs and produces routes as outputs. ERA is unable to verify configurations for all environments.

Bagpipe allows network operators to express BGP policies as declarative specifications. It represents configurations as a set of network traces. Using an SMT-based symbolic execution solver, Bagpipe performs an initial network reduction to search through a finite set of traces and verify the control plane policies for a single AS. Since Bagpipe focuses on BGP, it does not model any IGPs.

Minesweeper represents network configurations as a logical formula N. Network operators can also use logical formulas to express properties P. By using an SMT solver to search for a solution to $N \wedge \neg P$, Minesweeper can verify that these properties are not satisfiable under any possible environment if a solution exists. Since Minesweeper uses an SMT solver, only one solution is generated that violates the formula instead of all possible solutions.

Automatic Test Packet Generation (ATPG) [ZKV14] automatically generates tests for a given data plane. It is a framework to test the forwarding rules and links of a network. This tool attempts to help the network operator determine if the data plane matches their intended configuration specification. Similarly, our tool can help network operators determine if the control plane matches their intended configuration specification. To perform this control plane testing, we monitor the traversal and transformations of routing announcements instead of the data plane test packets used in ATPG.

OpenConfig [ope] is a working group focused on compiling vendor-neutral data models, using YANG as the data modeling language. The OpenConfig data model includes much of the same configuration information as the Batfish data model. One eventual goal is for aspects of the OpenConfig data models to be standardized. We did not find a way to generate OpenConfig data models when given router configurations. Since OpenConfig's data models

have not been standardized yet, we based the implementation of our IR on the Batfish data model.

2.5 Design Overview

In order to model a router's BGP process, we identified the relevant information from a router configuration. At a high-level, this information consists of the import and export policies of a router and the network topology.

The import and export policies of a router are the route announcement filters, which are referred to as `route-map` in Cisco IOS. These filters modify route attributes as routes are installed in the RIB and control the propagation and transformation of BGP routing announcements through the network. A router can have different policies for each of its BGP neighbors. The network topology must identify all the nodes and edges in the network graph. Given a set of router configurations, we needed a mechanism to extract this network information.

Router configurations can come in a variety of languages depending on the router's vendor. For example, configurations written for Cisco IOS have a different syntax to configurations written for Juniper Junos OS. It would be tedious work to write a parser for each vendor's configuration language. Instead, we leverage the existing work from Batfish to extract a vendor neutral data model of the network [FFP15]. Using this data model allows our testing framework to accept any configuration language once we develop a data extractor for Batfish's JSON representation of the network data model.

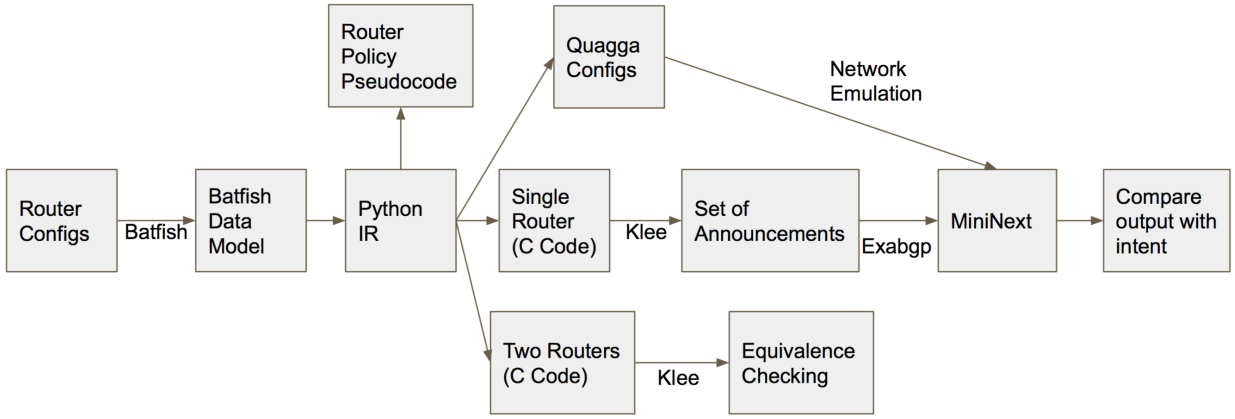

Figure 2.2: Design Model

We have four desired outputs from the Batfish data model. First, we express the import and export policies as human-readable pseudocode. Second, we want to represent these same policies as functions written in C to be used input for our test generation mechanism. Third, we want to generate C code to perform router equivalence checks among pairs of routers. Fourth, we need to generate configurations in a language suitable for the Quagga routing software suite since Quagga will implement the routing processes for our emulated network. To help us generate these outputs, we develop an intermediate representation of BGP configuration with Python classes, dictionaries, and lists. An overview of our design model is depicted in Figure 2.2.

CHAPTER 3

Internal Representation

To implement the data extractor, we identify relevant information from the data model that will be necessary to produce the four outputs. Then, we store this information into data structures (using Python classes, dictionaries, and lists), which we consider to be our internal representation (IR) of router's BGP policies. Our IR allows for convenient lookups to generate the configuration files and C code, separating the data gathering step from the analysis step. This chapter describes the IR in detail.

3.1 Classes

We describe the various Python classes which were created to organize our IR into an object-oriented design. The following Python code describes the initializers for each object type.

3.1.1 Session

We define a Python class object named `Session` to represent a BGP session between a router (specified by its hostname `router_name`) with one of its neighbors (specified by the neighbor's IP address `neighbor`). This neighbor is part of AS number `as_number`. The initializer for `Session` accepts the names of the import/export policies rather than the policy objects themselves. This implementation decision was due to the indirect references used in the Batfish data model. We leave `import_policy` and `export_policy` variables to be populated in the future once we gather information about these policies. We discuss how policies are represented in §3.1.2.

```
class Session:
```

```
def __init__(self, router_name, neighbor_ip, as_number, \
             import_policy, export_policy):
    self.router_name = router_name
    self.neighbor = neighbor_ip
    self.as_number = as_number
    self.import_policy = None
    self.import_policy_name = import_policy
    self.export_policy = None
    self.export_policy_name = export_policy
    self.interface = None
```

There exists a common structure for the export policy of each BGP neighbor. For example, suppose we are given a router whose neighbor has IP address 10.13.22.3. The Batfish data model will have a key "exportPolicy" within this neighbor's BGP information associated with value ∼*BGP_PEER_EXPORT_POLICY:default:10.13.22.3*∼. When looking up this export policy name in the "routingPolicies" section of the data model, we find a list of statements. One statement will have a key "class" with value "org.batfish.datamodel.routing_policy.statement.If". This statement object will also have key "guard" with two possible class types:

- org.batfish.datamodel.routing_policy.expr.Conjunction
- org.batfish.datamodel.routing_policy.expr.CallExpr

```
"~BGP_PEER_EXPORT_POLICY:default:10.13.22.3~" : {
  "name" : "~BGP_PEER_EXPORT_POLICY:default:10.13.22.3~",
  "statements" : [
    {
      "class" : "org.batfish.datamodel.routing_policy.statement.If",
      "comment" : "peer-export policy main conditional: exitAccept if true / exitReject if false",
      "falseStatements" : [
        {
          "class" : "org.batfish.datamodel.routing_policy.statement.Statements$StaticStatement",
          "type" : "ExitReject"
        }
      ],
      "guard" : {
        "class" : "org.batfish.datamodel.routing_policy.expr.Conjunction",
        "conjuncts" : [
          {
            "class" : "org.batfish.datamodel.routing_policy.expr.CallExpr",
            "calledPolicyName" : "~BGP_COMMON_EXPORT_POLICY:default~"
          },
          {
            "class" : "org.batfish.datamodel.routing_policy.expr.CallExpr",
            "calledPolicyName" : "as1_to_as3"
          }
        ]
      },
      "trueStatements" : [
        {
          "class" : "org.batfish.datamodel.routing_policy.statement.Statements$StaticStatement",
          "type" : "ExitAccept"
        }
      ]
    }
  ]
},
```


Figure 3.1: BGP Peer Export Policy Example

If the "guard" is of type "CallExpr", this BGP neighbor will only use the export policy named *~BGP_COMMON_EXPORT_POLICY:default~*. This refers to a JSON object in this router's "routingPolicies" which describes the router's common export policy. This common export policy is based on the router's configuration, and the effects of this policy are shared by all the router's BGP neighbors. For instance, it includes the list of IP prefixes being advertised by this router. It can also include the list of networks for which a summary-address is created in the routing table using the **aggregate-address** command. Furthermore, it includes information about redistribution of connected routes or redistribution of static routes into BGP as well as the route maps applied which may be applied to the route redistribution.

If the "guard" is of type "Conjunction", there will be a key "conjuncts" referencing a list of policies, and this list typically has two policies. One of these policies is the common export policy. The second policy is the name of the outgoing route map which is applied to

this BGP neighbor in the router configuration. For example, Figure 3.1 shows a router with an export policy filter named `as1_to_as3` applied to neighbor 10.13.22.3.

3.1.2 Policy/Route Map

A routing policy consists of a list of match-action clauses, where an action (e.g. setting an attribute value) occurs if an attribute of a route announcement matches a certain predicate. These predicates include satisfying a prefix match, matching a regular expression of a community list, and matching a regular expression of an AS path list. Since a routing policy can have multiple clauses, we represent a `Session`'s import and export policy as a list of `Route_Map_Clause`.

For each `Route_Map_Clause` object, we keep track of the name of the route map along with the sequence number of the clause. Each `Route_Map_Clause` has variables to keep track of the three possible match statement types. The **community_list**, **as_path_list**, and **route_filter_list** instance variables each refer to a list populated by their respective BGP filter objects (to be discussed in §3.1.3). If a `Route_Map_Clause` only has a match statement on a community list, self.route_filter_list and self.as_path_list will both be empty lists. Furthermore, instance variable **actions** contains a list of object `Action` since a route map clause can contain multiple set statements. Each `Action` has a field (e.g. MED, local preference, community tag, etc.) that will be updated to **value**.

```
class Route_Map_Clause:
    def __init__(self, name, seq="", permit=True, cl=[], rfl=[], \
            aspl=[]):
        self.name = name # name
        self.seq = seq # empty string "" as placeholder
        self.permit = permit
        self.community_list = cl
        self.route_filter_list = rfl
        self.as_path_list = aspl
```

```
        self.actions = [] # List of possible Actions

    class Action:
        def __init__(self, field=None, value=None, additive=False):
            self.field = field
            self.value = value
            self.additive = additive # Bool for community tags
```

Figure 3.2 depicts the Batfish representation for a single route map clause; more specifically, this statement is the first clause of route map `as1_to_as3`. Together with Figure 3.3, which represents the second clause, we have all the information about policy `as1_to_as3`. If we tokenize the comment string by character '∼', the last character contains the sequence number of this clause.

```
"as1_to_as3" : {
  "name" : "as1_to_as3",
  "statements" : [
    {
      "class" : "org.batfish.datamodel.routing_policy.statement.If",
      "comment" : "~RMCLAUSE~as1_to_as3~1~",
      "guard" : {
        "class" : "org.batfish.datamodel.routing_policy.expr.MatchPrefixSet",
        "prefix" : {
          "class" : "org.batfish.datamodel.routing_policy.expr.DestinationNetwork"
        },
        "prefixSet" : {
          "class" : "org.batfish.datamodel.routing_policy.expr.NamedPrefixSet",
          "name" : "101"
        }
      },
      "trueStatements" : [
        {
          "class" : "org.batfish.datamodel.routing_policy.statement.SetMetric",
          "metric" : {
            "class" : "org.batfish.datamodel.routing_policy.expr.LiteralLong",
            "value" : 50
          }
        },
        {
          "class" : "org.batfish.datamodel.routing_policy.statement.AddCommunity",
          "expr" : {
            "class" : "org.batfish.datamodel.routing_policy.expr.InlineCommunitySet",
            "communities" : [
              {
                "prefix" : {
                  "class" : "org.batfish.datamodel.routing_policy.expr.LiteralCommunitySetElemHalf",
                  "value" : 1
                },
                "suffix" : {
                  "class" : "org.batfish.datamodel.routing_policy.expr.LiteralCommunitySetElemHalf",
                  "value" : 3
                }
              }
            ]
          }
        },
        {
          "class" : "org.batfish.datamodel.routing_policy.statement.Statements$StaticStatement",
          "type" : "ReturnTrue"
        }
      ],
      "falseStatements" : [
```


Figure 3.2: Route-Map `as1_to_as3`: Statement 1

In Figure 3.2, ∼RMCLAUSE∼as1_to_as3∼1∼ represents the statement with sequence number 1 in route map `as1_to_as3`. From the "guard," we know that this clause performs the actions of "trueStatements" if the route announcement matches the prefix filter list `101`. The actions of this trueStatement include setting the MED to "50" and updating the community tag to add "1:3" for routes matching extended ACL `101`. Furthermore, this clause has key "falseStatements" to reference any additional clauses from the route map. Figure 3.3 depicts the second clause of `as1_to_as3` having sequence number 2. This dictionary object

follows a similar structure containing "guards," "trueStatements," and "falseStatements" keys. The second clauses similarly sets MED to "50" and updates community to add "1:3" for routes matching extended ACL `102`.


```
  "falseStatements" : [
    {
      "class" : "org.batfish.datamodel.routing_policy.statement.Statements$StaticStatement",
      "type" : "ReturnLocalDefaultAction"
    }
  ],
  "guard" : {
    "class" : "org.batfish.datamodel.routing_policy.expr.MatchPrefixSet",
    "prefix" : {
      "class" : "org.batfish.datamodel.routing_policy.expr.DestinationNetwork"
    },
    "prefixSet" : {
      "class" : "org.batfish.datamodel.routing_policy.expr.NamedPrefixSet",
      "name" : "102"
    }
  },
  "trueStatements" : [
    {
      "class" : "org.batfish.datamodel.routing_policy.statement.SetMetric",
      "metric" : {
        "class" : "org.batfish.datamodel.routing_policy.expr.LiteralLong",
        "value" : 50
      }
    },
    {
      "class" : "org.batfish.datamodel.routing_policy.statement.AddCommunity",
      "expr" : {
        "class" : "org.batfish.datamodel.routing_policy.expr.InlineCommunitySet",
        "communities" : [
          {
            "prefix" : {
              "class" : "org.batfish.datamodel.routing_policy.expr.LiteralCommunitySetElemHalf",
              "value" : 1
            },
            "suffix" : {
              "class" : "org.batfish.datamodel.routing_policy.expr.LiteralCommunitySetElemHalf",
              "value" : 3
            }
          }
        ]
      }
    },
    {
      "class" : "org.batfish.datamodel.routing_policy.statement.Statements$StaticStatement",
      "type" : "ReturnTrue"
    }
  ]
}
```


Figure 3.3: Route-Map `as1_to_as3`: Statement 2

With the Python IR, we represent route map `as1_to_as3` as a list of two Route_Map_Clause objects [RMC1, RMC2]. The two clauses have the following attributes.

RMC1:

```
    name =  as1_to_as3
    seq = 1
    permit = True
    community_list = None
    route_filter_list = None
    as_path_list = 101
    actions = [Action1, Action2]

RMC2:
    name =  as1_to_as3
    seq = 2
    permit = True
    community_list = None
    route_filter_list = None
    as_path_list = 102
    actions = [Action3, Action4]

Action1:
    field = metric
    value = 50
    additive = False
Action2:
    field = community
    value = "1:3"
    additive = True
Action3:
    field = metric
    value = 50
    additive = False
```

```
Action4:
    field = Community
    value = "1:3"
    additive = True
```

3.1.3 BGP Filters

As discussed in §2.2.4, a route announcement can be filtered based on a router's policies. A BGP route filter can permit or deny routes based on the prefix being announced. For example, **match ip address prefix-list as4-prefixes** or **match ip address 101** specify that their corresponding route maps will filter based on the prefixes of the announced networks in prefix list `as4-prefixes` and ACL 101, respectively. Additionally, a route announcement can be filtered based on other route attributes. For instance, **match community as4_community** will compare a route's community tag value to the regular expression defined by community list `as4_community` using the command **ip community-list expanded as4_community**. Similarly, **match as-path 10** will match a route announcement with an AS path matching the regular expression defined by AS path list `10` with the command **ip as-path access-list 10**.

BGP filters can have multiple clauses associated with a single BGP filter entry. The Batfish data model implements this by mapping key "lines" to a list of JSON objects where each object represents a unique clause. So, for each router in a network, we maintain Python lists to represent the following filter types: community lists, AS path list, and route filter lists.

This required defining class objects for the three match statement types. Community lists and AS path lists are defined by a name, a matching regular expression, and a Boolean for permit/deny. The `Route_Filter_List` class consists of a name, a permit/deny Boolean, a sequence number (to determine priority of statements), an IP address prefix, and a prefix length range. Since a list corresponds to a BGP filter entry, each class instance represents a single clause of a filter entry.

```
class Route_Filter_List:
    def __init__(self, name, permit, prefix, mask_lower, \
            mask_upper, seq):
        self.name = name
        self.permit = permit
        self.prefix = prefix
        self.mask_lower = mask_lower
        self.mask_upper = mask_upper
        self.seq = seq

class Community_List:
    def __init__(self, name, permit, regex):
        self.name = name
        self.permit = permit
        self.regex = regex

class AS_Path_List:
    def __init__(self, name, permit, regex):
        self.name = name
        self.permit = permit
        self.regex = regex
```

Figure 3.4 depicts how each router has a dictionary of "communityLists" entries. For instance, community list `as1_community` accepts routes with a community tag matching the regular expression "_1:".

Figure 3.4: Community list `as1_community`

Figure 3.5 shows how the Batfish data model represents an IPv4 route filter list, which is referenced by key "routeFilterLists." These route filter lists can be implemented as prefix lists or ACLs that are applied to a route map. So, prefix list `as4-prefixes` is a route filter list since it is applied to route map `as4_to_as1`, and it permits route announcements for networks matching `4.0.0.0/8` with a subnet mask length between 8 and 32.

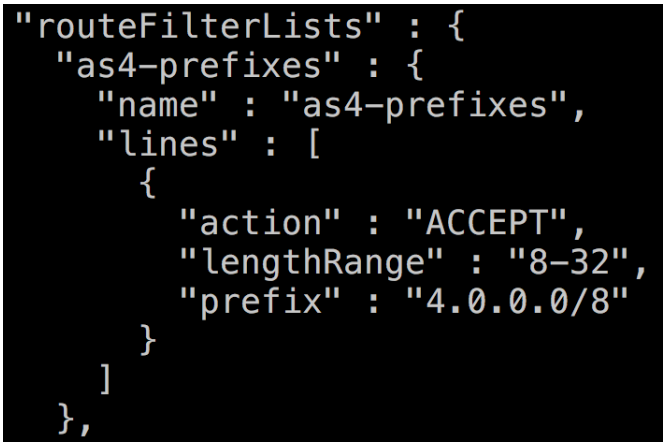

Figure 3.5: Prefix list `as4-prefixes`

3.1.4 Router

We created `class Router` to store router-specific attributes that affect the control plane policy. Similar to how routes can be redistributed into OSPF as described in §**??**, route redistribution allows routes from one routing domain to be redistributed into BGP. This

affects BGP policy by adding to the set of route announcements that may be propagated by a router. We include various route redistribution properties as instance variables in `Router` to model its affect on a router's policy.

A router with the **redistribute connected** command in its BGP configuration process will lead to setting `Router` instance variable **connected** to True. Similarly, a router with the **redistribute static** command in its BGP configuration process will lead to setting `Router` instance variable **static** to True. Also, we have instance variable `networks` to keep a list of subnet prefixes being announced. We have instance variable `static_routes` to hold a list of Route objects representing static routes used by this router. We define `class Route` to store the destination network prefix, the IP address of the next-hop, and the administrative distance of a route.

```
class Router:
    def __init__(self, name):
        self.name = name
        self.static = False # Bool redistribute static route
        self.connected = False # Bool redistribute connected route
        self.static_name = None # Name of applied static RM
        self.static_policy = None # Static route map policy
        self.connected_name = None # Name of applied connected RM
        self.connected_policy = None # Connected route map policy
        self.networks = [] # Advertised subnets
        self.static_routes = [] # Static routes
class Route:
    def __init__(self, network, next_hop, cost):
        self.network = network
        self.next_hop = next_hop
        self.cost = cost
```

If applicable, the Batfish data model embeds route redistribution information in the

~BGP_COMMON_EXPORT_POLICY:default~ of a router. Figure 3.6 gives an example of a router with route map `STATIC-TO-BGP` applied to the redistribution of static routes into BGP. This requires setting `self.static` = True and `static_name` = `STATIC-TO-BGP`. We will later update `static_policy` to the contents of this policy, referencing the list of this policy's `Route_Map_Clause`.


```
"~BGP_COMMON_EXPORT_POLICY:default~" : {
  "name" : "~BGP_COMMON_EXPORT_POLICY:default~",
  "statements" : [
    {
      "class" : "org.batfish.datamodel.routing_policy.statement.If",
      "guard" : {
        "class" : "org.batfish.datamodel.routing_policy.expr.Disjunction",
        "disjuncts" : [
          {
            "class" : "org.batfish.datamodel.routing_policy.expr.Conjunction",
            "comment" : "Redistribute static routes into BGP",
            "conjuncts" : [
              {
                "class" : "org.batfish.datamodel.routing_policy.expr.MatchProtocol",
                "protocol" : "static"
              },
              {
                "class" : "org.batfish.datamodel.routing_policy.expr.WithEnvironmentExpr",
                "expr" : {
                  "class" : "org.batfish.datamodel.routing_policy.expr.CallExpr",
                  "calledPolicyName" : "STATIC-TO-BGP"
                },
```


Figure 3.6: redistribute static route-map STATIC-TO-BGP

Figure 3.7 show how *~BGP_COMMON_EXPORT_POLICY:default~* can include the network prefixes (e.g. 1.0.1.0/24) to be advertised. We append these prefixes to instance variable `networks`.

```
"~BGP_COMMON_EXPORT_POLICY:default~" : {
  "name" : "~BGP_COMMON_EXPORT_POLICY:default~",
  "statements" : [
    {
      "class" : "org.batfish.datamodel.routing_policy.statement.If",
      "guard" : {
        "class" : "org.batfish.datamodel.routing_policy.expr.Disjunction",
        "disjuncts" : [
          {
            "class" : "org.batfish.datamodel.routing_policy.expr.Conjunction",
            "conjuncts" : [
              {
                "class" : "org.batfish.datamodel.routing_policy.expr.MatchPrefixSet",
                "prefix" : {
                  "class" : "org.batfish.datamodel.routing_policy.expr.DestinationNetwork"
                },
                "prefixSet" : {
                  "class" : "org.batfish.datamodel.routing_policy.expr.ExplicitPrefixSet",
                  "prefixSpace" : [
                    "1.0.1.0/24"
                  ]
                }
              },
```


Figure 3.7: network 1.0.1.0 mask 255.255.255.0

Figure 3.8 shows how Batfish data model represents a router's static routes as a list of dictionary objects where each entry represents a unique static route. For each element in the list, we create a `Route` object and append to `static_routes`.


```
"staticRoutes" : [
  {
    "class" : "org.batfish.datamodel.StaticRoute",
    "administrativeCost" : 1,
    "network" : "169.232.90.0/24",
    "nextHopInterface" : "dynamic",
    "nextHopIp" : "169.232.110.7",
    "tag" : -1
  }
```


Figure 3.8: ip route 169.232.90.0 255.255.255.0 169.232.110.7

3.2 Python Dictionaries

So far, we have defined classes to represent the various elements of a router's configuration which we deemed relevant towards modeling the propagation of BGP routing announcements. We have numerous Python dictionaries where the key is a router name and the value is a dictionary or list containing objects created by the defined classes. These dictionaries allow for convenient look ups in order to perform code synthesis and configuration generation.

We discuss in §3.1.1 how Batfish's representation of a BGP neighbor's export policy requires following the indirection of routing policy $\sim BGP\_PEER\_EXPORT\_POLICY\sim$ to determine if an outgoing route map should be applied to this neighbor. We have dictionary `policy_dict` to map a router name to a dictionary object mapping these export policies to the applied route maps. It is a data structure used to populate the `export_policy` of `Session` objects.

In §3.1.3, we define the three object types used to represent route map match statements. We also define three BGP filter dictionaries to store the information of each BGP filter entry type. First, the `community_list_dict` dictionary maps a router name to a dictionary mapping community list names to community list entries. Each community list entry is represented by a Python list of `Community_List` objects. Second, the `as_path_list_dict` dictionary maps a router name to a dictionary mapping AS path list names to AS path filter entries. Each AS path filter entry is represented by a list of `AS_Path_List` objects. Third, the `route_filter_list_dict` dictionary maps a router name to a dictionary mapping route filter list names to route filter list entries. Each route filter list entry is represented by a list of `Route_Filter_List` objects.

The `route_map_dict` dictionary maps a router name to a dictionary mapping route map names to route map entries. Each route map entry contains a list of `Route_Map_Clause` objects. Since we create BGP filter objects to populate the BGP filter dictionaries, we can update the instance variables `cl`, `rfl`, and `aspl` of `Route_Map_Clause` objects to reference the same lists that populate the BGP filter dictionaries.

The `session_list_dict` dictionary maps a router name to a list of `Session` objects, where each `Session` object corresponds to a single neighbor of a given router. We use `policy_dict` to update the `Session` object `export_policy_name` appropriately. Once we have the actual policy names, we update `import_policy` and `export_policy` of the `Session` object to reference the route maps in `route_map_dict`.

The `prefix_to_intf_dict` dictionary maps a router name to a dictionary object, where each indexed dictionary contains mappings between a network prefix and the associated

interface name on the specified router. This data structure is primarily used to infer the network topology, pairing interfaces where their corresponding interfaces have long prefix matches. The number of matching bits should be at least as long as their network prefix length.

The `router_dict` dictionary allows us to access the remaining components of a router's BGP configuration attributes by mapping a router name to a `Router` object. As we discover additional configuration directives that affect the BGP routing process, we can add these attributes to the `Router` class definition. If new BGP filters are used, we need to construct a new class and use a dictionary to hold a list of these objects as well as update the `Route_Map_Clause` class definition to have a new instance variable for this BGP filter type.

After populating our Python dictionaries, is it much easier to generate the four outputs enumerated in §2.5 in a systematic fashion. One of the benefits of the object-oriented IR is the convenience of accessing a policy with a neighbor. If we were to interpret a route map's match conditions and set statements directly from the Batfish data model, we would first need to index the "routingPolicies" section. Then, depending on the match statements of the route map, we may need to look up the following sections: "communityLists," "routeFilterLists," and "asPathAccessLists." Our `Session` objects have accumulated all this information to easily get a router's policy with any of its neighbors. Also, this extraction of route map information from the data model only needs to be performed once instead of once per output created. We use Python due to our familiarity with the language and its object-oriented features. The built-in dictionaries and lists have been suitable for our current needs to analyze small networks. If there is a future need to perform queries on the network configuration, we may want to encode the information as database tables in the future.

CHAPTER 4

IR Output

This chapter describes the outputs we generate as a result of reading a router's Batfish data model and extracting the Python IR described in Chapter 3. §4.1 explains our interpretation of BGP policies and describes how we will create the pseudocode to express these policies. §4.2 discusses how test route announcements are generated from BGP policies. §4.3 describes how we perform router equivalence checks using Klee. §4.4 details the process of writing router configurations for the Quagga software suite to be used for network emulation.

4.1 Pseudocode Generation

As we have discussed earlier, it is difficult to understand a router's intended policy when only given the router's configuration file due to the complexity of router configuration languages. Leveraging our Python IR representation of BGP router policies, we attempted to simplify the readability of a policy. At a high level, routing policies can permit or drop route announcements as well as modify certain attributes of the installed route (for inbound route maps) or outgoing route announcement (for outbound route maps).

One can view a router's import policy as a function *import_policy*() which takes as input a set of route announcement A and produces as output an updated RIB containing the set of received routes filtered by the import policy B. Then, following BGP's best path algorithm and applying the router's local policies with function *local_policy*() , the set of received routes is reduced to the set of selected routes C. Finally, export policies *export_policy*() are applied to the set of selected routes. Similarly, these export policies can be viewed as functions taking a set of route announcements as input and producing the set of announcements that

will be propagated to certain neighbors by this router D. Given these variables for each set of route announcements (A, B, C, D) and these function declarations, we define a router's policy and behavior as follows:

```
def router_policy(A):
    import_policy(A) = B
    local_policy(B) = C
    export_policy(C) = D
    return D
```

The function *router_policy*() abstracts away the details of a router's import policy, local policy, and export policy. Since *router_policy*() returns a set of announcements, we can perform function composition across routers in the network. For example, we are given a network with two routers $R1$ and $R2$. We also have functions *router_policy_*$R1$ and *router_policy_*$R2$ to represent the respective routing policies of each router and variable x to represent the set of all possible route announcements A. To get the set of announcements y that are first filtered by $R1$ before being exported by $R2$, we have:

```
y = router_policy_R2( router_policy_R1( x ) )
```

By defining the functions called in *router_policy*(), we can expand on the details of the router's policy. We use pseudocode (similar syntax to C) to represent the *import_policy*() and *export_policy*() for the neighbors of a given router. As *local_policy*() for all routers follow the standard BGP best path algorithm, we currently do not produce pseudocode to model this function. We provide an example of the generated pseudocode in §5.1.

In order to generate the pseudocode for a single router, we iterate through the router's `session_list_dict` dictionary entry to get the `Session` objects for each BGP neighbor. Within the `Session` object, we can access the import policy and export policy that map to a function. A policy consists of a list of route map clauses, and each clause represents a new if statement in the function. We check if a prefix list, a community list, or an AS path filter list exists for each clause to write the condition of the route map clause, which is represented by a conjunction of the three BGP filters. Otherwise, this clause has no match statement,

so we set the condition to `True`. The body of the if-statement contains the actions from the clause. We provide some examples of these function definitions in §5.1.

4.2 Test Generation

The goal of automated testing is to generate a minimal set of tests that achieve a desired level of coverage. Some examples, in increasing granularity, include *all routers*, checking that announcements can be sent and received between neighbors, *all links* which checks all pairs of interfaces, and *all rules*, checking each active filter in a configuration. Currently, we focus on *all rules* testing for a single router.

This goal is similar to the aim of generating high path coverage in software testing via symbolic execution [CS13]. Our approach is to take advantage of existing program testing frameworks by translating BGP configuration into self-contained C programs. Starting from the object-oriented IR, we translate each router configuration into a program that uses functions to represent each filter.

The functions in these C programs are analogous to the functions described in §4.1, as they both reflect router policy. Therefore, generating the C functions used for test generation is similar to pseudocode generation. They both require iterating through a router's `session_list_dict` dictionary entry to get the policies for a given router's BGP neighbors.

However, the C functions are less readable than the pseudocode because these functions must conform to a real programming language and because we optimize the code to operate on integers whenever strings are used. For example, if we are given route map `as4_to_as1`, we can parse from the resulting pseudocode that the community tag value of an announcement must match "4:" and the prefix must match "4.0.0.0/8" and have a prefix mask length between 8 and 32. We convert the community tag (2 16-bit values separated by ':') and IP address (4 8-bit values separated by '.') strings into unsigned 32-bit integers. Then, we use the following boolean helper functions to replace operator '==' in the conditional statements of the pseudocode.

```
int ann_match_prefix(Announcement ann, uint32_t prefix,
    int32_t mask, int32_t ge, int32_t le);
int ann_match_community(Announcement ann, uint32_t match);
```

Route map `as4_to_as1` would be represented by the following C functions where community "4:" is represented by integer 262144 and prefix "4.0.0.0/8" is represented by integer 67108864:

```
Announcement as4_to_as1_168695300(Announcement a){
    if ( (ann_match_community(a, 262144)) &
            (ann_match_prefix(a, 67108864, 8, 8, 32)) ) {
        a.local_pref = 350;
    }
     else {
        a.is_dropped = 1;
    }
    return a;
}
```

We define a C struct that represents an announcement, where each announcement consists of a set of integer fields representing possible attributes of the announcement (e.g. prefix, community list, AS path list, local preference). Also, we add another Boolean variable to the struct, `is_dropped`, to account for the implicit drop. Starting from the object-oriented IR, we treat each match-action clause as a function that accepts an announcement and returns an announcement. We order match-action clauses by priority, where the clause with lower sequence numbers have higher priority. Then, we transform the clauses into `if-else` statements that return the possibly modified announcement. The following code segment shows a sample C-representation of a simple configuration for a router `r1`.

```
// Import policies
Announcement r3_to_r1(Announcement a){
    if (a.metric == 50){
```

```
        a.metric = 30;
    }
    else {
        a.is_dropped = 1;
    }
    return a;
}
//Export policies
Announcement r1_to_r2(Announcement a){
    if (a.metric == 30){
        a.local_pref = 100;
    }
    else {
        a.is_dropped = 1;
    }
    return a;
}
```

Because each function both accepts and returns a single announcement, we can compose the functions to represent the path an announcement takes through a router or network. By enumerating all interface-interface paths through a router, we can easily push a symbolic announcement through each path. We use Klee [CDE08] to generate fields for announcements that traverse a new path by either satisfying or failing a different set of failures. The following segment of code defines a `main()` function that tests the only path through router `r1`, receiving routes from `r3` and sending announcements to `r2`. Klee finds two path constraints: `a.metric == 50`, `a.metric != 50`.

```
int main(int argv, char **argv){
    Announcement a1;
    int metric;
```

```
    klee_make_symbolic(&metric, sizeof(metric), "metric");
    a1.metric = metric;
    a1.is_dropped = 0;
    //import from r3 and export to r2
    Announcement r = r3_to_r1(a1);
        if (!r.is_dropped){
            r1_to_r2(r);
        }
}
```

In this simple example, only the metric field is marked as symbolic. In practice, since we don't know which fields of the announcement the filter will branch on, all fields of the announcement are marked as symbolic. Since filters are per interface, we also generate a mapping that tags each generated test announcement with which interface it tests (and consequently, to which interface it should be sent). Klee uses an SMT solver to generate concrete instances of the constraints guarding each path. We harvest these concrete instances and translate them back into BGP announcements, tagged with which interface it tests which can then be sent into the network emulator.

4.3 Router Equivalence

Many networks may have pairs of routers with similar configurations if they have similar roles in the network [BGM17]. We define two routers to be functionally equivalent if we find some mapping of their neighbors where the routers have equivalent inbound and outbound policies with respect to the neighbor mapping and if the route maps applied to route redistribution are functionally equivalent. Once we get the neighbor mapping, we can compare the import and export policies attributed to the `Sessions` for all neighbor pairings.

If we find that two routers are functionally equivalent, we guarantee that the routers will have equivalent behavior in the network. However, the converse may not always be true; two

routers can have equivalent behavior while being functionally nonequivalent. For example, two routers with differing output filters but input filters that drop all announcements from all neighbors would have equivalent behavior without having functional equivalence. Although functional equivalence is a stronger guarantee than we need, it is a good first step to check for equivalence. It is efficient since it only requires linear number of checks with respect to number of neighbors. We could have checked for pathwise equivalence by comparing all interface-interface paths through a router. This would eliminate some false negatives from the functional equivalence check, but is much less efficient as it requires an exponential number of tests.

The process to generate router equivalence checking code is very similar to that of the test generation code. The route maps are represented as C functions as shown in §4.2. We append '1' or '2' to the function names in case the routers use identical route map names, which is often the case when two routers are configured to behave identically. The difference in the router equivalence code is in `main()`. Below, we provide an example of generated code to determine equivalence between two routers with route map `r3_to_r1` as defined above.

```
int main(int argv, char **argv){
    Announcement ann1, ann2;
    int m1, m2;
    klee_make_symbolic(&m1, sizeof(m1), "m1");
    klee_make_symbolic(&m2, sizeof(m2), "m2");
    klee_assume(m1 == m2);
    ann1.metric = m1;
    ann1.is_dropped = 0;
    ann2.metric = m2
    ann2.is_dropped = 0;
    // Check if route maps r3_to_r1 are equivalent
    Announcement r1 = r3_to_r1_1(ann1);
    Announcement r2 = r3_to_r1_2(ann2);
    assert(r1.is_dropped == r2.is_dropped);
```

```
    if(!r1.is_dropped && !r2.is_dropped){
        assert(r1.metric == r2.metric);
}
```

We have two announcements `ann1` and `ann2` with which we mark fields as symbolic. Since this route map only deals with the metric field, we only mark `metric` and `is_dropped` as symbolic. Normally, we mark all attributes of an announcement as symbolic. If the two route maps are not equivalent, running this C code through Klee will result in assertion errors, and Klee will output the lines of code containing the failing assertions.

In earlier work, Minesweeper is capable of verifying their notion of local equivalence. They define local equivalence as routers that make the same forwarding decisions and export the same set of route announcements given equal input environments. The logical formula which expresses this formula is the following:

$$\begin{aligned} in_1 = in_2 \quad \Rightarrow \quad & (out_1 = out_2) \wedge \\ & (datafwd_{R1,P1} = datafwd_{R2,P1}) \wedge \\ & (datafwd_{R1,P2} = datafwd_{R2,P2}) \end{aligned}$$

The formula represents input announcements as in_1 and in_2, output announcements as out_1 and out_2, routers being compared as $R1$ and $R2$, and two peers of routers $R1$ and $R2$ as $P1$ and $P2$. This formula is dependent on $R1$ and $R2$ being connected to the same peers $P1$ and $P2$. In a large university network, we observed that routers which appeared to be equivalent did not share any peers. Even if the routers did not have identical neighbors, we paired neighbors together which may play similar roles in the network, loosening the idea of "same peers" to "similar peers."

Minesweeper's logical formula for local equivalence verifies for identical control plane behavior given the two router configurations. Our idea of functional equivalence is a stronger guarantee of equivalence, checking the equivalence of control plane configurations by directly comparing the implemented policies. Our approach may find some false positives in some situations where Minesweeper would verify router equivalence.

Minesweeper can be a useful tool for network operators to determine the existence of bugs, but it does not identify the issue in the configuration that causes this bug. Our router equivalence method identifies which attribute of a route announcement has a discrepancy when two supposedly identical functions are applied to the symbolic announcement. With future work, we will be able to map the assertion errors to the corresponding route map in the configuration.

4.4 Quagga Configuration

As mentioned in §2.5, if we are given router configurations in a variety of languages, we would like to create configuration files that are compatible for the Quagga routing software package. This allows us to emulate a network using containers (**MiniNExT**) instead of real hardware.

After Batfish reads in the original router configuration, the resulting data model contains the information required to generate these Quagga configurations. For convenience, we can index dictionaries of the IR to to write some sections of the configuration file rather than directly access the data model. We divide the Quagga configuration generation procedure into the following function calls (including the referenced section from Chapter 2, if applicable):

1. `write_interfaces()`: Iterates through 'interfaces' key in the data model to write interface names and their attributes (e.g. IP address, speed, duplex, etc.). [§2.2.1]

2. `write_ospf()`: Accesses the 'ospfProcess' key in the data model to configure the router's OSPF process. [§2.2.2]

3. `write_bgp()`: Accesses the 'bgpProcess' key in the data model to configure the router's BGP process. [§2.2.3]

4. `write_address_family()`: Configures address-family (e.g. IPv4) specific information for the BGP process. We get the list of advertised subnets and route redistribution information from the `Router` object in `router_dict`. Also, we map route map names to neighbor IP addresses from the `Session` objects in `session_list_dict`. [[§2.2.4]]

5. `write_static_route()`: Reads `Router` object from `router_dict` to write static routes. [§2.2.6]

6. `write_community_list()`: Reads list of `Community_List` objects from `community_list_dict` to define community lists [§2.2.5]

7. `write_as_path_list()`: Reads list of `AS_Path_List` objects from `as_path_list_dict` to define AS path lists [§2.2.5]

8. `write_route_filter_list()`: Reads list of `Route_Filter_List` objects from `route_filter_list_dict` to define route filter lists (i.e. prefix lists) [§2.2.5]

9. `write_route_map()`: Reads the list of `Route_Map_Clause` objects in `route_map_dict` to define the route maps. [§2.2.7]

CHAPTER 5

Results

This chapter describes concrete use cases for the resulting outputs of the object-oriented IR. §5.1 provides an example of our pseudocode representation of BGP policies. §5.2 discusses results obtained from performing equivalence checks for routers in the campus network of a large university. §5.3 goes through an example scenario where we use our network emulation framework to send route announcements to a single router.

5.1 Pseudocode Example

Suppose we have a router with the following two neighbors: 10.14.22.4 and 10.13.22.3. For neighbor 10.14.22.4, we have import policy $as4\_to\_as1()$ and export policy $as1\_to\_as4()$. For neighbor 10.13.22.3, we have import policy $as3\_to\_as1()$ and export policy $as1\_to\_as3()$. We included generated pseudocode for the four functions representing the router's $router\_policy()$ excluding its $local\_policy()$. Each policy function consists of if statements with conditional expressions. The conditionals represent BGP filters (i.e. match statements of route maps). The body of these if statements describe the modifications to route attributes (i.e. set statements of route maps). Additionally, each policy function should also have a final else statement (as show in $as4\_to\_as1$) to express the implicit drop if no earlier condition was satisfied by a given route announcement.

```
// Policies for Neighbor: 10.14.22.4
// Import Policies:
void as4_to_as1() {
    if ( ( ( community == "4:" ) ) &&
```

```
            ( ( prefix == "4.0.0.0/8", prefix_length <= 32 ) ) ) {
        local_pref = 350
    }
    else{
        drop announcement
    }
}
// Export Policies:
void as1_to_as4() {
    if ( ( True ) ) {
        metric = 50
        community += "1:4"
    }
}

// Policies for Neighbor: 10.13.22.3
// Import Policies:
void as3_to_as1() {
    if ( ( ( community == "3:" ) ) ) {
        local_pref = 350
    }
}
// Export Policies:
void as1_to_as3() {
    if ( ( ( prefix == "1.0.1.0/24") ||
            ( prefix == "1.0.2.0/24") ) ) {
        metric = 50
        community += "1:3"
    }
```

```
    else if ( ( ( prefix == "2.0.0.0/8") ||
            ( prefix == "2.128.0.0/16") ) ) {
        metric = 50
        community += "1:3"
    }
}
```

From looking at the comments for function *as4_to_as1*(), we learn that this function represents an import policy with neighbor 10.14.22.4. Function *as4_to_as1*() says that a route announcement received from neighbor 10.14.22.4 with a community value matching "4:" and a network prefix matching 4.0.0.0/8 with prefix length less than or equal to 32 should be accepted and installed as a route in the RIB with a local preference of 350. If a route announcement from 10.14.22.4/32 does not satisfy both these two conditions, the announcement will be dropped and the route will not be installed. These functions are easily interpretable due to the minimal complexity, as they only contain a single level of conditional statements.

§2.2 includes the corresponding lines of Cisco IOS router configuration which express these routing policies. Function *as4_to_as1*() corresponds to the following lines in the configuration:

```
neighbor 10.14.22.4 peer-group as4
neighbor as4 route-map as4_to_as1 in
ip prefix-list as4-prefixes seq 1 permit 4.0.0.0/8 le 32
ip community-list expanded as4_community permit _4:
route-map as4_to_as1 permit 100
 match ip address prefix-list as4-prefixes
 match community as4_community
 set local-preference 350
```

One would need to understand the semantics of the Cisco IOS commands used to reason about the same policy from the configuration. For instance, one must know about the peer-group concept to understand that route map `as4_to_as1` is applied to neighbor 10.14.22.4.

Also, one must understand how route maps are defined and the meaning of match and set statements. Finally, one must be able to interpret the prefix list and community list that are applied to route map `as4_to_as1` to derive the condition of the match statement. There is a large amount prior knowledge required to interpret router configurations. The corresponding pseudocode is much easier to read and understand in comparison.

5.2 Router Equivalence Results

We performed equivalence checks using router configurations taken from a campus network of a large university. We focused on router configurations containing BGP processes that use route maps to control routing policies; otherwise, the routers would have no policies to compare with one another. Through a combination of manual inspection and a neighbor pairing algorithm, we identified nine potentially equivalent router pairs and listed their results in Table 5.1.

Router 1	Router 2	Same # of Neighbors	# of Functions	# of Errors
f98af7	cc4c19	True	6	0
8011c1	b41db8	True	3	1
82e495	c1f60e	True	2	2
54b94c	7c90b6	True	4	2
8eee7f	7e6174	True	3	1
7a0e46	242229	True	4	2
e63866	6c97a7	True	2	1
44a0b7	8b0d2f	False	N/A	N/A
801fde	5d8007	False	N/A	N/A

Table 5.1: Router Equivalence Results

Before we run Klee on the generated equivalence checking code, we check for both routers

to have the same number of neighbors in order to generate the neighbor mapping. We would not know which route maps should be compared without the neighbor mapping. Our neighbor mapping scheme uses a naive algorithm that pairs neighbors with the smallest difference in IP address integer values. We found two instances where the pair of routers had a discrepancy in their number of BGP neighbors. In the case of *44a0b7* and *8b0d2f*, we found that one router included a **neighbor shutdown** command for a BGP neighbor in its configuration file. This command was not present in the second router's configuration file, resulting in a different number of neighbors.

Among the seven pairs of routers that we check for functional equivalence, six pairs had assertion errors with respect to MED values of the announcement. We include an example of a MED discrepancy in outbound route map `43749b_to_e4ccdd`.

```
route-map 43749b_to_e4ccdd permit 10
 match ip address 43749b_246e3b_3c0f46
 set metric 50

route-map 43749b_to_e4ccdd permit 10
 match ip address 43749b_246e3b_3c0f46
 set metric 100
```

ACL `43749b_246e3b_3c0f46` has identical definitions in the two routers' configurations. The only difference is that one router sets the MED for the set of announcements matching network prefixes in `43749b_246e3b_3c0f46` to 50 while the second router sets the MED for this same set of announcements to 100. Our equivalence checking code was able to catch this configuration difference, generating an `Assertion Fail` alert when running with Klee. MED values are used to hint to external neighbors to prefer a path with a smaller MED value [cise]. This implies that the differences in how the route maps set the MEDs may be intentionally configured by the network operator.

A pair of routers that are nearly functionally equivalent may have identical roles in the network. For example, one router could be the main router, while the second router could be the backup router for fault tolerance purposes. If this network wants to notify its neighbors

about this preference, the main router could set a smaller metric in its export route maps compared to the backup router. When external neighbors receive announcements from this router with these modified MEDs, BGP best path algorithm will select the main router as the best path to the advertised network. So although we find that six of the seven tested pairs contained errors, these pairs of routers may have identical roles without satisfying the strict requirements of our equivalence definition.

5.3 Sending Test Announcements

We walk through an example which uses our test announcement code to generate and send announcements through an emulated network, which was set up with MiniNExT using the inferred topology. Then, we verify that the Quagga routing processes update RIBs and propagate announcements as specified by each router's configurations. For the purpose of evaluating our system, we borrowed a sample topology from Batfish [exa]. The topology is shown in Figure 5.1. In this topology there are three different autonomous systems (1,2,3). The routers are running EBGP, IBGP, or both with their neighbors; some routers also run other protocols like OSPF.

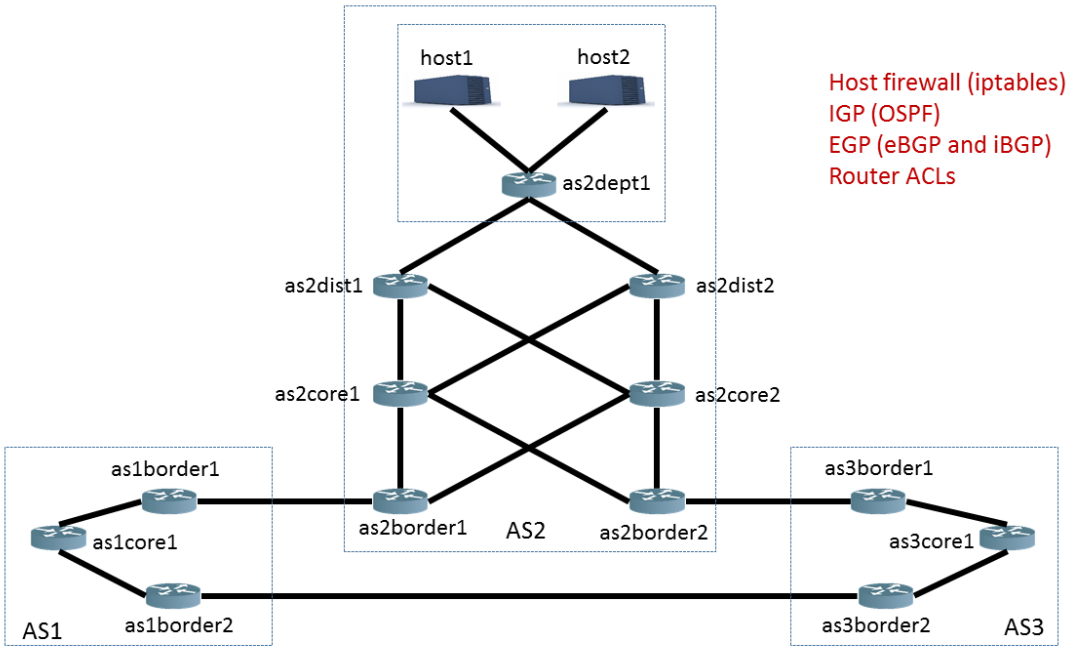

Figure 5.1: Example topology from Batfish

Let `as1border2` be our system under test (SUT). From the IR, we use two of the outputs: Quagga configuration files and test generation C code. Running Klee with our C code gives us the announcements to be sent to the SUT. We emulate the SUT with Quagga process running in the container of `as1border2` and two other containers (`as1core1` and `as3border2`) running an instance of ExaBGP in each of these two containers. From Figure 5.1, `as1core1` and `as3border2` are the sole neighbor nodes of `as1border2`.

The configuration for `as1border2` is the same as the router configuration described in §2.2. We focus on the sections describing BGP configuration (§2.2.3), IPv4 address-family information (§2.2.4), and BGP filters (§2.2.5).

When applying route maps in §2.2.4, we observe that an inbound filter is defined for incoming announcements from AS3 (10.13.22.3). To exercise this route map, the test generation module will decide to send two announcements from as3border2 to our SUT: one with a community tag matching "_3:" and another announcement that does not satisfy the community tag. We observe the effect on as1border2 as the updates to its RIB. The routing table of `as1border2` after the Quagga service is started in `as1border2` and before any route

announcements have been injected is shown in Figure 5.2.

```
as1border2# show ip bgp
BGP table version is 0, local router ID is 1.2.2.2
Status codes: s suppressed, d damped, h history, * valid, > best, = multipath,
              i internal, r RIB-failure, S Stale, R Removed
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*> 1.0.1.0/24       0.0.0.0                  0         32768 i
*> 1.0.2.0/24       0.0.0.0                  0         32768 i

Displayed  2 out of 2 total prefixes
as1border2#
```

Figure 5.2: Routing Information Base of as1border2 before route injection

In the `as3border2` container we start the ExaBGP instance, which will begin injecting routes into `as1border2`. The running snippet of ExaBGP in Figure 5.3 shows two route announcements sent 5 seconds apart. Both announcements advertise the prefix 0.0.0.0/0. The first announcement has community value "3:0," whereas the second announcement has community value "0:0." This exercises both paths of route map `as3_to_as1`, which sets local preference to 350 for routes matching community tag "_3:" and otherwise dropping route announcements that do not satisfy this condition.

```
11:27:49 | 60     | configuration | performing reload of exabgp 4.0.2-1c737d99
11:27:49 | 60     | reactor       | loaded new configuration successfully
11:27:49 | 60     | reactor       | connected to peer-1 with outgoing-1 10.13.22.3-10.13.22.1
11:27:54 | 60     | api           | route added to neighbor 10.13.22.1 local-ip 10.13.22.3 loc
al-as 3 peer-as 1 router-id 10.13.22.3 family-allowed in-open : 0.0.0.0/0 next-hop self commun
ity 3:0
11:27:59 | 60     | api           | route added to neighbor 10.13.22.1 local-ip 10.13.22.3 loc
al-as 3 peer-as 1 router-id 10.13.22.3 family-allowed in-open : 0.0.0.0/0 next-hop self commun
ity 0:0
```

Figure 5.3: ExaBGP working snippet

After the first announcement is sent, as1border2's RIB changes as shown in Figure 5.4. The announcement matches the community tag condition of route map `as3_to_as1` defined for incoming routes from neighbor 10.13.22.3, so the SUT accepts and installs this route, updating the local preference to 350 as specified by the set statement in the route map.

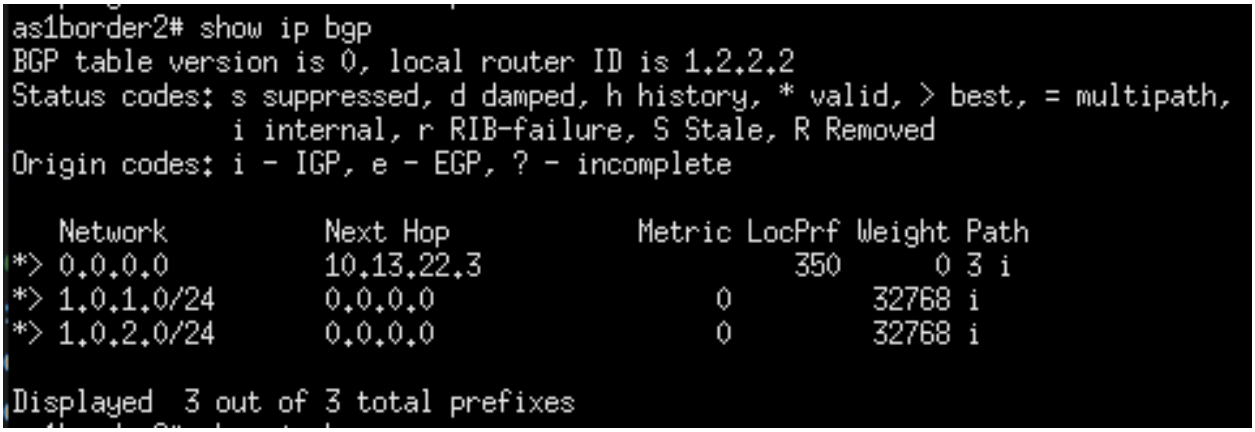

Figure 5.4: Routing Information Base of as1border2 after the first announcement

After the second announcement is made, the RIB will revert back to its initial state (Figure 5.2). This occurs because the second announcement advertises the same prefix as the first announcement, so as1border2 considers it an update to the attributes of this prefix. The community tag being announced is "0:0," which ultimately gets filtered at `as3_to_as1`, so this route is no longer installed in the RIB.

CHAPTER 6

Conclusion

This thesis introduced an object-oriented representation of router configurations, encoding information about a router's BGP policy. Using this IR, we produce a set of outputs with a variety of uses.

We present a method to perform router equivalence checks by evaluating for functional equivalence in C programs generated by the IR. We were able to accurately find BGP-related discrepancies in router configurations. Future work will be done to alert the network operators of the discovered configuration differences to compare with their intended policy.

The C programs created from the IR can be run with Klee to generate high coverage tests. These tests consist of BGP announcements which can be sent into an emulated network. We have established the framework for control plane testing in the emulated environment using Quagga, MiniNExT, and ExaBGP. Our generated Quagga configuration files are used as input to this network to model real routing processes. As our control plane testing infrastructure is in it infancy, future work can be done on automating the analysis of the effects of test announcements on the network, and expanding test generation from analyzing a single node to computing an optimal number of tests over a whole network.

After reading router configurations and vendor documentation, we have a better understanding of how BGP configuration errors arise. Configuration languages, such as Cisco IOS, often have default values and settings which can hide configuration semantics. Router configurations are typically generated based on a common template, and then network operators modify the template to implement the router's policy. As the modifications are made, the network operator's intended policy may not match the semantics of the configuration.

Furthermore, Cisco IOS configurations use concepts (peer groups, route maps, commu-

nity lists, prefix lists, etc.) to express policies, but these levels of indirection can be confusing for people who want to understand the policy being implemented by the configuration. The generated pseudocode provides an easier way to interpret a router's import and export policies by representing these policies as functions of code with conditional statements. Network operators can compare their intent with the pseudocode rather than directly with the router configurations, which are much more difficult to comprehend.

We currently generate C code to use in conjunction with Klee, but there can be further uses for a C representation of BGP policies. For instance, if a network operator has identified an issue where an announcement does not reach an intended target, we can use the C code with a debugger (GDB) to step through each router's policy and determine which router along the path has a route filter which causes the input announcement to be dropped. Any tools that can be used with C code can potentially be used to help debug or better understand the network.

Future work includes augmenting our IR to include additional BGP features (e.g. route reflectors, route aggregation, etc.) and including other networking protocols (e.g. OSPF, IS-IS, etc.) to have a complete model of the network. Additional future work to understanding BGP configurations and policies includes being able to infer a router's role in a network. This would allow for comparing the configurations of routers with identical roles. Furthermore, we may be able to expand on our equivalence testing by performing surgeries on router configurations before checking for symmetric behavior [PBL16].

REFERENCES

[bat] "OpenConfig." https://github.com/batfish/batfish.

[BGM17] 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, pp. 155–168, New York, NY, USA, 2017. ACM.

[bgp] "BGP Best Path Selection Algorithm." https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/13753-25.html.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs." In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pp. 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[cisa] "Cisco IOS IP Command Reference, Volume 2 of 3: Routing Protocols, Release 12.2." https://www.cisco.com/c/en/us/td/docs/ios/12_2/iproute/command/reference/fiprrp_r.html.

[cisb] "Cisco IOS IP Configuration Guide, Release 12.2: Configuring BGP." https://www.cisco.com/c/en/us/td/docs/ios/12_2/ip/configuration/guide/fipr_c/1cfbgp.html.

[cisc] "Configuring IP Access Lists." https://www.cisco.com/c/en/us/support/docs/security/ios-firewall/23602-confaccesslists.html.

[cisd] "IP Routing: BGP Configuration Guide, Cisco IOS XE Release 3S: BGP Additional Paths." https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_bgp/configuration/xe-3s/irg-xe-3s-book/bgp-additional-paths.htmll.

[cise] "OpenConfig." https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/13759-37.html.

[CS13] Cristian Cadar and Koushik Sen. "Symbolic Execution for Software Testing: Three Decades Later." *Communications of the ACM*, **56**(2):82–90, February 2013.

[exa] "Batfish Example Topology:." https://github.com/batfish/batfish/tree/master/test_rigs/example.

[FB05] 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, pp. 43–56, Berkeley, CA, USA, 2005. USENIX Association.

[FFP15] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. "A General Approach to Network Configuration Analysis." In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pp. 469–483, Oakland, CA, 2015. USENIX Association.

[FSF16] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. "Efficient Network Reachability Analysis Using a Succinct Control Plane Representation." In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pp. 217–232, Savannah, GA, 2016. USENIX Association.

[GVA16] 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, pp. 300–313, New York, NY, USA, 2016. ACM.

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. "A Network in a Laptop: Rapid Prototyping for Software-defined Networks." In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, HotNets-IX, pp. 19:1–19:6, New York, NY, USA, 2010. ACM.

[Man] Thomas Mangin. "ExaBGP project documentation, available at:." `https://github.com/Exa-Networks/exabgp`.

[MWA02] Ratul Mahajan, David Wetherall, and Tom Anderson. "Understanding BGP Misconfiguration." In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, pp. 3–16, New York, NY, USA, 2002. ACM.

[ope] "OpenConfig." `http://www.openconfig.net`.

[PBL16] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. "Scaling Network Verification Using Symmetry and Surgery." In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pp. 69–83, New York, NY, USA, 2016. ACM.

[Qua] "GNU Quagga Project." `https://www.nongnu.org/quagga/`.

[RLH06] Yakov Rekhter, Tony Li, and Susan Hares. "A Border Gateway Protocol 4 (BGP-4)." RFC 4271, RFC Editor, January 2006.

[sin] "Mininet multiple container problem." `http://mininet.org/walkthrough/#interact-with-hosts-and-switches`.

[SZC14] Brandon Schlinker, Kyriakos Zarifis, Italo Cunha, Nick Feamster, Ethan Katz-Bassett, and Minlan Yu. "Try Before you Buy: SDN Emulation with (Real)

Interdomain Routing." In *Presented as part of the Open Networking Summit 2014*, ONS 2014, Santa Clara, CA, 2014. USENIX Association.

[ZKV14] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. "Automatic Test Packet Generation." *IEEE/ACM Transactions on Networking*, **22**(2):554–566, April 2014.