Automated Testing of Forwarding Policies

Vibhaalakshmi Sivaraman

Adviser: David Walker

Abstract

BGP routers can prove to be challenging to configure, manage and debug owing to their distributed nature. "Propane", a system aimed at overcoming this, compiles policies at the AS (Autonomous System) level from the network operator down to individual router configurations. In this paper, we propose a testing framework aimed at establishing correctness of the Propane compiler. This generates exhaustive test cases to ensure that valid advertisement sequences lead to actual network traffic paths and that inherent preferences between multiple traffic path choices are exercised. We encode the test generation procedure as a SAT instance on the Propane Product Graph. We simulate the generated tests as individual BGP sessions, verifying network properties in the simulation. The test infrastructure generates and verifies test cases on small networks upto 100 *routers within a reasonable time interval. This approach paves way for further testing of router configurations' adherance to more generic policies than Propane itself.*

This work represents my own work in accordance with university policies.

Vibhaalakshmi Sivaraman

May 4, 2017

Contents

1. Introduction

Today's world runs on the Internet, which connects over 3 billion people across various university and corporate networks, regional and nationwide Internet service providers [26]. These networks, comprise of a number of Autonomous Systems (ASs) or a collection of routers each of which use the Border Gateway Protocol (BGP) to reach other routers both within the AS as well as on the Internet. The protocol uses a series of advertisements from neighboring ASs or adjacent routers in the same AS that detail which other ASs and/or prefixes can be reached from the neighboring AS. In addition, these advertisements also contain the actual path towards the destinations. Based on these advertisements and its own local policy, a router can update its view on which prefixes it can reach and what the paths to those prefixes are. Individual router policies are also responsible for restricting and altering certain advertisements before they are sent out to more peers to prevent misuse of these messages for route-hijacking or potentially turning down an entire network. Broadly speaking, the propagation and processing of these advertisements based on individual routers' local rules help them construct their view of the network and reachability within it, which are crucial to the network functioning as a whole. Consequently, reliable configuration of the routers in the network is incredibly important for actual traffic transmission in the network.

However, often BGP routers are misconfigured leading to widespread outages ([12], [1], [2]). In 2008, for example, Pakistan attempted to block Youtube in the country by advertising a fake route for all traffic directed towards Youtube. This led to all such traffic being dropped. However, it ended up advertising this fake route to not just Pakistan but the rest of the world [12]. Consequently, all Youtube traffic across the world was routed through this fake black-hole route, affecting global access to Youtube. While this was addressed shortly after, a correct implementation wouldn't have caused these issues.

In practice, each AS comprises of numerous routers working together in a distributed fashion. How network traffic traverses this AS is solely dependent on how individual routers process their

neighbors' messages. This, in turn is determined by individual router configurations. These configurations are typically installed and maintained by network operators manually without any help from tools. The distributed nature of BGP makes this particularly challenging, making the system highly error-prone in the process [26]. This becomes even worse as the network size grows.

Naturally, this makes fertile ground for extensive research and improvements to make router configurations more robust. Most of the existing research can be classified into a few broad categories. A number of solutions aimed at analyzing router configurations catch bugs before these configurations are deployed and used in real networks. Most of these solutions ([15], [20], [28]) are static in nature and do not adapt very well to the changing dynamics of BGP configurations. A second solution is to see traffic in action and capture dataplane snapshots ([31], [22]). These solutions rely on events going wrong in the network and then later attempt to understand what configuration errors may have led to this. A subset of these solutions ([19], [21]) are aimed at checking properties dynamically over incremental updates.

A whole other direction of network verification involves creating models of what the network and packets look like under any environment and then performing symbolic executions in that environment. This often involves program verification style theorem proofs to establish the properties satisfied by the network ([26], [14]). Alternatively, one could test properties by simulating a wide-variety of network environments. [17], [25] and [30] use simulations of actual BGP sessions to verify that set of common properties hold. This isn't as exhaustive as symbolic execution which theoretically verifies *all* possible network environments.

In a somewhat more radical manner, there has been effort to get rid of this distributed nature altogether. For instance, the Routing Control Platform was proposed as a means to centralize all decision making on how to reach a certain prefix [9]. The advent of Software Defined Networking [16] is also in the same vein. There has been effort to produce "correct-by-synthesis" configurations. [6] and [24] use a higher level language to make it easier to specify what the network requirements are at the operator's end and then compile it to low level network configurations that adhere to those

requirements. Propane [6], in particular allows a network operator to specify policies regarding what paths the traffic should take, what prefixes should be advertised where, which path should be preferred in the event of multiple paths, etc. This system essentially "bridges the gap" between the network operator's specifications and the low-level router configurations.

In this paper, we combine two of the existing procedures to ensure the correctness of the router configurations. We take Propane and test it for its compiler correctness under a variety of network environments. We verify that the network behavior corresponds to the expected behavior when each of these are simulated as individual BGP sessions. In particular, we define the following two notions of coverage for testing Propane and develop a system to test Propane under those notions of coverage.

- Any valid sequence of advertisements from source to destination allows for a corresponding forwarding path for traffic
- If a particular router receives multiple advertisements which differ in history, it forwards out the preferred advertisement

The test generation problem is reduced to an instance of the SAT problem which can be solved with general purpose solvers. The outputted test cases, which are essentially individual BGP sessions, are simulated in a C-BGP environment and the expected traffic path properties are tested. The test infrastructure generates exhaustive environments for networks upto 125 routers (moderately sized by today's network standards) within a reasonable time period. Within a span of a couple of minutes, these tests can all be simulated as BGP sessions and the network properties verified to ensure that the Propane compiler is indeed correct.

The rest of this paper is organized as follows. §2 provides a brief overview of the Border Gateway Protocol and what risks it exposes, followed by more detailed descriptions of current solutions. §3 discusses the architecture of our proposed test framework and how it related to the existing Propane framework. §4 defines the notions of coverage for our test framework while §5 presents the exact SAT encodings needed to generate the test cases to meet those coverage notions. §6 presents the

implementation details on how these tests were simulated as individual BGP sessions. We present the evaluation of the system in §7 and briefly discuss the results and possible extensions to the framework in §8.

2. Problem Background

2.1. BGP Background

BGP or Border Gateway Protocol is a path-based routing protocol wherein a given router learns about paths to a certain destination prefix or router via its neighbors. It allows multiple Autonomous Systems or ASs (a set of routers managed by a single entity) to be connected to each other via this learning process. The links between routers belonging to two ASs is different compared to the links between routers within the same ASs in that they are treated differently. The former are considered eBGP links while the links internal to an AS are considered iBGP links.

Communication between routers involves sending advertisements from a given router to all its neighbors. These advertisements contain information on which prefix the neighbor can reach and what AS level path the neighbor has towards that prefix. These advertisements might also have certain other attributes like community values and multi-exit discriminators. These values are added within the AS and help classify the advertisements to establish different ways of processing them at routers. For instance, a particular community value could be associated with a certain region from where the traffic came or a certain class of routers being present on the path. The multi-exit discriminator value is used to establish preference multiple points of exiting an AS out towards an external neighbor or peer.

Once an advertisement (and its associated attributes) is received by a certain router, it processes it based on its local rules before exporting it or its way of reaching a certain prefix onto its peers. For a given prefix, the router makes its choice on its most preferred path (if multiple advertisements corresponding to multiple paths are received by the router) and sends only that one path onto its neighbors. The local rules at a given router consist of import and export filters that perform select

actions on the advertisements according to conditions. For instance, based on a community string that the advertisement matches, it might assign a different local preference when importing it. This helps the router differentiate between multiple potential advertisements it might receive before deciding which one must be exported out. It also helps prefer customer paths over peer or provider paths. The export rules control what peers a certain route is disseminated to (if at all it is to be disseminated and not merely denied) and with what properties. For instance, if the community value must be changed or removed since the differential processing at this router is complete, this is specified as part of the export rules of the given router. The router takes care of this before passing advertisements on to its peers.

Based on the router configurations and the actual advertisements received by a router, the router is able to update its routing table with information on how to reach a certain prefix. This dictates all movement of traffic in the network. Thus if these routers are incorrectly configured, they won't have the right routing tables consequently affecting traffic movement in potentially extremely harmful ways. Given how common router configurations are ([12], [1], [2]), a natural question to ask is: *Can we make router configurations more robust*?

2.2. Related Work

As mentioned in §1, there have been a number of solutions aimed at mitigating the issues associated with router misconfigurations. These can be classified into a few broad categories which have their own pros and cons associated with them. In the following paragraphs, we discuss some of these categories and the specifics of the most popular approaches within them in more detail.

Static Configuration Analysis: Given that router configurations pose bulk of the threat that leads to outages, a natural place to check for errors would be the configuration files themselves. For instance, rcc [15] analyzes the configurations before they are deployed for route validity errors. Such errors occur when the route either corresponds to an invalid or unusable path or certain valid routes are missed altogether. Fireman [28] restricts itself to firewall configuration analysis. [27]

and Header Space Analysis [20] both use a model for packets and generate modification/transfer functions based on the configuration of individual routers. These can then be used to identify the various end states a packet can end up in and check if any of these are concerning or inconsistent with what is expected. However, all of these are based solely on configuration analysis and do not simulate real traffic.

Dynamic Dataplane Analysis: Anteater [22] is an approach that uses the dataplane instead of router configurations, but still performs a static analysis of dataplane. But, a number of other solutions have been developed that are directed towards a more dynamic analysis of the dataplane, particularly as incremental updates are performed and rules inserted. Veriflow [21] serves as a layer between the network controller and device configurations that verifies invariants before any updates go through to the devices. NetPlumber [19] draws from HSA [20], but also checks for compliance of state changes as they happen. It does so by maintaining a dependency graph between rules.

Testing Approaches: Testing approaches are based on a certain notion of exhaustiveness depending on the property of interest. For instance, Batfish [17] checks dataplane snapshots for a wide range of forwarding properties and generates actual packets that violate these expected properties. Automatic Test Packet Generation [30] also does something similar but based on router configurations in a device-independent manner. When failures are detected, a different mechanism is initiated that localizes the fault. C-BGP [3] is often used for simulating a number of these network environments for property verification. Outside the context of network verification, DART [18] is a random testing approach for software verification that generates specific inputs to test every path of program execution. This is better than generic random testing that might generate multiple inputs that use the same execution path. Some of our techniques are inspired by this approach.

Network Models: A broad set of network verification solutions that use a model of the network environment to prove that certain invariants are always maintained fall under this category. Bagpipe [26], for instance, encodes the network environment and packets symbolically and evaluates them using an SMT-based execution engine. It searches for counterexamples that violate the expressed

policies. Given that it doesn't restrict itself to actual network environments, but rather abstract ones, this is considered more exhaustive that simulations or testing. ERA [13] builds a model of the network control plane that generates all the lower network environments or dataplane components. By modeling the controller itself, this allows one to explore all possible network environments that might be generated by the central controller. Buzz [14] is a combination of this approach and testing. It is directed at stateful networks and operates by building a model for the dataplane before generating abstract traffic to test its properties.

Centralized Control: With the advent of SDN [16], there has been a lot of effort to create high level languages that make it easier to centralize all the control in a network. Decisions are made at this central authority that are then disseminated to individual routers or switches as a series of match action rules. The Routing Control Platform [9], a much earlier solution, is in line with this spirit and discusses a central platform that has full visibility into all the BGP advertisements in the network and is thus able to make the best decisions on which routes should be taken towards a given prefix. This is in direct contrast to the distributed approach where each router only gets information from its neighbors. However, this centralized approach comes with the possibility of a single failure bringing down the entire network.

High Level Languages: Propane[6] and ConfigAssure [24] argue for bridging the gap between network requirements and the low-level configurations that implement them. Propane takes in the requirements for the AS (Autonomous System) as a policy specification which restricts what paths traffic can take, what advertisements should be sent where and which path should be preferred. It then compiles these restrictions down to router configurations. ConfigAssure, on the other hand, treats the specification as a set of constraints and the configurations as information on the variables and checks if a satisfying solution exists. If it does, that solution is mapped to the configuration directly. If a solution doesn't exist, ConfigAssure can pinpoint to the constraint that is not solvable or in other words, the error in the configuration. It also suggests repairs to these errors. While these systems are very practical and greatly simplify the network operator's job, there are no

guarantees on the system without testing its correctness. Thus, in this project, we take inspiration from two categories of prior work and propose a testing framework for Propane that can establish its compiler's correctness.

3. System Architecture

In this section, we describe the architecture of the testing framework we develop. The broad context that the framework fits in is described in Fig. 2 and the testing stack itself is shown in Fig. 4. We describe the front-end that the tests are designed for in §3.1, the input to the test framework in §3.2 and the tools that the framework uses in §3.3.

3.1. Propane Frontend

Propane [6] is a high-level language that allows network operators to specify objectives for a given autonomous system (AS). It then compiles this "high-level policy" down to low-level BGP configurations. These policies can be used both to constrain the shape as well as express preferences on the routes that traffic within the network takes. For instance, Propanes allows one to restrict classes of traffic to be solely internal to the AS, to start or end at a particular router, to never traverse a certain router, etc. Furthermore, the high-level language allows you to express preferences amongst the available paths. These paths in contention are specified using a certain set of rules that appear like regular expressions on the path itself.

Before specifying the policy itself, Propane needs a notion of what the topology of the network is. This is specified in a separate file wherein the routers in the network and the connections between them are defined in markup style. Propane allows one to define a topology consisting of both *internal* as well as *external* routers. The internal routers are the ones for whom the router configurations are to be generated, while the external routers might still be required to define how traffic should flow to or from nearby ASs.

Let us assume a network has the topology as in Fig. 1. The network operator is concerned about

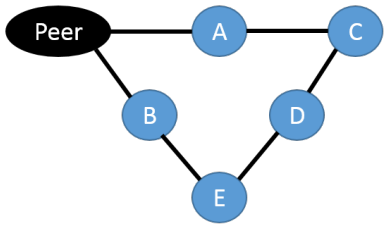


Figure 1: Sample network topology consisting of an external *Peer* and internal routers *A* through *E*

controlling the paths that the traffic takes in this topology. In particular, all traffic matching the prefix 172.0.0.0/24 must end at the external "Peer" and when exiting the AS should prefer the router *A* over router *B*. Secondly, all traffic matching the prefix 1.2.3.4/24 should end at the internal router *E* and prefer entering the AS through router *A* over router *B*. No other traffic should enter this network. The propane network-level policy for this behavior is as simple as

```
define main = {
    172.0.0.0/24 => exit(A » B) & end Peer
    1.2.3.4/24 => enter(A » B) & end E
    true => drop
}
```

The propane compiler [6] uses a series of intermediate steps to translate the above mentioned high-level policy specification into low-level router specifications, which are described in Fig. 2. The policy is captured in the form of a Propane Frontend. These rules are then converted using rewriting of the rules and substitution in terms of regular expressions to give the Regular Intermediate representation (RIR). These policies are then checked for well-formedness to ensure that there are no rules on traffic that is not owned by the AS or on traffic that might not even enter the network. This is then combined with the topology of the network to result in the Product Graph Intermediate Representation (PGIR). The PGIR captures the actual states that advertisements can be in as they pass through the given network under the constraints imposed by the policy. This

PGIR is then processed by the compiler to come up with abstract BGP configurations to process the advertisements in the desired manner. This can then be translated into configurations for vendor specific targets like Quagga or Cisco. This processing, from PGIR to abstract BGP, forms the bulk of the system and a number of strategies are employed to prevent BGP instability and black-hole aggregation (aggregation of traffic that might result in it being lost at a certain internal router). In this work, we test this portion of the compiler, as shown in Fig. 2.

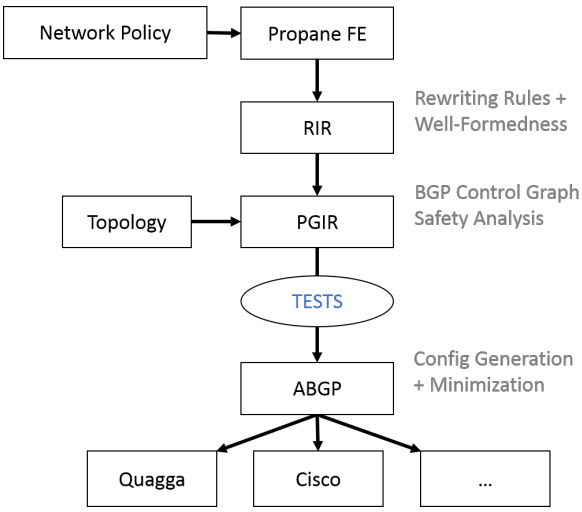


Figure 2: The testing infrastructure tests the conversion of the Product Graph Intermediate Representation (PGIR) into the abstract BGP configurations (A-BGP) (Figure modified from [6])

3.2. Propane Product Graph

The Propane Product Graph Intermediate Representation (PGIR), as described in Fig. 2, is the step that immediately follows the front-end. A sample PGIR is shown in Fig. 3. While the front-end captures the user specified policy and constraints on the traffic flow itself, the PGIR takes in both the topology and the traffic constraints to capture what the actual BGP advertisements in the network look like. In particular, for every constraint and its associated Regular Expression Intermediate Representation (RIR), a PGIR is constructed. This PGIR essentially captures what the policy states (whether the advertisement has matched a given policy or not or is in some intermediate partial match case) are for a given automaton as route advertisements are disseminated from one location

to another on the actual topology. Every PGIR node has both a state number and a router location associated with it as shown in Fig. 3. Some PGIR nodes might have a rank associated with them to denote the preference of an advertisement path ending at them. Most paths in this PGIR corresponds to a valid path in the topology that the advertisements can take, resulting in a certain path that the traffic will take. Since traffic flow is opposite to the flow of advertisements, the PGIR is constructed using the reverse form of the RIR, consequently making traffic paths opposite to the paths from source to destination in the PGIR.

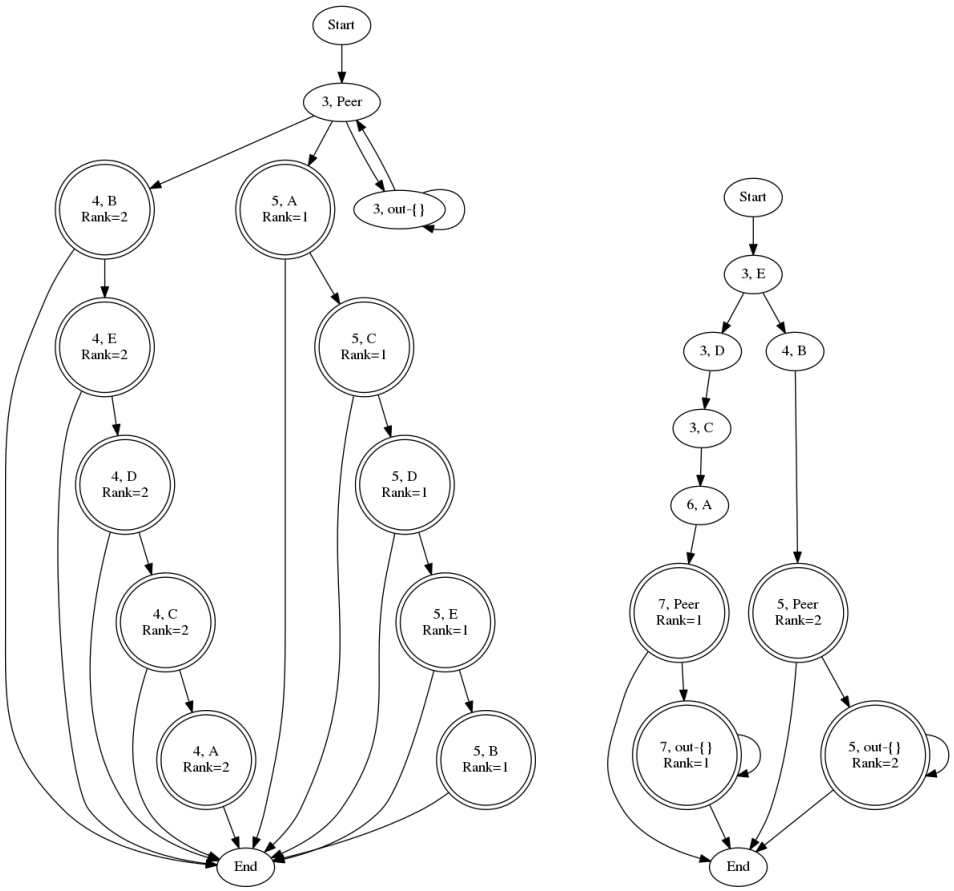
For instance, for the second constraint in the policy above for traffic in the prefix 1.2.3.4/24, we could express the RIR in the form of

$$(\mathbf{out}^{+} \cdot \mathtt{A} \cdot \mathbf{in} \cdot \mathtt{E}) \gg (\mathbf{out}^{+} \cdot \mathtt{B} \cdot \mathbf{in} \cdot \mathtt{E})$$

The minimized PGIR (original PGIR, followed by some optimization to make it smaller) for that constraint is shown in Fig. 3b. According to this PGIR, the BGP advertisements can take two paths. One is $E \cdot D \cdot C \cdot A \cdot Peer$ and the other is $E \cdot B \cdot Peer$. The Ranks on the two "Peer" nodes in the graph suggests that the former is more preferred, which is consistent with the policy specification. In our testing infrastructure, we use this PGIR and generate a number of paths corresponding to the expected behavior, permissible paths and preferences amongst them in order to test the system.

3.3. Tools

In order to test the system, we take in the product graph generated by the compiler and generate an exhaustive set of tests cases, where "exhaustive" is defined by the notion of coverage employed. We define our notions of coverage in §4. Each test case corresponds to an expected path that the traffic should take between source and destination under certain network conditions. Network conditions are determined by which links are up/down and what sets of BGP advertisements might be received by a given router, which consequently affect routing tables and the paths available for traffic. The testing infrastructure, as represented in Fig. 4 involves a series of steps:



(a) First (b) Second

Figure 3: Minimized product graph for the first and second constraint on the topology respectively in Fig. 1. Each node has an associated state number from the automaton, router name and a rank if it is adjacent to the "End" node

1. Use a z3 SAT Solver to generate "exhaustive" test paths for the coverage notion (§5)
2. Translate each test into individual files with "C-BGP" import/export filters for routers (§6)
3. For each test case
   (a) Simulate an individual BGP session using C-BGP
   (b) Verify that the actual path taken by traffic matches the expected path.

A more detailed description of the tools used for this purpose is below

Z3 ([29], [10]) is a high-performance theorem prover that takes in a series of constraints on variables and returns whether the given set of constraints is "SATISFIABLE" or "UNSATISFIABLE". For instance, the Z3 API lets you define Boolean variables corresponding to whether an edge or a vertex

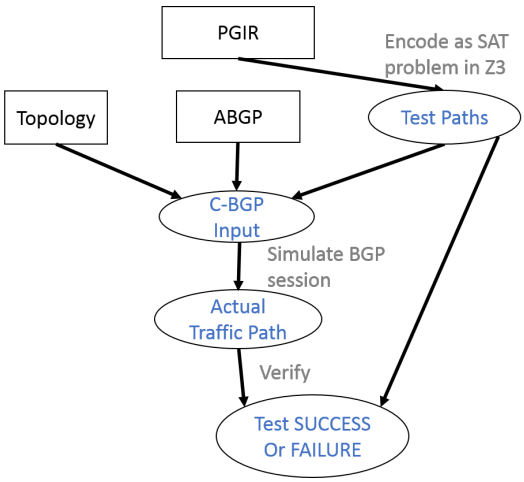


Figure 4: The testing infrastructure constructs a SAT instance on the PGIR that generates test paths that are exhaustive. The topology is used to specify the network topology in C-BGP format. Using the router configurations from the Abstract BGP for the routers on the test path, the C-BGP import and export filters are constructed. This is then simulated as a unique BGP session. The actual path taken by the traffic is compared against the expected path to determine if the test succeeded or failed.

is on a given path or not. Furthermore, you could combine these expressions together using simple boolean operators like "Or" and "And". One can also construct more complex implication statements between individual expressions as well as expressions of the type "atleast-one", "exactly-one" or "utmost one" amongst a set of existing variables. If the constraints, as described and asserted, are satisfiable, a model of the satisfiable solution can be extracted. Given that our path generation problem and its need for exhaustiveness point towards an NP-Hard problem, encoding it in the form of a SAT problem allows us to exploit existing tools and advances ([7], [23]) that can solve these problems in as efficient a manner as possible. The Z3 interface also allows for a number of optimizations and reusing of prior work done to solve the instance that can make iterative solving faster. The precise encoding of the problems according to the coverage notions can be found in §5

cBGP [3] is a tool that simulates the BGP decision process given a network topology, a set of routers and configuration rules for each of them. It allows the presence of both internal routers with specific rules as well as external routers and their associated advertisements. In particular, it allows you to specify import and export filters on the advertisements for internal routers. These can be specified using a set of match-action rules. The actions range from basic acceptance and denial

of the route to setting or removing community or MED (Multi-Exit Discriminator) values or even performing AS path-prepending. Further, once the BGP session has been simulated, the actual path can be traced using a "Traceroute" type command for BGP. This lists the ASs that the monitored traffic visited between its source and destination ASs if the destination prefix is accessible and owned by a certain AS in the network. Note that Propane allocates a separate AS to every router in the network, so all the sessions in the simulated C-BGP environment are external BGP (eBGP) sessions. Consequently, every router between the source node and the destination node (that owns the prefix) will be recorded as a separate AS. This allows us to use traceroute to get the per-hop path, since the per-hop path is the same as the per-AS path. This simulation process is described in more detail in §6.

4. Notions of Coverage

§3 discussed how the testing stack looks like and where it fits into Propane's pipeline. It uses z3 to generate an exhaustive set of test cases on the Propane Product Graph and C-BGP to simulate these individual test cases. In this section, we describe in detail the notions of "exhaustiveness" in the context of our testing framework. §5 goes into more detail on the precise encoding of the test generation problem as a SAT instance.

Since Propane [6] is a system that inherently lets operators specify constraints on the paths that traffic should take in the network and particularly preferences on the paths that the traffic can take, we would want to test that the traffic indeed takes the path that its expected to under all circumstances. Furthermore, the PGIR as described in §3.2 allows for a natural computation of expected paths in the network and preferences amongst them, by tracing through the flow of advertisements in the network. Note that the traffic path is opposite to the advertisement path as denoted by the PGIR. Bearing this "path" approach in mind, we define two notions of coverage:

- **Link Coverage:** Any valid sequence of advertisements from source to destination allows for a corresponding forwarding path for traffic. This notion enforces individual link usage under

different advertisement sequences.

- **Preference Coverage:** If a particular router receives multiple advertisements which differ in history, it forwards out the preferred advertisement. This leads to the preferred path being taken when reaching a given destination prefix from a the router when multiple paths exist.

We define these notions in detail below guided by the topology and propane policy presented for Fig. 1.

4.1. Link Coverage

The idea with link coverage is that every single edge (PGIR edge maps to an advertisement) in the product graph should lead to atleast one valid path via a sequence of advertisements in the opposite direction. In particular, this path should use the topological link referred to by that product graph edge (say l) in the event that there is no other more preferred path in the network. So, if we first turn off all other links and sequences of advertisements in the network and just turn on everything on the generated test path that is utilizing l, there is no other preferred path possible. Consequently, traffic should traverse this path if the routers are configured correctly. Furthermore, a given PGIR for a policy has exactly one source router for a given prefix and that router owns that particular prefix. Hence this advertisement path using l will always start at that particular router and traffic on the opposite path will end there.

We generate multiple such paths or advertisement sequences for every edge l until we have exhausted all the possible edges or no more paths exist in the PGIR even if some topological links or PGIR edges are reused across multiple paths. We define each path or an ordered set of topological links as a single network environment (comprising of a history or sequence of advertisements) under which we want to test the BGP router configurations. We test all these network environments through BGP simulations in C-BGP [3] as described in §6 and ensure that the path taken is compliant with the expected path as specified by the Propane policy specification and expressed in the PGIR.

For instance, in the context of the propane policy discussed for the topology in Fig. 1, we consider

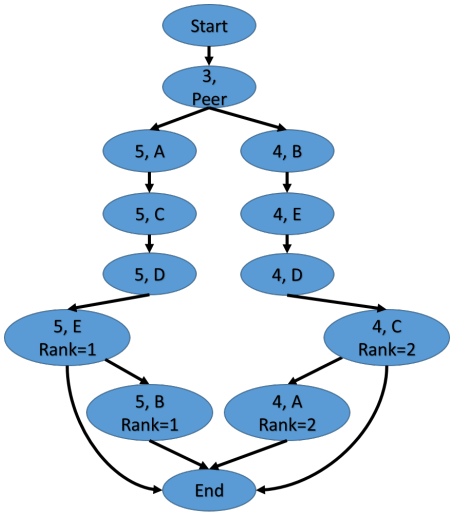


Figure 5: Smaller portion of the PGIR in Fig. 3a that leads to the paths in Fig. 6 when the link coverage testing criteria is applied and the paths in Fig. 8 when the preference coverage testing criteria is applied

the subgraph Fig. 5 of the original PGIR in Fig. 3a. The four paths (broken lined edges) in Fig. 6 show what it would mean to *exhaustively* test this subgraph of the main PGIR for *link or edge coverage*. These paths also need to satisfy some other properties in order to be "valid" paths. This includes not having cycles in the paths generated, both from the product graph, but also in the context of not reusing the same topological node or router (across two different product graph nodes) twice in one path. We also avoid paths through the special "Out" nodes. This is shown in Fig. 7 which comprises of the exhaustive test cases on the PGIR in Fig. 3b.

If a router that is "external" to the AS itself is present in the path, special care needs to be taken while setting up the router configurations for the simulation. This is described in more detail in §6. In practice, we cannot control what advertisements these routers send out. However, for the sake of the simulation, if a certain path's validity and consequent verification relies on this router advertising its ownership or knowledge of a certain destination prefix, we need to ensure that these "external routers" are instructed to advertise this information even though programming them is outside the network operator's jurisdiction. Furthermore, since we have a unique PGIR associated with a given policy constraint on a given prefix, the test generation procedure needs to be repeated for all the PGIRs associated with a network, until each of their edges has been exhausted or cannot

be exhausted any further.

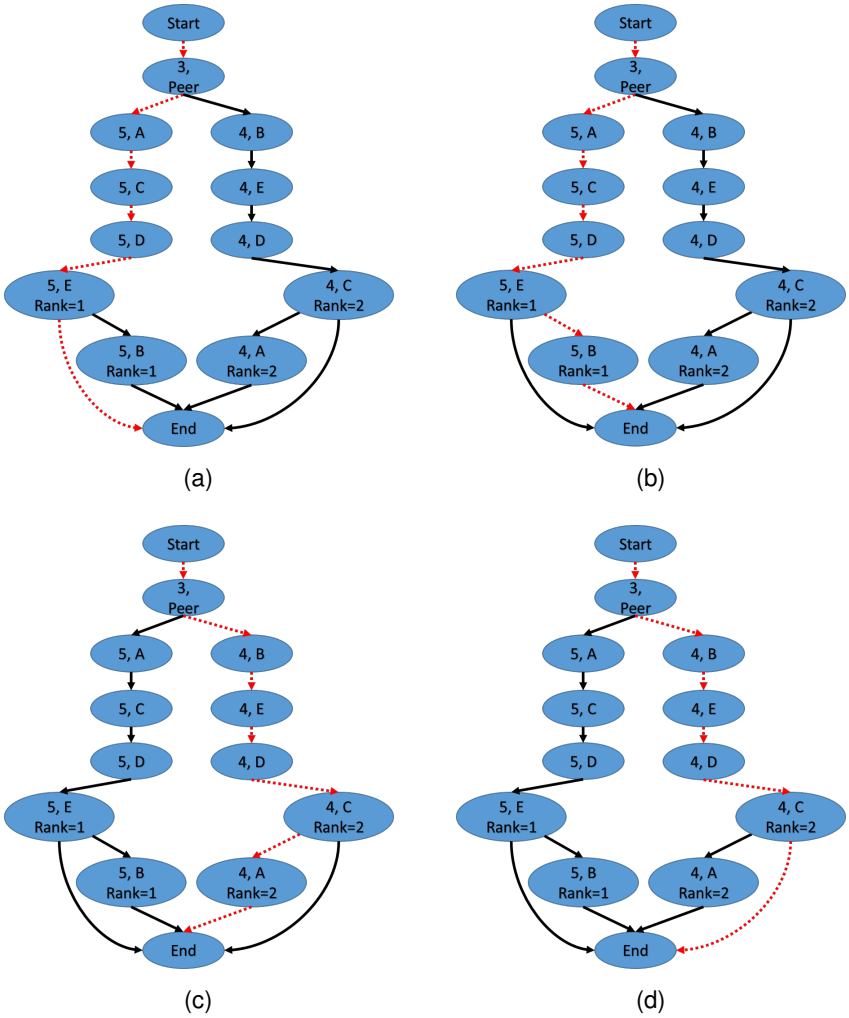


Figure 6: Test cases (indicted in red broken lines) for link coverage on the PGIR from Fig. 5 that give exhaustive paths from "Start" to "End" that between them use all the edges in the Product Graph

4.2. Preference Coverage

Preference coverage goes one step beyond link coverage in that it tests multiple paths at once. When the network operator specifies that he/she would prefer a certain path p_1 over p_2, the preference coverage notion ensures that when both of these paths are feasible in the network, p_1 is the path that is employed by the traffic. In the context of Propane's PGIR, such a policy manifests in the form of a particular PGIR node having a higher rank than another node in the same PGIR that it shares its

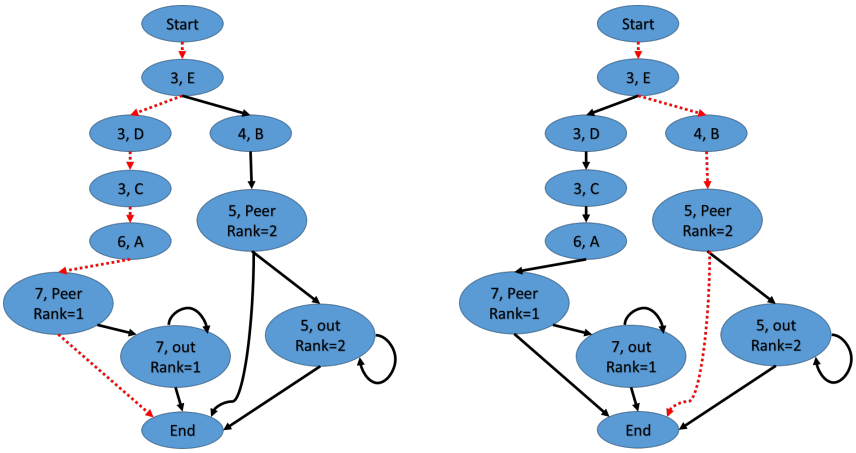


Figure 7: Test cases (indicted in red broken lines) for link coverage on the PGIR from Fig. 3b that give exhaustive paths from "Start" to "End" that between them use all the edges in the Product Graph, skipping those that include "Out" nodes that belong to unknown Autonomous Systems

topological node or router location with. By topological node, we refer to the node in the actual network topology or the router that is referred to by a given PGIR node. Recall that every PGIR node has both a state number and a router location associated with it (Fig. 3). Given this structure, one could use the PGIR to look for pairs of nodes with such a property and generate individual paths through each of them. This essentially allows the links and advertisements through both paths to be available. After setting this up, we want to ensure that the preferred path or node is the one the traffic passes through.

This notion of preference coverage extends beyond a preference between exactly two PGIR nodes with a common topological nodes. There might often be more than two PGIR nodes corresponding to a given topological node, each of which might have an associated preference. An *exhaustive* set of tests would check all the preference relationships. In essence, if $p_1 \text{»} p_2 \text{»} p_3 \cdots \text{»} p_n$, then checks enforcing $p_1 \text{»} p_2$, $p_2 \text{»} p_3$, $p_3 \text{»} p_4$ and so on until $p_{n-1} \text{»} p_n$ are considered exhaustive since the rest of the preference relationships like $p_1 \text{»} p_3$ and $p_2 \text{»} p_4$ are implied by the intermediary preference relationships.

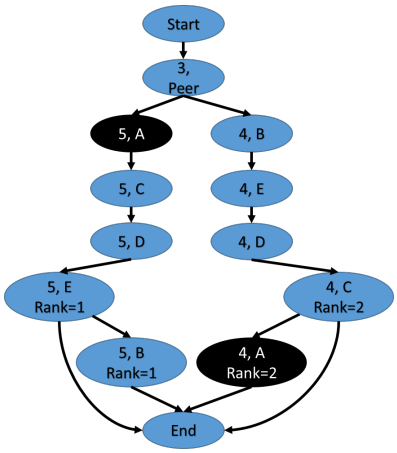
To put these tests in context, we consider the same subgraph PGIR as in Fig. 5. In this subgraph, routers A through E are shared between two PGIR nodes each. The preference coverage tests for

them are shown in Fig. 8 and are indicated by the broken lines. For instance, in Fig. 8b, we have two paths that the traffic towards the Peer (in the opposite direction as the PGIR and advertisements that control the traffic path). Between them, the path on the left has a higher rank and hence, is more preferred over the path on the right. When originating at E, this traffic should take the path $E \rightarrow D \rightarrow C \rightarrow A \rightarrow Peer$ as opposed to $E \rightarrow B \rightarrow Peer$. We cannot expect traffic starting at A to not reach *Peer* just because A is the origin router on the less preferred path. Since the preferred path has a link between A and *Peer*, traffic would indeed reach *Peer* in one hop. Hence, in our tests, we only check the former and avoid the latter un-reachability test since that depends on the context. Similar tests are shown for the shared router B in Fig. 8d. Any of these two tests could account for routers C, D and E since they are all involved in both sets of paths.

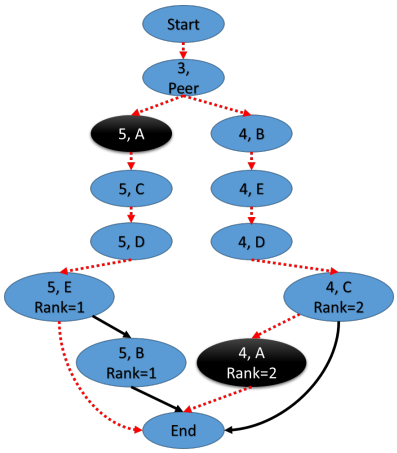
This test is also carried out similarly to the Link Coverage tests described in §4.1. Tests are generated for a sequence of PGIR nodes that share a given topological node at a time for a given PGIR before proceeding onto the next topological node. Tests are only generated for adjacent nodes in the particular *node sequence* of PGIR nodes that share a router once the sequence is ordered by preference. However, what is crucial here is that care needs to be taken to ensure that the path for a given pair of adjacent nodes in a given node sequence doesn't inadvertently turn on links or advertisements for a more preferred node (in the node sequence for the same router) that is not involved in the current test. If this happens, the path through the more preferred node will be used by the traffic and the test will fail. Further, routers on the "less" preferred path might still be able to reach the destination prefix via a subset of the more preferred path if some links on the less preferred path connect to routers on the preferred path allowing the preferred advertisements to flow through. We account for some of these possibilities in §5.

5. Test Generation

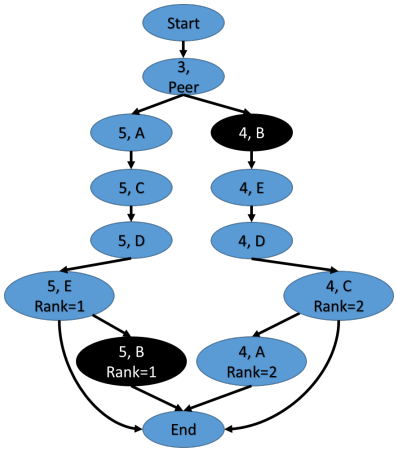
As discussed in §3.2, Propane's compiler reduces the policy specification down to an intermediate representation that is combined with the provided network topology to generate a Product Graph



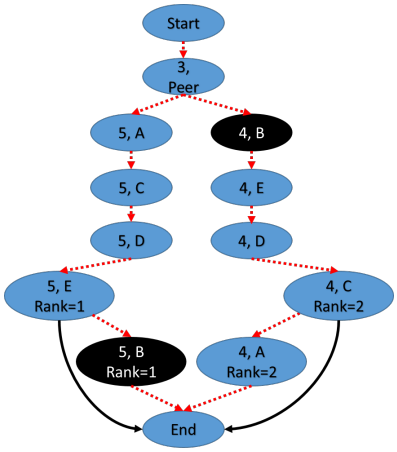
(a) Base PGIR where two nodes share router A



(b) Paths through the two A nodes for preference testing



(c) Base PGIR where two nodes share router B



(d) Paths through the two B nodes for preference testing

Figure 8: Test cases for preference coverage on the PGIR from Fig. 5 indicated by the red broken lines. The tests for routers A and B both encompass the tests for the shared locations between routers E,C and D also. These tests between them are exhaustive for all the PGIR nodes that share topological nodes. Note that there could be other path combinations that could also be exhaustive and these are just two examples

Intermediate Representation (PGIR) [6]. This reduction only involves translating the specified policy using regular expressions and direct substitution, which are assumed to be correct. Hence, we use the PGIR as the input format for the specification to our test generation system as shown in Fig. 4. Each edge in the product graph represents a valid advertisement (with a particular prior advertisement history) on a single given link in the actual network. Hence, exhaustively finding paths through the PGIR translates to finding exhaustive paths through the original topology except

for paths involving external and unknown ASs which cannot be controlled for. We discuss the notions of "exhaustiveness" in §4. In this section, we discuss the specifics of how we generate tests to meet those notions via encoding it as a SAT problem. §5.1 discusses the motivation for constructing SAT instances while §5.2 and §5.3 provide the SAT instances associated with the two test generation problems respectively.

5.1. Why SAT/SMT based solutions?

Given that the goal here is to generate a series of exhaustive paths, a very natural solution would be to perform a depth first search starting at the source vertex. A straw-man solution would include performing continuous searches through the graph keeping track of the new edges covered in the path from a particular iteration. This process would continue until all edges are covered, for the link coverage test generation (§4.1). There are a number of problems associated with this approach. Firstly, there is no way to directly guarantee that the depth first search would use new edges or a different path across iterations. We cannot remove the covered edges from an earlier path from the graph because that might break the connectivity of the graph. We could use a heuristic to prefer the "uncovered" edges, however this might not still guarantee that we find converge towards finding exhaustive paths eventually.

Furthermore, with the preference coverage tests, the number of constraints are numerous. We need a path that passes through a specific pair of nodes within the graph without permitting paths through other conflicting nodes. The exact reason for this is elaborated in §5.3. This means that a simple path that is obtained from a breadth-first or depth-first search might not be sufficient. The complex nature of the rules imposed on the paths that need to be generated for preference coverage almost immediately yield themselves to a SAT solving approach. Over and above this, given the body of work in this space ([7], [23]), there have been a lot of advances in this regard that can be directly exploited to optimize the test generation problem if we map the test generation as a SAT instance. Most of the constraints can be solved using a SAT solver, except that the prevention of

cycles in the path generation problem is overcome by assigning a random integer to each node on the path. At this point, we need Satisfiability-Modulo Theory ([5], [11]) to solve the constraints. We elaborate on the constraints and the intuition behind them in §5.2 and §5.3.

5.2. Link Coverage:

Given our definition of Link Coverage in §4.1, our goal is to generate a unique set of paths through the PGIR that between them cover all valid edges or advertisements in the PGIR. This all-paths problem can be reduced to the SAT problem which can then be run through a SAT solver. We use Z3 [10] (§3.3) in particular, to get one satisfying test path at a given time. A given path maps to a certain combination of links and particular advertisements being received between routers on those links. This also implies that there is a complete sequence of advertisements from the originating prefix router all the way to the router attempting to contact it. Once a given path is generated, we would like to generate more paths or advertisement sequences that are different from the ones generated so far, in that they use atleast one advertisement or link that hasn't already been utilized. Hence, we input this need for a "new" edge as an additional incremental clause to the SAT problem across iterations to ensure that paths aren't repeated. We continue until all the edges of the PGIR have been exhausted or the SAT instance is no longer solvable. At this point, the remaining advertisements (PGIR edges) cannot be part of any advertisement paths.

Below are the key elements of the formulation of the SAT instance for the Link Coverage definition.

1. **Variable Definitions**: We define a variable per node and one per edge in the product graph. Node variables are denoted using n. Edge variables are denoted using $e = (a, b)$ where a and b are the node variables associated with the endpoints of edge e. If a given node or edge is present in the current path, the variables corresponding to them evaluate to $true$
2. **Connectivity Constraints:** This is responsible for obtaining a path starting at the source and ending at the target. This means that the source and the target nodes must themselves be present

in the path. Following this, if a certain node is present somewhere inside a path (except for source and target), there should be exactly one edge going out of it and atleast one edge coming into it. Further if an edge is present in the path, both its end nodes should be present in the path too. This ensures a continuous alternating sequence of nodes and edges from source to destination, which forms a path.

3. **Unknown Node Constraints**: If a given node belongs to an unknown router, effectively connected to a known router that is external, such a node should not be part of a test path generated. Consequently, we identify these nodes and negate their associated SAT variables to ensure that they are set to false or not present in any valid solution.
4. **Topological Loopfree Constraints:** These constraints prevent multiple occurrences of the same topological node or router in a given path. This is important to ensure that we don't have two different advertisement sequences being differently processed leading to different export rules at the same router. Each router can only export one advertisement regardless of how many it receives and in the absence of such a constraint, the generated paths might violate this rule. This is also why there is exactly one outgoing edge from every router in the generated test path.
5. **No-cycle Constraints:** The loopfree constraint prevents multiple occurrences of the same topological node in the same path, however there could still be cycles present in the generated path even if every topological node occurs only once. A simple example would be a bi-directional link between nodes n_1 and n_2 that refer to different topological locations where both the nodes as well as the two edges between them are set to true. This is possible since the PGIR is actually bidirectional, but represented unilaterally here for sake of simplicity. This particular situation satisfies all the above constraints, yet denotes a cycle within the context of BGP and general graph definitions. BGP will reject such a path due to the presence of cycles. To eliminate this, we introduce a new clause that associates an integer value with every node. The source node is given the value 0. If any edge is present in the graph, a constraint is imposed on it that ensures that its start vertex should have one less than the integer value associated with its destination

vertex. This resolves the bidirectional link problem, among other cycle issues, because both n_1 and n_2 cannot have one more than the other one.

6. **New Edges Constraint:** If the PGIR edges yet to be covered by the tests are tracked in a separate set E', the edges in the current output path p are first removed from E' if present. Following this, a new constraint is added based on the current status of E' to ensure that atleast one edge from E' is covered in the next solution to the path generation SAT instance. This helps us eventually reach full coverage.

Each of these constraints are summarized (parallel to their definitions above) in concise mathematical notation below and these are repeatedly solved until the SAT instance returns *Unsatisfiable* or the edges of the current PGIR have been exhausted before moving onto the next PGIR and SAT instance.

1. Let $var(e) = true \iff$ edge e is in the current path p

   Let $var(n) = true \iff$ node n is in the current path p
2. s and t denote the source and target vertices respectively

   $var(s) = true, var(t) = true$

   $var(e) = true; e = (a,b) \iff var(a) \wedge var(b) = true$

   $var(n) = true \iff$ atleast one incoming edge e to n satisfies $var(e) = true$

   $var(n) = true \implies$ exactly one outgoing edge e from n satisfies $var(e) = true$
3. n = "Out" $\longrightarrow var(n) = false$
4. All nodes A_i with $1 \leq i \leq k$ in the PGIR refer to same topological node or router A

   Utmost one amongst $var(A_i)$ is true where $1 \leq i \leq k$
5. $int(n)$ is the integer value associated with node n

   $var(e) = true; e = (a,b) \iff int(a) + 1 = int(b)$
6. E' is the set of edges that haven't yet been covered

   $\bigvee_{e \in E'} var(e) = true$

5.3. Preference Coverage:

Given our definition of preference coverage in §4.2, our goal is to enable a pair of competing traffic paths via a corresponding pair of advertisement paths through the PGIR (§3.2) without enabling any other more preferred paths. In particular, this involves identifying pairs of PGIR nodes that refer to the same topological location or router and generate paths through both of them. This translates to individual routers receiving two competing sets of advertisements that they need to choose from. If correctly configured, they should only forward out the preferred advertisement, consequently causing the traffic to follow this "more" preferred path (in the opposite direction to the advertisement path). As mentioned before, it is likely that there might be other nodes in the graph that share the same topological location or router with this pair in context, some of which might have a higher preference associated with them. If one of the paths through those more preferred nodes (lets define these as adversarial paths) “inadvertently” becomes available because the path through the pair of nodes we are generating tests for employs the same links, this adversarial path might become even more preferred. The traffic will choose the adversarial path as opposed to the expected path between the pair of nodes one was constructing the test for. It is important to avoid this by setting up separate constraints to prevent such a situation. This seems to yield itself naturally to SAT and SMT based solutions over a naive graph search algorithm due to the nature of the constraints on the path.

To generate a particular pair of paths through the pair of nodes in the PGIR, we first identify the pair of nodes on each of the paths involved in the preference relationship. We then supply these nodes themselves as individual constraints to ensure that the paths generated indeed pass through the nodes in context. Similarly, for the other nodes that might potentially share topological location, a path through those nodes (adversarial paths) will necessarily pass through those competing nodes as well. Ofcourse the links on these competing paths, will only be turned on if the paths on the main two nodes use those links. This means that we have additional implication constraints for this purpose between the desired paths and the adversarial paths. For the nodes involved in the

adversarial path, this will be followed by a negation to prevent such paths from becoming available. We encode all of these requirements in a SAT instance and run it through a SAT solver to obtain a test for a given pair of nodes in the PGIR in the preference relationship. We continue this until all pairs in the preference relationship are exhausted for a given topological node or router and then repeat it for other topological nodes/routers. In order to avoid redundancy, we only perform this procedure for paths from adjacent places in the preference relationship, once we have sorted by preference all the PGIR nodes referring to the same router. That is, if $A_1 » A_2 » A_3$, we construct one test to ensure $A_1 » A_2$ and another to ensure $A_2 » A_3$, but not one for $A_1 » A_3$, since the last one is implied if the first two preference relationships are indeed true .

Below are the key elements of the formulation of the SAT instance for the Preference Coverage definition when generating tests for a given pair of nodes that share a topological location or router, say A. Let's assume that a total of k PGIR nodes A_i where $1 \leq i \leq k$ refer to the same topological location A.

1. **Variable Definitions**: We define a variable per node and one per edge in the product graph. Further, for every node in the PGIR that refers to the same topological location A, we duplicate these edge and node variables to compute paths through each of those PGIR nodes. In essence, we have k sets, in total, of variables for all the edges and nodes in the PGIR. The i^{th} set is meant for a path through the node A_i involved in the precedence relationship. Node variables corresponding to the path for the i^{th} node are denoted using (n,i) while edge variables are denoted using (e,i) and $e = (a,b)$ where a and b are the node variables associated with the endpoints of edge e. If a given node or edge is present in the current path associated with a given node A_i, the variables corresponding to them evaluate to $true$
2. **Connectivity Constraints:** This is meant to obtain a path starting at the source and ending at the target through the given node A_i. This means that the source and the target nodes must themselves be present in the path. The given node A_i that we are generating a test for should also be present in the path. The rest of the connectivity constraints are similar to the corresponding

ones in link coverage except that we duplicate them across all the k paths to be generated. This ensures a continuous alternating sequence of nodes and edges from source to destination that passes through the node A_i in context, which forms a path in all k cases.

3. **Unknown Node Constraints**: If a given node in any of the k paths for the k nodes that share the router A belongs to an unknown router, effectively connected to a known router that is external, such a node should not be part of a test path generated. This is again similar to the corresponding constraint in link coverage.
4. **Topological Loopfree Constraints:** This is mainly to prevent multiple occurrences of the same topological node or router in a given path. This is important to ensure that we don't have two different advertisement sequences being differently exported at the same router within a given path through A_i from source to destination. Each router can only export one advertisement regardless of how many it receives. In essence, the i^{th} path through A_i cannot have two occurrences of any topological router because then this path no longer pertains to just one of them and also reflects two advertisements being exported which is clearly not feasible. This constraint also isolates the preference testing to happen only between the two nodes in context when issuing a particular test. It prevents other nodes that share a different topological location (other than A) to be doubly available within a given path, because then it would be unclear which preference is being tested.
5. **No-cycle Constraint**: The loopfree constraint prevents multiple occurrences of the same topological node in the same path, however there could still be cycles present in the generated path even if every topological node occurs only once.This is similar to the same constraint in the link coverage except that it is duplicated across all the k paths to be generated. The source node is given the value 0. If any edge is present in the graph, a constraint is imposed on it that ensures that its start vertex should have one less than the integer value associated with its destination vertex.
6. **Implication Constraints** The biggest concern with generating the tests for the preference coverage tests is that we are no longer operating with a single path of advertisements and links

but rather two separate paths which could share routers and links between routers even. This means that if a certain link or advertisement is active on one path, it is active on the others too since topologically they refer to the same link. So, though we formulate the path finding problem as k unique problems for the k PGIR nodes sharing the topological router A, if a certain link is up on the i^{th} path, it is up on all other paths. Consequently, we add implication constraints across the k problems wherein edges between the same topological routers as end points are either all true or all false (across all k instances) as long as the edges' endpoints share state with the node A_j that is to be covered in that j^{th} instance of the problem. The latter clause is necessary to ensure that the shared topological link between the k problems matches up with that particular path which is a solution path for the j^{th} instance and not other random paths that don't go through A_j.

7. **Preference constraints**: Once we have all the above constraints set up as individual instances associated with the k nodes sharing the router A, we consider the two nodes in context, say A_i and A_j and add a constraint that would satisfy the above i^{th} and j^{th} instances, but not satisfy any others amongst the $k-2$ other instances.

Each of these constraints are summarized (parallel to their definitions above) in concise mathematical notation below and these are repeatedly solved to exhaust the PGIR node pairs that share a topological location (A in the formulation below).

1. For the i^{th} path involving the i^{th} node or node A_i in the PGIR that shares router A

   Let $var(e,i) = true \iff$ edge e is in the current path that p_i

   Let $var(n,i) = true \iff$ node n is in the current path p_i

2. s and t denote the source and target vertices respectively

   $var(A_i,i) = true$

   $var(s,i) = true, var(t,i) = true$

   $var(e,i) = true; e = (a,b) \iff var(a,i) \wedge var(b,i) = true$

$var(n,i) = true \iff$ atleast one incoming edge e to n satisfies $var(e,i) = true$

$var(n,i) = true \implies$ exactly one outgoing edge e from n satisfies $var(e,i) = true$

3. n = "Out" $\longrightarrow var(n,i) = false$
4. All nodes A_j with $1 \leq j \leq k$ refer to same topological node A

   Utmost one amongst $var(A_j,i)$ is true where $1 \leq j \leq k$
5. $int(n,i)$ is the integer value associated with node n in problem i

   $var(e,i) = true; e = (a,b) \iff int(a,i) + 1 = int(b,i)$
6. For $j = 1$ through k: $topo(x)$ refers to the topological location/link of a node/edge x and $st(x)$ refers to the state of the PGIR node x

   $topo(A_i) = topo(A_j) = A$

   Let $e = (a,b)$; $st(a) = st(A_i)$; $e' = (c,d)$; $st(A_j) = st(c)$; $topo(e) = topo(e')$ Then, $var(e,i) = true; \iff var(e',j) = true$
7. $N = \{1 \cdots k\} \backslash \{i,j\}$ and P_i is the SAT instance for a path through A_i

   $P_i \wedge P_j \wedge (\bigwedge_{x \in N} \neg P_x) = true$

6. BGP Simulation

Assuming we have generated the test cases by solving the SAT instances constructed using the encoding in §5, the next step would be to run these tests under the specific network environments that the tests correspond to. Each one of them is simulated as an individual C-BGP [25] session. This means that every router in the test case needs to be mapped to its rules in C-BGP format. Every test case from §5 is reported as a set of edges in the test path and the expected path maps to the policy for a particular prefix or predicate. As described in Fig. 4, the PGIR for a given prefix or predicate is fed through a SAT solver to get the test paths, which are combined with the topology and the abstract BGP configurations to produce the C-BGP input files, one per test case associated with a given prefix or predicate. The test path is involved in setting up the topology, BGP sessions

as well as in deciding which routers are featured in the router configuration rules in C-BGP. On the other hand, the Abstract BGP configurations alone determine the precise rules per router. The verification procedure uses both the test path generated as well as the path employed by the traffic in the simulation to detect any discrepancies. Each of these steps is detailed below.

Topology Setup: The topology structure associated with the PGIR has a set of nodes with edges between them. These nodes have an AS number and a node type associated with them. Every router in Propane is assigned its own AS number and this number uniquely identifies the router. However, in C-BGP, while setting up the topology, we also need a unique IP address associated with every router. Consequently, we assign a random IP address to every router and maintain a map between the router AS number and the IP address for all future references. We add the nodes first to the C-BGP input files using the "net add node" command. We further explicitly add links to parallel the edges from the topology using the "net add link" command. This sets up the basic topology of the network to be tested.

BGP Link Setup: Following this, we need to set up individual BGP links between the router that are used in the test path. For this, we process the test path to identify the neighbors of every given router and then set up links between them. For instance, if there is a link between router 0.0.0.1 and router 0.0.0.2, we need to set up two sets of link, one for each direction. In other words, we need to explicitly add a route towards $0.0.0.2/32$ in the routing table of the router 0.0.0.1 and vice-versa. This is done using the "net node route add" command in C-BGP. The weight associated with every link is uniformly 1 so that every valid link is equally preferable.

Abstract BGP to C-BGP: In order to generate the precise export and import rules between a given router and its peer, we process the router configurations of the given router in the form of abstract BGP on a peer to peer basis. Once we identify a given router and its peer from the test path, we first "turn up" that link. Further, if a router owns a certain prefix, we add that as a separate instruction via the "network add" command so that it advertises that prefix. We also add a rule for that particular router to accept all traffic that matches that prefix regardless of which peer it came from. If this

router happens to be an external router that is at the source end of the advertisement path generated, we still explicitly use the "network add" command so that it simulates traffic coming from an external peer even though as a network operator, we have no control over external routers.

In parallel, we look at the router configurations for a given router and process each one of them as a series of predicates with associated actions for them. This action might be a deny or accept on import that is applicable to all peers who may have sent advertisements concerning the predicate in context. Alternatively, the rule might be to import advertisements matching that prefix only as long as it is from a subset of the peers that meet a certain criteria. The import filter often involves assigning the advertisement a local preference too. This match criteria on the peer aspect can be specified either as a wild-card (applicable to all peers), or might be applicable only to internal or external peers or merely a given single router. The match can also be on a community tag or a regular expression that the advertisement path should match. Depending on what peers the match is applicable to, we add the C-BGP match-action rules to the corresponding peer's import list under the "filter in" rules such that it would import any advertisement matching it and assign it the associated local preference, within its action body. This local preference is then used to determine which advertisement is preferentially processed in the event of multiple advertisements at the router that can be accepted by the router.

After the advertisement has been imported, the router needs to decide what advertisement to send out and to which neighbors. This needs a series of export rules in the router. Each import rule, as described above, usually has as export rule associated with it. In order to make sure the right export rule is associated with the import rule, in C-BGP, we add a new community tag (a random number in the 10000 range) that is typically unused in the policy itself upon import. On export, we match on this community tag and then add an associated action depending on the exact nature of the export rule, as specified in the Abstract BGP. The export action either involves setting a new community value, a MED value or to prepend a certain AS a certain number of times, all of which can be done using C-BGP "actions" for the "filter out" (the export filter) rule. Depending on the

precise peers (all, internal, external or a specific router) this needs to be exported out to, these set of "match-action" rules are added to the "filter out" portion of the relevant peers of the router.

Verification Procedure: In order to verify that the actual path taken by the traffic corresponds to the expected path, we keep track of the expected path at the AS level by parsing through the set of edges that are contained in the expected path. This is equivalent to tracking the per-hop path of the traffic since every router has its own AS as per the Propane implementation [6]. Further, we make note of the last router on the path, which is where the "traceroute" would need to start from. The path on the PGIR is the path of advertisement flow while traffic flows in the opposite direction. So, in order to verify that the traffic flow corresponds to the advertisement sequence in the test path, we start a traceroute on the last node in the advertisement path towards the originating prefix that is owned by the first node in the advertisement path. We expect traceroute to succeed and the output to be the list of AS's in order between the destination node and the source node. We verify this expected output against the output of the traceroute command after the BGP simulation is complete.

7. Evaluation

We now evaluate the test infrastructure in terms of its exhaustiveness and the overheads involved in generating tests. We briefly describe the setup for the evaluations we ran as well as the metrics of interest for overheads and exhaustiveness in §7.1. Following this, we describe individual results for the link coverage tests in §7.2 and the preference coverage tests in §7.3 respectively.

7.1. Methodology:

Setup: We run all our evaluations on data center networks with the routers arranged in fat-tree topology. This is similar to a portion of the evaluations run for Propane [6]. A fat-tree topology essentially looks like the network described in Fig. 9 wherein the routers are arranged in a hierarchical manner. The servers themselves are clustered into units called pods. For every k pods present in the

network, the total number of routers in the same network is $1.25k^2$. Hence, as the number of pods increases, the number of routers grows quadratically. Additionally, these routers in the evaluation benchmarks are connected to a certain number of external peers that are outside the AS of interest.

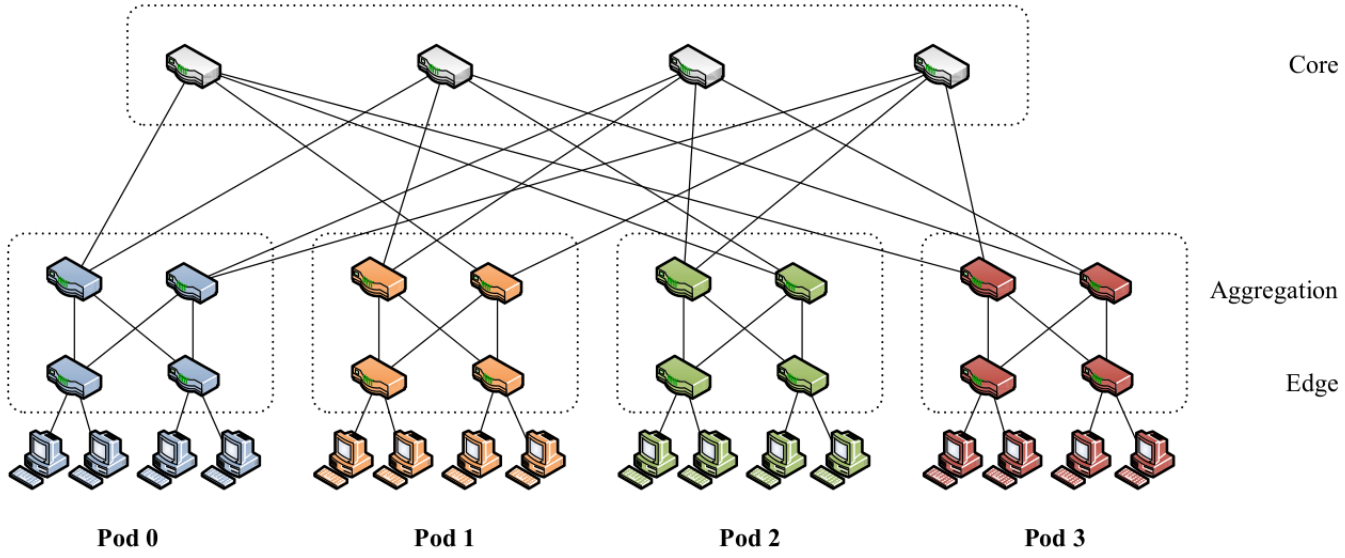


Figure 9: Routers in a datacenter network arranged in fat-tree topology. The topology follows a hierarchical structure with the servers ultimately being arranged in units called pods. The number of routers is $1.25k^2$ where k is the number of pods. [4]

The propane policy for these datacenter networks includes defining certain private prefixes that are meant to be kept internal as well as "bogon" or unallocated prefixes whose traffic shouldn't be observed in the network. There are more specific policies to restrict prefixes to certain parts of the network. Traffic is instructed to enter or exit the AS only at specific routers. This is achieved, in practice, by attaching communities to advertisements concerning these prefixes and enforcing rules that would prevent traffic from exiting the local parts of the network that they are restricted to. Certain prefixes are also restricted to particular tiers of the topology. In addition to these policies, the essential rules of preventing no-transit traffic between two other external ASs are also added. Lastly, there is a single preference to establish that all traffic should exit via one particular peer over the other.

To run these evaluations, we vary the number of pods in the datacenter network, consequently affecting the size of the network itself, in terms of the number of routers. In order to vary the extent of coverage for the tests, we randomly select a portion (proportional to the coverage amount) of the network to test. In the link coverage test, this involves finding a random set of edges of the PGIR until the size of the set of edges to cover is as much as the coverage amount demands. In the preference coverage tests, this translates to finding a set (with size proportional to the coverage extent) of random pairs of nodes (adjacent in the preference list) that share a topological location. Furthermore, since there is a unique PGIR for every prefix or predicate, the coverage for the tests for a given network configuration is the average coverage across all the tests for all its PGIRs. While varying the tests along the size of the network and coverage extent, we measure the metrics mentioned below.

Metrics: In order to measure the overhead associated with the test generation, both link coverage and preference coverage, we first isolate the time spent in each portion of the test stack Fig. 4. The first portion involves generating the test paths from the PGIR via Z3. This is measured as the **test generation time**. The second portion involves translating these test cases into C-BGP input using the topology, abstract BGP as well as the test paths themselves. This is measured as the **test printing time** since this involves printing individual C-BGP input files. The last portion involves simulating these as individual BGP sessions within C-BGP. This is measured as the **test running time**. We also measure the **memory** needed to store the test files. Following this, we perform the verification of the expected path against the obtained traffic path from the simulation. We do not measure this verification time since it is a single "diff" command that takes a very small amount of time. But, we perform this validation to ensure that times are obtained only when the tests pass. Lastly, to measure the exhaustiveness of the test cases, we also record the number of tests generated or individual C-BGP input files themselves and what percentage coverage of the graph they correspond to.

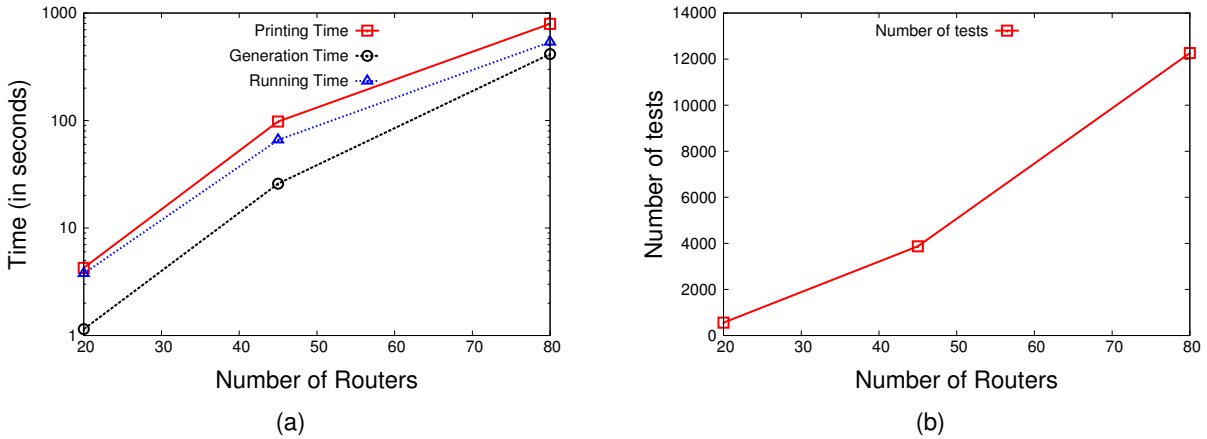


Figure 10: (a) Overhead of performing link coverage tests on datacenter networks with fat-tree topology. Time taken to perform tests scales at over 10x with a step increase in number of pods. (b) Exhaustiveness of link coverage tests on the same networks. Tests exhaust over 97 percent of the Product Graph and grow at about 5x with a step increase in number of pods.

7.2. Link Coverage:

For the link coverage tests, we ran a series of evaluations varying the size of the network as well as the coverage extent. The broad results of these evaluations are presented in the next few paragraphs.

How exhaustive are the tests? When we run the link coverage algorithm using our system to exhaust all edges in the PGIR generated from the propane policy on the datacenter fat-tree topology as described above, the number of tests produced for smaller networks is quite large. For instance, as shown in Fig. 10b, our test infrastructure produces 558 tests (corresponding to around 98% coverage) on a network with only 20 routers, with this scaling to 3874 (99% coverage) at the immediate next datacenter network size. These are quite indicative of exhaustiveness, with all of them covering over 97% of the PGIR's associated with the prefixes on average. Our infrastructure doesn't achieve absolute 100% coverage because it might not be possible to generate a path through some edges or they might be edges involving unknown ASs or special "Out" nodes.

How expensive is testing? While these tests are quite exhaustive, the costs associated with testing is quite high. As shown in Fig. 10a which holds a log-scale on the y-axis, the time taken to generate, print as well as run tests, scales at about 10x for every step increase in network size. More

specifically, the whole testing cycle takes only about 10s for a fat-tree with 4 pods or 20 routers. But, it is well close to 200s for the next fat-tree with 6 pods or 45 routers. This further grows to a little less than 1800s for the fat-tree with 8 pods or 80 routers and takes well over a few hours at further sizes. Furthermore, since we are printing the tests as individual files, the memory consumed by these files is also quite high as shown in Table. 1. We need close to 8GB of storage space to test a network with just 80 routers.

Number of Pods	Number of Routers	Number of Tests	Memory Consumed by Tests (in MB)
4	20	558	47
6	45	3874	984
8	80	12263	7700

Table 1: Memory consumed by Link Coverage tests for different fat-tree topology datacenters. Memory grows at almost an order of magnitude for every step increase in network size

Exhaustiveness vs Overhead Tradeoff: From Fig. 10 and Table. 1, it is visible that there is a tradeoff between how exhaustive the tests are and how expensive it is to generate and run them. So, an alternative approach to this would be to observe how much time it takes to generate tests for a given level of exhaustiveness or test coverage. In the context of link coverage, as previously defined, this exhaustiveness is defined as the fraction of the PGIR edges covered by the tests. For instance, we vary this percentage of PGIR edges covered from 10 to 100% of the edges of the graph generated running from the described Propane policy on a fat-tree topology with 8 pods or 80 routers. The resulting overheads in terms of time taken to generate and run tests as well as number of test files are shown in Fig. 11. The number of tests produced sees a monotonic increase as seen in Fig. 11b. In general, the overhead increases as the coverage amount is increased as seen in Fig. 11a. The test generation, printing and running time all also follow an upward trend. However, we see a small dip when going from 70% to 80%. The reason for this could be just the randomness involved in generating the edges to be covered, since we repeatedly generate "random" edges until we hit the target coverage. If the random process was producing the same edge repeatedly, it would take much

longer to generate unique edges to ensure the size of the set is as needed. This effect can be also mitigated by running the same multiple times to average results.

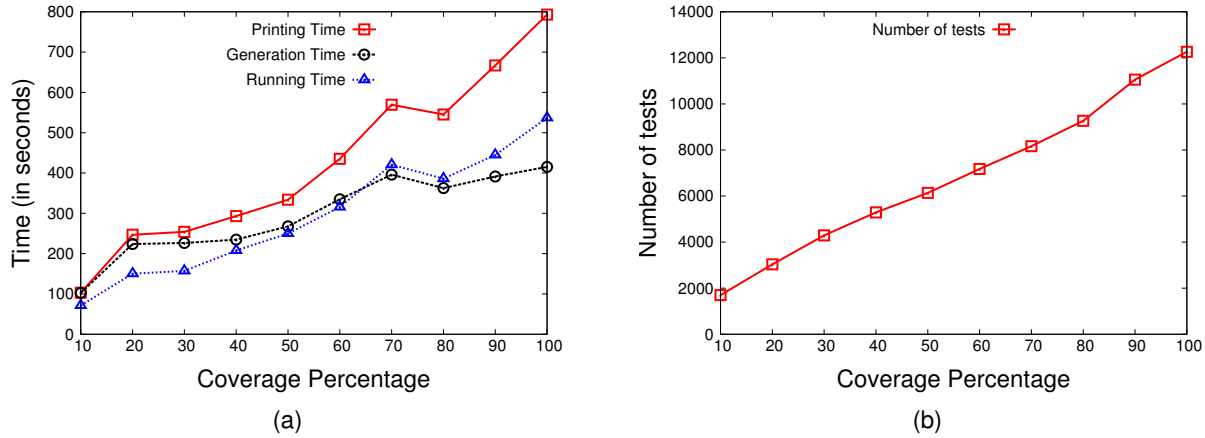


Figure 11: Tradeoff between the testing overhead and the exhaustiveness of the test cases for the link coverage tests when Propane is run on a fat-tree topology with 8 pods or 80 routers. Generally, overhead increases as the test cover a larger portion of the PGIR or become more exhaustive. However, the increase in overhead is lesser when moving between lower coverage amounts than it is at higher coverage amounts.

The second noteworthy effect in Fig. 11a is that the increase in test generation, printing and running time is steeper at the right end of the graph or for higher coverage amounts than for lower coverage amounts. So, the increase in time in going from 20% to say 30% is significantly lesser than going from 80% to 90%. This is expected since the number of remaining edges in the graph to pick from becomes progressively lower as the coverage amount increases and a number of paths have already been chosen making it harder to find new paths.

Testing within Time Bounds: Another alternative that we considered instead of fixing the exhaustiveness at a certain level was to cap the amount of time spent on exhausting the PGIR's edges across paths. In other words, the test generation times out on a given PGIR and we observe how much coverage we are able to achieve within that timeout period. We consider three different timeout amounts, 10s, 60s and 100s and plot the coverage obtained in that time period across increasing sizes of datacenter networks as shown in Fig. 12. Overall, the coverage increases with increasing amounts of time for a network of a given size. At smaller network sizes like 4 pods or 20 routers,

the coverage is unaffected my timeout period since it only takes a little time to hit the maximum coverage of around 98%. For the intermediate sizes like 80 routers (8 pods) or 125 routers (10 pods), there is a visible increase in coverage as time is increased. For instance, in moving from a 10*s* to 60*s* timeout period, coverage improves from 87% to 97% on the topology with 80 routers. Similarly, in moving from 60*s* to 100*s* at 125 routers, the coverage improves from about 41% to 91%. Thus, as expected, for larger networks, we need more time to see a higher amount of coverage or major returns on increasing the timeout period.

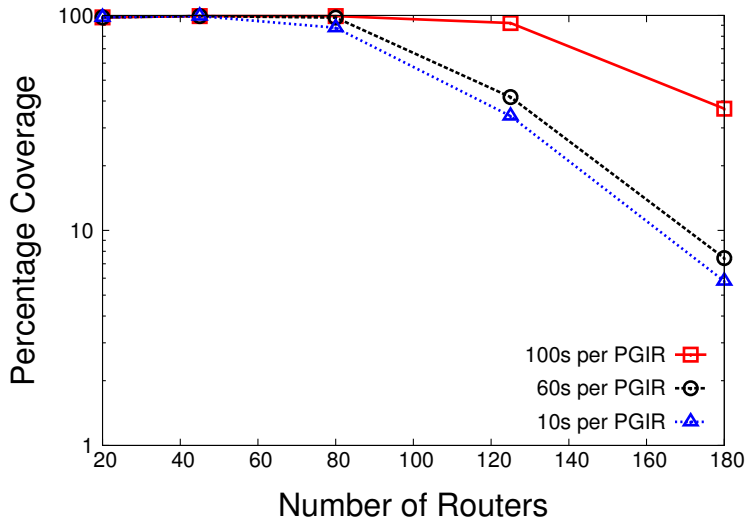


Figure 12: Coverage obtained within a fixed time limit on fat-tree topology datacenter networks as the number of routers is increased when the link coverage tests are run. The smaller networks do not benefit much from the increase in time since their product graphs have already been exhaustively tested with even a 10*s* time limit on each PGIR, while some of the larger networks see a sizeable increase in graph coverage when the time spent per PGIR/prefix is increased.

7.3. Preference Coverage

For the preference coverage tests, similar to the link coverage tests, we ran a series of evaluations varying the size of the network as well as the coverage extent. The broad results of these evaluations are presented in the next few paragraphs.

How exhaustive are the tests? When we run the preference coverage algorithm using our system to exhaust all nodes sharing routers in the PGIR generated from the propane policy on the datacenter

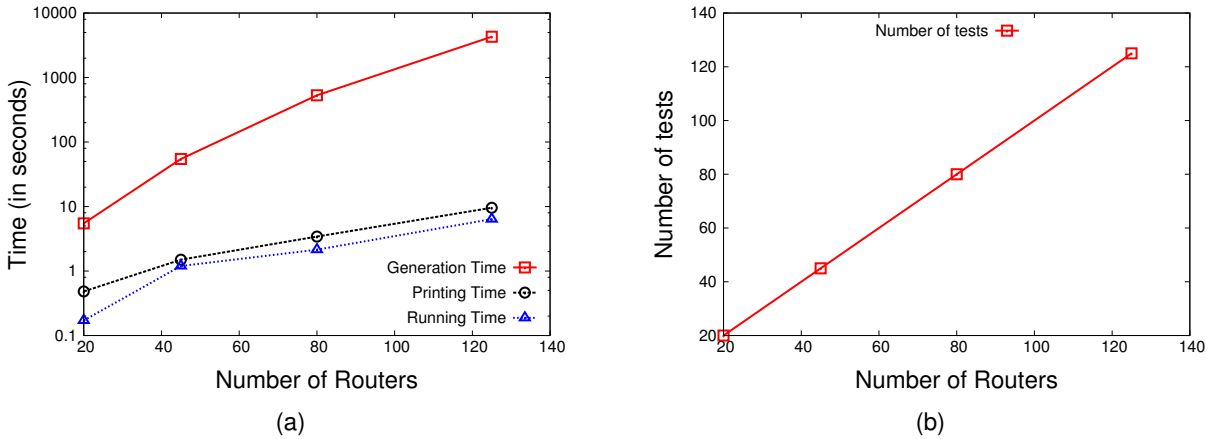


Figure 13: (a) Overhead of performing Preference coverage tests on datacenter networks with fat-tree topology. Time taken to generate tests scales at around $10x$ with a step increase in number of pods, while test printing and running grows at only about $1s$ for every step increase in pod size (b) Exhaustiveness of preference coverage tests on the same networks. The number of tests is exactly equal to the number of routers in the network and this translates to a 100% coverage of the graph since there are as many repeated PGIR nodes as there are routers in the network.

fat-tree topology as described above, the number of tests produced matches the number of routers in the network. This is shown in Fig. 13b. This is because there is exactly one preference relationship in the propane policy that specifies that exiting via one peer is more preferable than another. This means that for every router, when it has a path through both of these peers, it should prefer one over the other. This translates to two possible paths for every router towards the external peers. In other words, there are exactly two PGIR nodes per internal router and one of these nodes has a higher rank than the other. Our test infrastructure produces 1 test per router, thus testing all the PGIR nodes that share routers. This translates to 100% coverage or true "exhaustiveness" in the PGIR preference sense.

How expensive is testing? While these tests are exhaustive, the costs associated with testing continues to be high even though the number of tests aren't as high as the corresponding link coverage case. As shown in Fig. 13a which holds a log-scale on the y-axis, the time taken to generate the tests scales at about $10x$ for every step increase in network size. More specifically, test generation takes only about $5s$ for a fat-tree with 4 pods or 20 routers. But, it is well close to

55*s* for the next fat-tree with 6 pods or 45 routers. This further grows to a little less than 530*s* for the fat-tree with 8 pods or 80 routers and takes well over an hour at the fat-tree with 10 pods or 125 routers. However, compared to the link coverage tests, the test printing and running time are only within a few seconds. This is largely because the number of tests generated in the preference coverage test is smaller by a few orders of magnitude. This is expected given that the number of nodes in the PGIR with shared topological locations is naturally lesser than the number of edges in the PGIR, thus making it easier to exhaust the former than the latter. The memory concerns associated with storing the printed files are also almost non-existent in this scenario.

Exhaustiveness vs Overhead Tradeoff: Given that the time taken to generate fully exhaustive tests for the preference coverage notion is fairly high, a natural question would be to observe if we can save on the overhead by giving up some coverage. In other words, if we fixed the amount of coverage we want or the fraction of the PGIR nodes that share topological locations that we want to test, would be be able to achieve smaller amounts of coverage within a significantly shorter time period? We vary this percentage of such PGIR nodes covered from 10% to 100% of the total number of such nodes in the PGIR generated by running Propane on a fat tree topology with 8 pods or 80 routers. The resulting overheads in terms of time taken to generate and run tests as well as number of test files are shown in Fig. 14. In general, the overheads increase as the coverage amount is increased. The number of tests produced just sees a monotonic exactly linear increase as seen in Fig. 14b. This is because each of those intermediate coverage amounts is exactly achievable by covering that many routers that are recurring between PGIR nodes. For instance at 10% coverage, exactly 8 out of 80 routers have been tested and at 50% coverage, 40 out of 80 routers have been tested.

The test generation, printing and running time all also follow an upward trend as seen in Fig. 14a. However, unlike the case with the link coverage tests where the increase is steeper at higher coverage, the growth is fairly linear with the preference coverage tests across the entire range. Furthermore, the time taken for generating the tests is significantly higher than the time taken to print or run them,

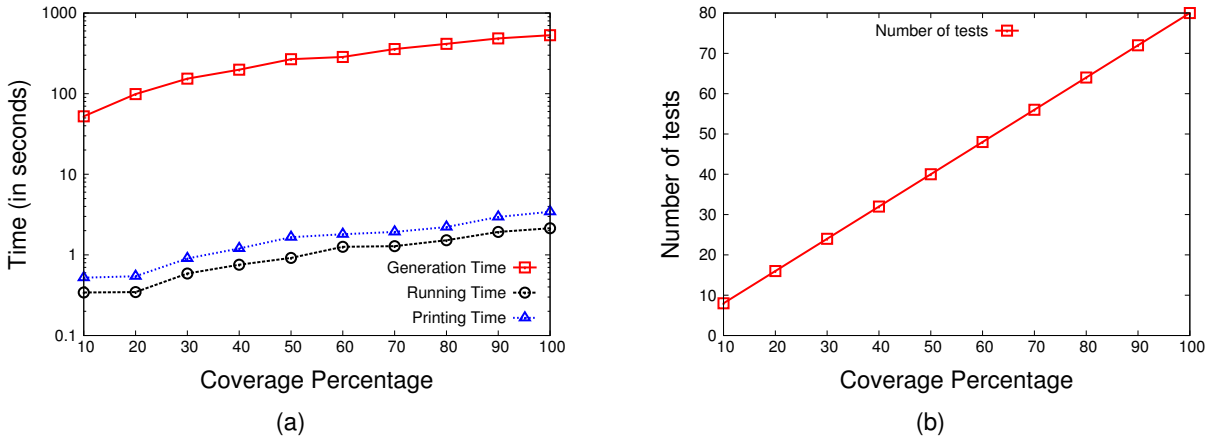


Figure 14: Tradeoff between the testing overhead and the exhaustiveness of the test cases for the preference coverage tests when Propane is run on a fat tree topology with 8 pods or 80 routers. Generally, overhead, in terms of time taken for the testing cycle, increases as the test cover a larger portion of the PGIR or become more exhaustive. This overhead scales fairly linearly with increase in coverage.

as expected. However, the relative increase is similar across all three of them. Thus, unlike the link coverage case, there isn't a clear point or reasonable coverage amount at which the returns from the additional time spent on generating more tests starts to dwindle. There is always a proportional increase in coverage as more time is allocated to the cause or vice-versa.

Testing within Time Bounds: The other alternative that we considered, similar to the link coverage evaluations, was to cap the amount of time spent on exhausting the PGIR nodes that shared topological locations instead of fixing the coverage at a desired amount. In other words, the test generation times out on a PGIR and we observe how much coverage we are able to achieve within that timeout period. We consider three different timeout amounts, 50s, 300s and 1500s and plot the coverage obtained in that time period across increasing sizes of datacenter networks as shown in Fig. 15. Unlike the link coverage test where there are multiple PGIRs on which this is run repeatedly with individual timeouts, in the preference coverage test, this is performed only on on the one PGIR corresponding to the preference policy of one particular external peer over another. Overall, the coverage increases with increasing amounts of time for a network of a given size. At smaller network sizes like 4 pods or 20 routers, the coverage is unaffected my timeout period since it only takes a little time to hit the 100% coverage. For the intermediate sizes like 80 routers (8 pods) or

125 routers (10 pods), there is a visible increase (almost an order of magnitude) in coverage as time is increased by $5-6x$. For instance, in moving from a $50s$ to $300s$ timeout period, coverage improves from 10% to almost 60% on the topology with 80 routers. The coverage does improve at larger network sizes too, but is a little less pronounced for the same increase in timeout period

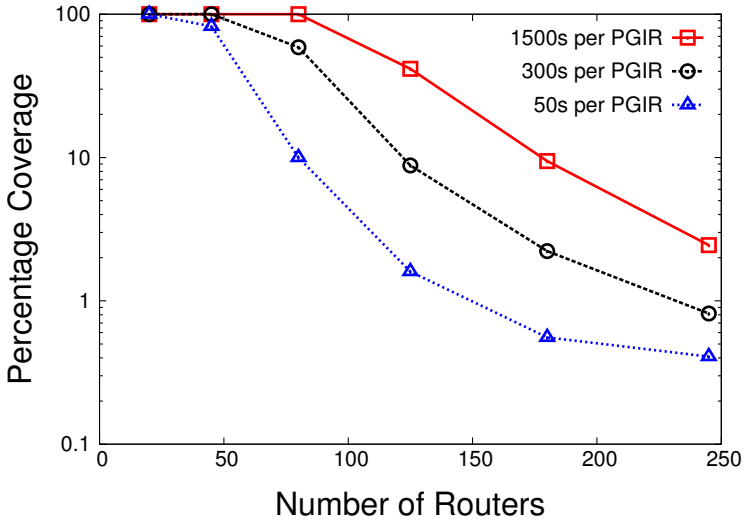


Figure 15: Coverage obtained within a fixed time limit on fat-tree topology datacenter networks as the number of routers is increased when the preference coverage tests are run. The smaller networks do not benefit much from the increase in time since their product graphs have already been exhaustively tested with less than $50s$ time limit on the PGIR, while some of the larger networks see an increase in graph coverage when the time spent per PGIR is increased.

8. Discussion

8.1. Results

In general, the trend with both link coverage and preference coverage tests is that the tests are feasible for a small networks or ASs, such as Internet 2, which have only about 20 routers. In such scenarios, this testing infrastructure can certainly help test and verify that the configurations meet the specifications. However, the evaluations and particularly the test generation times suggest that this approach, as it stands might not be suitable for datacenter networks with a larger number of routers since it doesn't scale well with size.

Despite this, the analysis of the trade-off between the testing overhead and the corresponding coverage achieved shows promise. It suggests that it is easier to get to a reasonable coverage amount (say 60 - 70%) than it is to go from there upwards towards 100%, particularly with the link coverage tests. This means that if operators are okay with less than 100% coverage as long as the portions of the graph tested are generated randomly, this might still be a viable solution. This is the case with the preference coverage tests too, where the time taken for the tests increases linearly as one desires more and more test coverage. Timing out after a certain while is also a reasonable strategy often used within the realm of program verification and systems testing [8] which when employed in this context, still gives a good amount of coverage. Further, we see a proportional improvement in coverage amount as the timeout period is increased.

8.2. Future Work

This work and the nature of results on it leaves room for further exploration and improvements which are described in detail below. Secondly, this is a testing framework currently amenable to one form of policy specifications that has potential for further expansion beyond that single format like Propane[6].

Further Optimizations: One concrete direction is to optimize the SAT and SMT solvers themselves to run faster on the larger test cases. But, assuming this is hard, we could improve the nature of the inputs we supply it and allow a little more leeway in terms of what we expect from the solvers. This could involve pruning the graph for where bugs are most likely to be and only testing those portions. A second choice that would be helpful with the link coverage tests would be towards heuristics that choose paths that maximize the new number of edges covered and minimize reused edges across tests. A concrete way to do this would be to supply the SAT solver with soft and hard constraints. Soft constraints are those that don't need to be met necessarily, but if a solution does meet them, such a solution is preferred over some other solution that doesn't meet them. In the preference coverage case, some tests already have multiple pairs of PGIR nodes that share topological location.

Instead of duplicating these tests on a pair to pair basis, some of these can be reused, thus reducing the total number of tests. There are a number of optimizations even within the Z3 framework that can be further exploited which include setting timeouts for a single assertion, simplification of constraints, reuse of previous work on solving the constraints and so on. Exploiting higher amounts of computing power could also help speedup this process.

The current iteration cycle for the test framework is such that all the tests are generated within the Z3 framework first, stored as sets of the edges on the path, printed out as individual C-BGP files and then run as individual sessions. This implies that all the C-BGP files are stored at once on the system which creates severe memory overheads. If instead the iteration cycle were changed to generate, print and immediately run the test, the test file could be deleted and the memory overhead overcome.

Lastly, it might be possible to eliminate the printing altogether if it was possible to simulate BGP sessions from within Propane itself via a C-BGP API or something similar. This might eliminate some of the intermediary steps speeding up the process of testing altogether.

Features to support: Currently, the test infrastructure only supports matching on a peer criteria or on community tags while Propane allows matching based on regular expressions on the path that the advertisement takes to that router. The reason for this is that the regular expression doesn't map to a C-BGP instruction as neatly as the peer or community tag. This needs to be investigated further and the regular expression thoroughly parsed to translate it into an equivalent C-BGP instruction or set of instructions when constructing rules for advertisements to be imported and assigned a local preference.

Coverage extent: The results in Fig. 10b suggest that even when we aim for 100% coverage of the product graph, we don't quite achieve it because no paths through those other edges exist. There might be other ways to test these edges without needing a complete path between source and destination. Some intermediary nodes might still use a path to the AS owning the prefix for whom we are generating tests. This hasn't been explored within the scope of this work, but certainly poses

potential to improve the coverage within the realm of link coverage test generation.

Further, with preference coverage tests, we do make the assumption that if $p_1 \gg p_2 \gg p_3 \cdots \gg p_n$, then checks enforcing $p_1 \gg p_2$, $p_2 \gg p_3$, $p_3 \gg p_4$ and so on until $p_{n-1} \gg p_n$ are exhaustive since the rest of the preference relationships are implied. With additional computing power, it would be good to verify that the implied preference relationships actually hold. For instance, checking that $p_1 \gg p_3$ is indeed implied by $p_1 \gg p_2$ and $p_2 \gg p_3$.

New notions of coverage: This work currently defined two notions of coverage: one to enforce validity of advertisements and the other to verify the validity of preferences. However, Propane provides numerous other capabilities including keeping private prefixes internal to the AS, not assigning Bogon prefixes to any AS, not permitting transit traffic and so on. These properties could also be tested. Further properties like equivalence of router configurations and multipath consistency can also be investigated in the context of Propane. If this framework were to be used outside the context of Propane, a number of new properties might emerge which can also be investigated.

9. Conclusion

In this paper, we develop a testing framework that generates and simulates an exhaustive set of test network environments to verify that router configurations meet specifications. In particular, we define two notions of coverage: one to test validity of paths in the network and another to test exercise of preferences between paths. The framework in this paper is directed at testing compiler correctness for a particular high level language called Propane, but holds potential for extensions towards any general specification of network policy. Evaluations suggest that such a framework is indeed feasible for networks upto 100 routers but with further optimization, might become more feasible for even larger networks.

Acknowledgments

I thank my advisor, David Walker for having been a great source of guidance on this thesis and for having been patient with me when I struggled to make progress. I thank Aarti Gupta for her invaluable inputs on the framing of the SAT instance. I am grateful to Ryan Beckett who has been instrumental in helping me understand Propane as well as in clearing a number of roadblocks along the course of the project. I'd also like to thank my family and friends for supporting me to great extents both through the course of this thesis as well as in the last four years.

References

[1] https://www.theregister.co.uk/2016/03/01/google_cloud_wobbles_as_workers_patch_wrong_routers/.

[2] http://www.datacenterdynamics.com/content-tracks/servers-storage/microsoft-misconfigured-network-device-led-to-azure-outage/68312.fullarticle.

[3] http://c-bgp.sourceforge.net/.

[4] http://sdn-in-datacenters.blogspot.com/2014/04/literature-survey-portland-design.html.

[5] C. W. Barrett *et al.*, "Satisfiability modulo theories." *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

[6] R. Beckett *et al.*, "Don't mind the gap: Bridging network-wide objectives and device-level configurations," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 328–341.

[7] A. Biere *et al.*, "Conflict-driven clause learning sat solvers," *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pp. 131–153, 2009.

[8] C. Cadar *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[9] M. Caesar *et al.*, "Design and implementation of a routing control platform," in *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 15–28.

[10] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[11] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[12] M. Dugald, 2010, http://www.cnn.com/2010/US/11/17/websites.chinese.servers/.

[13] S. K. Fayaz *et al.*, "Efficient network reachability analysis using a succinct control plane representation."

[14] S. K. Fayaz *et al.*, "Buzz: Testing context-dependent policies in stateful networks," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, 2016, pp. 275–289.

[15] N. Feamster and H. Balakrishnan, "Detecting bgp configuration faults with static analysis," in *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 43–56.

[16] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn," *Queue*, vol. 11, no. 12, p. 20, 2013.

[17] A. Fogel *et al.*, "A general approach to network configuration analysis." in *NSDI*, 2015, pp. 469–483.

[18] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[19] P. Kazemian *et al.*, "Real time network policy checking using header space analysis." in *NSDI*, 2013, pp. 99–111.

[20] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks." in *NSDI*, vol. 12, 2012, pp. 113–126.

[21] A. Khurshid *et al.*, "Veriflow: Verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[22] H. Mai *et al.*, "Debugging the data plane with anteater," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 290–301.

[23] S. Malik and L. Zhang, "Boolean satisfiability from theoretical hardness to practical success," *Communications of the ACM*, vol. 52, no. 8, pp. 76–82, 2009.

[24] S. Narain *et al.*, "Declarative infrastructure configuration synthesis and debugging," *Journal of Network and Systems Management*, vol. 16, no. 3, pp. 235–258, 2008.

[25] B. Quoitin and S. Uhlig, "Modeling the routing of an autonomous system with c-bgp," *IEEE network*, vol. 19, no. 6, pp. 12–19, 2005.

[26] K. Weitz *et al.*, "Scalable verification of border gateway protocol configurations with an smt solver," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2016, pp. 765–780.

[27] G. G. Xie *et al.*, "On static reachability analysis of ip networks," in *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, vol. 3. IEEE, 2005, pp. 2170–2183.

[28] L. Yuan *et al.*, "Fireman: A toolkit for firewall modeling and analysis," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 15–pp.

[29] Z3Prover, https://github.com/Z3Prover/z3/wiki.

[30] H. Zeng *et al.*, "Automatic test packet generation," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 241–252.

[31] H. Zeng *et al.*, “Libra: Divide and conquer to verify forwarding tables in huge networks.” in *NSDI*, vol. 14, 2014, pp. 87–99.