MESSI: Behavioral Testing of BGP Implementations

Rathin Singha and Rajdeep Mondal, *University of California Los Angeles;*
Ryan Beckett, *Microsoft;* Siva Kesava Reddy Kakarla, *Microsoft Research;*
Todd Millstein and George Varghese, *University of California Los Angeles*

https://www.usenix.org/conference/nsdi24/presentation/singha

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the
21st USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

MESSI: Behavioral Testing of BGP Implementations

Rathin Singha[1] Rajdeep Mondal[1]
Ryan Beckett[2] Siva Kesava Reddy Kakarla[3] Todd Millstein[1] George Varghese[1]

[1]*University of California, Los Angeles* [2]*Microsoft* [3]*Microsoft Research*

Abstract

Complex network protocols like the Border Gateway Protocol (BGP) are prone to implementation errors that cause unintended behaviors with potentially global consequences. We introduce an approach and tool called MESSI (Modular Exploration of State and Structure Inclusively) to automatically generate tests for black-box BGP implementations. Our approach is *model-based*, leveraging an executable model of BGP to generate behavioral tests. However, doing so effectively requires addressing new challenges such as the stateful nature of BGP and the need to generate complex structures like regular expressions in route maps. We used MESSI to generate roughly 150K tests that capture different aspects of BGP, such as route-map filtering, the decision process, route aggregation, and dynamics. These tests identified 22 correctness bugs across several widely used open-source BGP implementations (FRR, Quagga, GoBGP, BIRD, Batfish) and one closed-source implementation. Eight of these errors have already been fixed. While our models are BGP-specific our approach is not: thus we expect it can be adapted to test other stateful protocols with complex structures.

1 Introduction

The Border Gateway Protocol (BGP) is the backbone of modern internet routing, as it connects the various autonomous systems to one another [47]. Due to its ability to enforce expressive policies, BGP is also widely used by organizations, for example, to establish reliable communication in data centers and to shape policies within enterprises. Due to its ubiquity and large blast radius, BGP errors can have large-scale effects across the globe. Unfortunately, BGP-related outages are quite common [7, 18, 31, 36, 41, 43, 50]. While there has been significant research on identifying BGP *configuration* errors (*e.g.*, [5, 24, 26, 34, 48]), there has been less work on automatically identifying BGP protocol *implementation* errors, which occur frequently in the various widely used BGP implementations and are the focus of this work [8, 9, 19]. According to a study conducted in [39], 36% of the significant and customer-impacting incidents in Microsoft's network are caused by software implementation bugs.

Our goal is to automatically generate tests for BGP implementations that cover a wide range of BGP behaviors. Doing so is challenging because each test case consists of a triplet of an incoming route announcement, a configuration that indicates how incoming announcements should be treated, and the current BGP state of the router. Traditional test-generation approaches are not sufficient: for example, fuzzing is unlikely to find combinations of these inputs that explore a wide variety of interesting behaviors. Instead, we are inspired by recent work that employs *model-based testing* to find errors in black-box protocol implementations [33]. The basic idea of that approach, called SCALE, is to build an executable model of a protocol and then symbolically execute the model in order to generate tests. The model encodes the intended behavior of the protocol, as specified in RFCs, and the use of symbolic execution ensures that the generated tests cover a wide variety of scenarios represented by different execution paths through the model. The resulting tests can then be executed on multiple black-box protocol implementations, with any differences among them identified as potential errors.

While promising, the SCALE approach targeted DNS nameserver implementations. The BGP setting introduces several new challenges for test generation. First, BGP is *stateful*: the handling of a BGP announcement depends on earlier announcements; by contrast, DNS nameservers are stateless. Handling state dramatically increases the search space for test generation and requires correlating state with other protocol inputs to identify interesting behaviors. Second, DNS configurations consist of a flat set of *records* in a zone file. By contrast, BGP configurations are like little programs, with complex and often hierarchically structured policies defined via *route maps*. Notably, BGP policies depend heavily on regular expressions: but it is infeasible to symbolically generate regular expressions both theoretically (the theory of strings with symbolic regular expressions describes non-regular languages) and in practice. Third, DNS nameserver implementations all accept a standard format for configurations; in contrast, the various BGP implementations like Cisco, Juniper, Quagga, BIRD, and GoBGP have distinct configuration languages that differ in both obvious and subtle ways, making it difficult to perform

differential testing across them.

We present an approach and corresponding tool called MESSI[1] (Modular Exploration of State and Structure Inclusively) to overcome these challenges. First, we handle the complexity of BGP by decomposing it into several modules and generating tests for each module individually. The benefit of this decomposition is that testing each module only requires us to model the portion of BGP inputs, configurations, and states that are relevant to that module, which makes a model-based approach tractable. For example, generating tests for route-map filtering requires us to model the way that a route map processes an input route, but it does not require us to model the state. In contrast, generating tests for the BGP decision process requires us to model router state, but only a specific portion of it — the single best route installed so far to a destination — and does not require us to model complex configurations. In addition to these models, MESSI also includes models for *route aggregation* as well as for *dynamics*, which models router state changes such as updates to a route map and updated route announcements.

Second, we handle test generation in the presence of regular expressions through a hybrid approach. Inspired by enumerative combinatorics [38], we use a form of enumerative testing to generate (not just count) all forms of regular expressions up to a specified size. We then generate positive and negative example strings for each regular expression, doing so in a manner that provides coverage guarantees over the regular expression's finite automaton representation. Finally, our model of route-map filtering treats regular expressions concretely — we supply it with specific regular expressions that we have generated, and the model uses only the given positive and negative example strings when building concrete routes to match against such regular expressions. Finally, to handle the diversity of BGP configuration languages, we have created our own intermediate representation for configurations along with a lightweight shim that translates from this representation to the various vendor-specific languages.

Our tool MESSI generated roughly 150K test cases for the BGP decision process, Route filtering, Aggregation, and Dynamics within two days. From 150K tests, we found around 1500 failed test cases across all implementations. We added tags to each test case based on the path they traverse in the code logic and thus grouped equivalent failed cases into buckets. Although the BGP implementations we tested are well maintained and in use for several years, we still found 22 bugs across different BGP implementations, including FRR [27], Quagga [45], GoBGP [28], and Batfish [3]. 11 of the bugs are acknowledged by the developers and 8 of them are already fixed. Our tool is able to find 18 bugs that were unknown previously, while the other 4 had been reported earlier. These bugs cover a range of BGP features, *e.g.*, route map logic, route aggregation, community lists, etc.

[1] https://github.com/rsingha108/MESSI

The rest of this paper is structured as follows. §2 describes the background and motivation of this work using some noteworthy bugs that were automatically found by our tool. §3 describes our solutions to the new challenges that automated testing of BGP introduces, such as regex undecidability and statefulness. Then in §4.2, we describe the experimental setup and methodology, showing how our solutions were executed to test black-box BGP implementations. § 4.3 presents our results – a list of major bugs, with their descriptions, that we were able to find using our tool. §6 surveys related work, while §7 outlines future work and draws conclusions.

Contributions: This paper's contributions are:

- The first automated approach and tool MESSI to identify RFC violations in black-box BGP implementations.
- Modular exploration to deal with protocol complexity.
- Efficient enumerative testing of regular expression inspired by enumerative combinatorics, which cannot otherwise be handled by symbolic testing.
- A testing framework to catch bugs that are generated due to dynamics of BGP often caused by incorrect implementation attempts to do an incremental computation

2 Background & Motivation

The Border Gateway Protocol (BGP) operates as an essential component of the Internet's control plane that connects different autonomous systems (ASes). The BGP protocol has many different aspects. We focus on four specific aspects — the decision process, route filtering, route aggregation, and dynamics — as these are complex and commonly used, and we observed many correctness bugs due to them in the bug databases for popular BGP implementations.

The *decision process* refers to the determination by a BGP router of the best route to some destination, which involves a comparison of route attributes (Local preferences, AS path length, Multi-Exit Discriminator, etc.). The best route is added to the routing information base (RIB) of the BGP router and advertised to neighbors. *Route filtering* empowers administrators to tailor route advertisements based on desired policies. It is performed by *route maps*, which are effectively functions that permit or deny route advertisements based on the properties of those advertisements. Route maps can also transform accepted routes by updating their attributes. Accepted routes then go through the decision process described above. *Route aggregation* is a feature that enables the representation of routes to multiple contiguous IP prefixes as a single summarized route, thus enhancing routing efficiency. Finally, BGP routers must constantly adapt to changes in the environment, for example, the withdrawal of route advertisements and updates to routing policies. We refer to such changes as BGP *dynamics*.

In this section, we demonstrate through concrete examples the challenges for testing BGP implementations as well as the capabilities of our MESSI approach and tool.

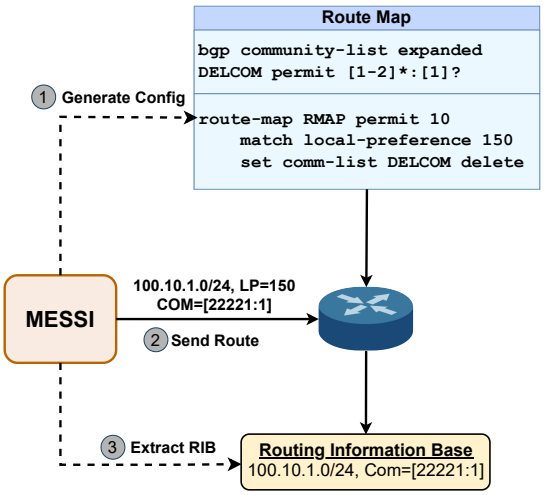

Figure 1: A MESSI test for the BGP route filtering with regular expression.

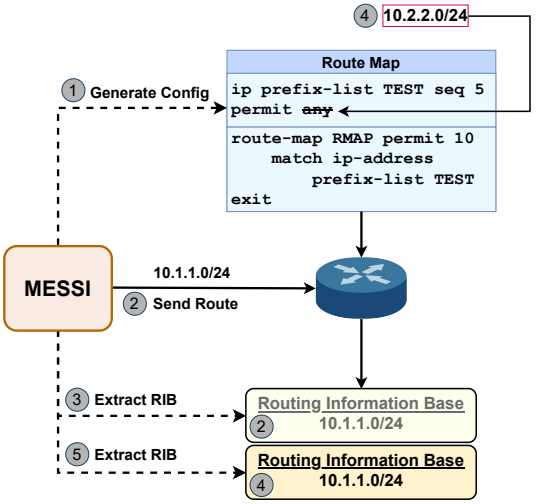

Figure 2: A MESSI test for route filtering with dynamics.

2.1 Route Filtering and Regular Expressions

Figure 1 shows a pair of a route map and route announcement that MESSI automatically generated to test BGP's route filtering behavior. The route map permits routes that have local preference 150 and deletes specific communities from such routes, namely those matching the regular expression in the community list `DELCOM`, which MESSI also generated. The generated route announcement should be permitted by the route map since it has the right local preference, and its community `22221:1` should be deleted since it matches the regular expression `[1-2]*:[1]?`. However, as shown in the figure, when we tested this route map and route announcement on the FRR BGP implementation, we found that the route was accepted, but the community was not deleted.

Upon investigation, we learned that the bug arises only when the matched regular expression has only one ? symbol in it that appears at the end, and the configuration is installed via the command line because this symbol is also overloaded as the "help" command in the configuration console. The result is that the `permit` line in the configuration is completely ignored. Finding this bug is non-trivial: it requires a configuration containing a regex that ends with ?, and there be no other ? symbol in it; it also needs a route announcement that matches the corresponding route-map stanza and matches the regex. When we raised this issue on FRR's GitHub page, they acknowledged it and suggested two workarounds. The first is to use `^+V+?` on the CLI to escape it. The second is to enter a blank space directly after the ?.

2.2 BGP Dynamics

Many BGP implementation bugs lurk in the dynamics of BGP. For example, when a router configuration is changed, the router may not behave as intended after the change is made. This is often caused by (incorrect) implementation attempts to do more efficient incremental computation. We describe an example of this kind that MESSI found automatically.

Dynamics Bug in Route Filtering. Figure 2 shows a pair of a route map and route generated by MESSI. In the FRR BGP implementation, initially, because of the `permit any` line in the generated prefix list, the announcement for `10.1.1.0/24` is permitted. Subsequently, suppose the definition of the prefix list is changed to only permit the prefix `10.2.2.0/24`. Therefore, when a route announcement to `10.1.1.0/24` is sent, it should now be rejected, but we found it was (erroneously) still accepted. The bug was caused by an `ANY` flag that the implementation maintained that was not properly reset upon configuration updates because of an incorrect (but efficient!) incremental computation. The bug was acknowledged and fixed by the developers.

2.3 Route Map Semantics

Figure 3 shows a test case generated by MESSI where it simultaneously generated a route map and a route. The route map refers to a community list with two communities `0.11` and `0:222` and permits any route that has one of those communities. MESSI generated a route with community `0:222` for this test, which should match with community-list `COMM`, and the route should be accepted by the router.

When we ran this test case on the FRR router, FRR did not accept the route. Further investigation showed that the FRR documentation stated the semantics of the community list to be "OR" – i.e., if the incoming route mentions *any* of the communities in the community list, then it should match the community list. However, the implementation displayed "AND" semantics: it only matched when the route contains *all* the communities in the community list. This was evident when we tested the same route map with an incoming route that has community `[0:11 0:222]`. When we posted this issue on the GitHub page of FRR, the developers confirmed it was a documentation bug and fixed the issue.

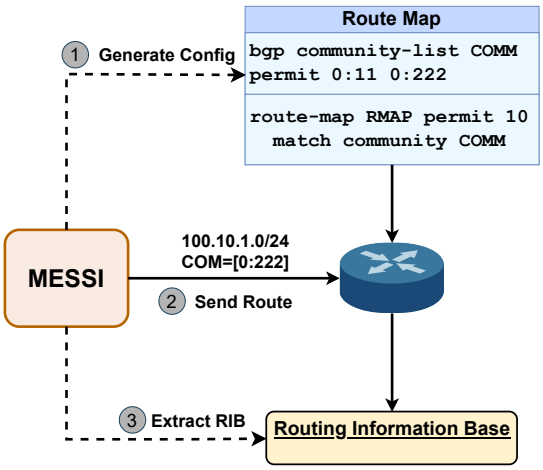

Figure 3: A MESSI test showing buggy behavior of Community list in Route Maps

3 Model Based Testing for BGP

This section describes our approaches to addressing the challenges of test generation for BGP in MESSI. First, we overview our approach at a high level. We then explain in detail how MESSI generates tests for each of the four key aspects of BGP on which we focus.

3.1 Overview

The MESSI approach builds on prior work on model-based testing for network protocol implementations [33]. Specifically, we built an executable model of (parts of) the BGP protocol based on the relevant RFCs. We generally adopted the behaviors that are most commonly used across all the implementations based on our interpretation of the documentation. To generate tests, we *symbolically execute* the model [35], which involves picking execution paths through the model, determining the constraints under which each such path will be taken, and then solving the constraints to generate a concrete test that traverses the path. If any unexpected behavior is observed during testing, we check whether it is a genuine bug or the software implementer's choice. It is a cyclic and iterative process, where we manually validate the model based on our reading of the RFCs but also through our testing setup – unexpected behaviors can indicate bugs in our model, which is then refined.

The model is built in C# using a constraint-solving library called Zen [4, 6]. Zen allows the model to be written as regular C# code but with some inputs declared as *symbolic*. It then performs symbolic execution to solve for these inputs automatically, leveraging Satisfiability Modulo Theories (SMT) solvers like Z3 [21] to solve the constraints. A model would typically have an unbounded number of paths, but Zen offers various ways to limit the set of desired paths, for example, those whose size is bounded by a depth.

Prior work that used this approach built a single monolithic model of the network protocol, which in that case was a DNS nameserver [33]. However, BGP is significantly more complex than DNS nameservers. For example, BGP is stateful, whereas DNS nameservers are stateless, and BGP configurations are much closer to programs — containing forms of conditionals, function calls, and nested structure, for example — than to DNS configurations, which are a sequence of flat records. Symbolic execution on a monolithic model of BGP is infeasible due to *path explosion*: the number of paths grows exponentially in the model size. Thus a monolithic model will have a prohibitively large number of paths, allowing only a small subset of protocol behaviors to be tested in practice.

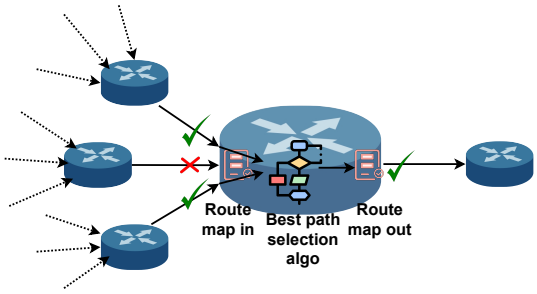

Figure 4: An overview of BGP route selection.

We address this challenge via modular exploration. Instead of a single monolithic model of BGP, we built separate models for key aspects of BGP. Figure 4 shows the BGP route selection process. When a route is received, a route map defined in the BGP configuration is used to determine whether the route is permitted or denied (as shown on the left of the main router); this is called *route filtering*. Subsequently, the best route to a destination is selected and installed in the RIB; this is called the *decision process*. Finally, the selected route is sent to neighbors after passing through a route filtering step.

Leveraging this structure, we built separate models for route filtering and for the decision process. In general, if there are m possible execution paths for route filtering and n paths for the decision process, then there can be $m \cdot n$ paths for a model that combines them, whereas in our approach, there are only $m + n$ paths to explore. Modular exploration also enables us to tailor each model's symbolic inputs to only the parts of BGP that are relevant to that feature, which further simplifies each model and makes test generation feasible. For example, route filtering involves a BGP route map and an input route to be processed, but it is independent of the state of the router, which, therefore, need not be modeled. On the other hand, the BGP decision process depends on the current router state but is independent of the route maps in the BGP configuration.

Figure 5 overviews the architecture of MESSI. The symbolic module contains separate executable models for the BGP decision process, route filtering, route aggregation, and forms of protocol dynamics, which we discuss in turn in the following subsections. As shown in the figure (and discussed further below), each such model has a different set of inputs and outputs tailored to that feature's role in the overall BGP process. In each case, we use Zen to generate concrete test cases (indicated

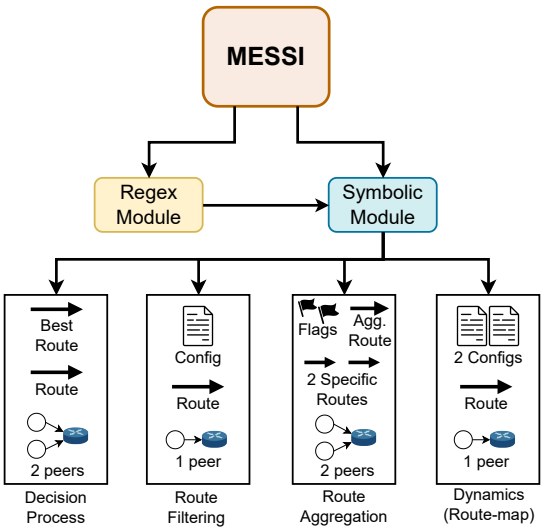

Figure 5: The architecture of MESSI

in the figure as the Symbolic Module), which we then execute on black-box BGP implementations. A particular challenge for test generation of route filtering is the need to generate regular expressions, as these are commonly used in route maps to match on communities and AS paths in routes. We have developed a special technique to handle regular expressions (shown as the Regex Module) that we describe in detail below.

3.2 Decision Process

As shown in Figure 5, we model the decision process as a function that takes two routes to a single destination and returns the route that is more preferred. We argue that these inputs are sufficient to identify any error in the BGP decision process. The decision process is used whenever a router receives an incoming route announcement and decides to permit it (possibly with some modifications). At that point, the newly permitted route is compared against the current best route to that destination in the RIB to determine whether the RIB should be updated. Our model captures exactly this scenario, where one of the two input routes models the current best route in the RIB and the other models the new route.

The BGP decision process involves comparing the various attributes of the two given routes in a particular order of precedence, in order to decide which route is more preferred. For example, first, the attribute called "local preference" of the two routes is checked, and the route with the higher local preference is preferred. If the local preference values are equal, then the BGP decision process moves on to check the values of other attributes, and so on.

Our Zen model is, therefore, simply a function that takes two routes and compares their attribute values in the correct precedence order to determine which route is preferred. Marking the two routes as *symbolic* ensures that Zen will explore all paths through this function and, for each path, will generate two concrete input routes that cause execution

```
1  Zen<Route> Compare(Zen<Route> r1, Zen<Route> r2){
2      // compare attributes
3      var gtLp = r1.LocalPref() > r2.LocalPref();
4      var neLp = r1.LocalPref() != r2.LocalPref();
5      var ltLen = r1.PathLen() < r2.PathLen();
6      var neLen = r1.PathLen() != r2.PathLen();
7      ...
8      var ltRid = r1.RouterId() < r2.RouterId();
9      var neRid = r1.RouterId() != r2.RouterId();
10
11     // compare the routes
12     var ridCmp = If(ltRid, r1, If(neRid, r2, r1));
13     ...
14     var lenCmp = If(ltLen, r1, If(neLen, r2, ...));
15     var lpCmp = If(gtLp, r1, If(neRid, r2, lenCmp));
16     return lpCmp;
17 }
```

Figure 6: Pseudo-code for BGP decision process.

to follow that path. For example, Zen will generate a pair of input routes for the case when the first route has a higher local preference than the second one, another pair for the case when the second route has a higher local preference than the first one, and many pairs where the local preferences of the two routes are equal so that other paths will be explored.

In Figure 6, we show the pseudo-code for the BGP decision process. Note that we assume that the `always-compare-med` and `compare-routerid` flags are enabled. Turning the first flag on allows us to have a complete ordering between routes. However, we can easily modify our model to accommodate other comparison modes for MED. In Lines 3 to 9, we first compare the routes' attributes according to the decision process preference. Next, we order these decisions Lines 12 to 15 by comparing the attributes in the correct order. We build the hierarchy of attribute comparisons in a bottom-up fashion using nested if-then-else expressions. If the local preference of the first route is higher, then we return the first route. Otherwise, if the second is preferred, we return it instead. If they are equal, then we move on to the AS path length. In this way, we mimic the BGP decision process by sequentially comparing the attributes of the two routes in order of preference.

3.3 Route Filtering

To test route filtering, we employ a *hybrid* approach, which leverages both an executable model for symbolic execution and a special approach to support generation of regular expressions. We describe each in turn.

3.3.1 Zen Model

As shown in Figure 5, we model the route filtering process in Zen as a function that takes two inputs: a configuration, specifically a route map and associated definitions like prefix lists and community lists; and an incoming route. The route filtering process applies the route map to the route to determine if the route is permitted or denied; in the case that it

is permitted, the route is also modified as specified in the route map. Notably, route filtering is *stateless*: the processing of an incoming route by a route map is independent of the current router state, such as the contents of the RIB. Hence there is no need to model that state. The previous section showed examples of pairs of route maps and routes that MESSI automatically generated from its route filtering model.

In general, a route map consists of one or more *stanzas*. Each stanza consists of an action (permit or deny), zero or more *match* statements, which indicate the conditions under which the stanza matches the input route, and zero or more *set* statements, which update the incoming route if it is permitted. Stanzas are processed sequentially until finding a stanza that matches the input route, in which case the corresponding action and set statements are applied. Route maps can also contain more sophisticated control flow; for instance, stanzas can be chained together, with one falling through to the next, and route maps can invoke other route maps.

Handling the full complexity of route maps described above is simply infeasible. Therefore, we take our cue from prior work on model-based testing for protocols, which found that small configurations were sufficient to surface many interesting implementation errors [33]. Our Zen model considers a route map to consist of exactly one stanza. While this is a significant simplification, it nonetheless creates a rich space of policies to explore. Within a stanza, we allow any number of match statements, and they can match on all of the different attributes of a route. We similarly allow any number of set statements, which can update those attributes in various ways, for example, adding and deleting communities. Finally, as shown in the earlier examples, our model also includes auxiliary structures like prefix lists and community lists, which can be generated by Zen and referred to in route maps.

3.3.2 Generating Regular Expressions

One significant source of complexity in route maps that we *cannot* avoid is the use of regular expressions to match on communities and AS paths. Their usage is ubiquitous, and regular expressions are also a common source of errors; thus, any test generation approach for route maps must support them. A naive approach would be for our Zen model to simply include the logic for checking whether a string satisfies a regular expression and ask Zen to symbolically execute this code. However, doing so would dramatically explode the number of paths through our model (since, for example, it would include the logic to convert a regex to a DFA). Further, solvers [21] cannot handle the theory of strings with *symbolic* regular expressions since they can describe non-regular languages [46].

Instead, we developed a hybrid approach to generating route maps containing regular expressions that avoids the need to symbolically execute the regex matching process while also obtaining useful coverage guarantees for both regexes and the strings that are matched against them. First, we exhaustively enumerate *regex structures* up to some size (a form of coverage), which are regexes that are parameterized by the base regexes within them. Next, we employ random generation to produce the base regexes for each regex structure, yielding concrete regexes. Third, for each such regex we generate a set of positive and negative example strings that together cover all nodes and edges in the DFA representation of the regex. Finally, these regexes and their associated examples are given as concrete inputs to our Zen model to use when building route maps and routes. Next, we describe these steps in more detail.

Algorithm 1 Enumerating regexes at a given level

```
1:  function RegexEnumerator(R_{n-1})
2:      R_n ← []
3:      for i in range(len(R_{n-1})) do
4:          r ← R_{n-1}[i]
5:          R_n ← R_n + [r, (r)*, (r)+, (r)?]
6:          for s in (R_{n-1}[i:]) do
7:              R_n ← R_n + [(r)|(s)]          ▷ Alternation
8:              R_n ← R_n + [(r)(s)]           ▷ Concatenation
9:          end for
10:     end for
11:     return R_n
12: end function
```

Regex enumeration algorithm. We are inspired by work in enumerative combinatorics to count [38] (not generate) regular expressions. Similar ideas can apply to other complex protocol structures for which a recursive formulation is feasible. We use the algorithm shown in Algorithm 1 to enumerate regex structures exhaustively up to some level n, given the regex structures up to level $n-1$.

We use the character 'x', which represents a regex to be filled in later, as the sole regex structure at level 0. According to the algorithm, level 1 will consist of the regex structures {`x`, `(x)*`, `(x)+`, `(x)?`}. Similarly at level 2 we will have regex structures like `(x*)|(x+)`, `(x+)?`, and so on. The number of regex structures at every level increases exponentially, so we stop at level 3 with 4686 such structures.

At the n^{th} level the complexity of Regex Generation is $O(R_{n-1}^2)$ where R_{n-1} is number of regexes generated at the $(n-1)^{th}$ level. If we solve the recursion $T(n) = T(n-1) * T(n-1)$ then we get $T(n) = O(2^{2^n})$

Random regex generation. For each of these structures, we randomly generate valid community regexes to fill in their placeholder values 'x', via the following grammar:

T	::=	$(E_1 \mid \ldots \mid E_k)$	*(top-level-regex)*
E	::=	`^S$`	*(end-to-end-regex)*
S	::=	$(C_1 \ldots C_k)$	*(sequence-regex)*
C	::=	`P:P`	*(community-regex)*
P	::=	$(B_1 \ldots B_k)$	*(community-segment-regex)*
B	::=	`R` | `R*` | `R+` | `R?`	*(base-regex)*
R	::=	`[N-N]`	*(regex-number-range)*
N	::=	`0` | `1` | ... | `9`	*(number)*

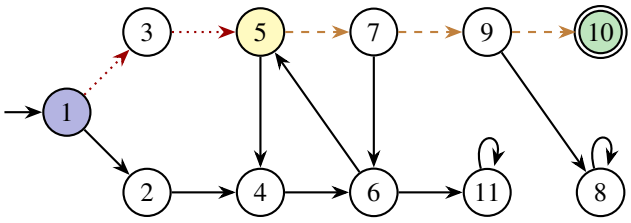

Figure 7: Node Coverage on the DFA: To generate a positive example that covers state 5 in the above figure, we first find a path P1 (dotted maroon) from the starting state (1) to state (5). In this case, it is (1) → (3) → (5). Then we find another path P2 (dashed brown) from state (5) to the accepting state (10).

An example random regex generated from this grammar is:

```
(^[2-4]+[3-5]*:[2] [3]:[1-2]?$)|(^[5-6]+:[4]*$)
```

After plugging these randomly generated strings into the regex structures generated by the enumeration algorithm, we obtain a set of concrete regexes.

String generation For each regex that the Zen model uses in a community list, we require both positive and negative examples to use as concrete community values in routes, and we would like to choose these examples in a way that covers many interesting behaviors of the regex. We have devised a simple and effective approach to doing this, through efficient graph algorithms on the DFA representation of each regular expression. Specifically, we generate positive examples that together cover all nodes as well as all edges of the DFA that are reachable, and similarly for the negative examples.

Figure 7 illustrates the approach on an example DFA. Suppose we wish to identify a positive example that includes node (5). First, we use depth-first search (DFS) to find a path from the start node (1) to node (5) (shown in the figure as the dotted path). Then we use DFS again to find a path from node (5) to the accepting node (10) (the dashed path). Each DFA edge is labeled with the input character that causes that edge to be traversed, as usual (not shown in the figure). Hence, we traverse the concatenation of these paths to collect these labels, yielding a positive example. We iterate this process over all nodes, keeping track at each iteration which nodes have already been covered. We use an analogous approach to cover each edge, using DFS to identify a path from the start node to the edge's source and a path from the edge's target to the accepting node. Finally, to generate negative examples, we use the same procedure but on the DFA that represents the complement of the original regex.

3.3.3 Regex Testing

Using the algorithms discussed above, we obtain a total of 253,958 test cases, each consisting of a regular expression and a string that should be either a positive or negative example. We use these test cases in two ways. First, we generate a route-filtering test case for each one to test the regex-matching logic of the BGP implementations. For example, given the regex `^[2-4]+[3-5]*:[2]+$` and positive example `2335:22`, we generate the following route-map, which accepts routes whose community matches our regex, as well as an input route whose community value is `[2335:22]`:

```
ip community-list 101 permit ^[2-4]+[3-5]*:[2]+$
route map FILTER_ROUTES permit 10
       match community 101
```

Second, we randomly choose a small subset of the generated regexes (4 in our experiments) and "hardcode" these into our Zen model. For each such regex, we also randomly choose a small set (3 in our experiments) of associated positive and negative examples to use in community lists. When symbolic execution explores a path that successfully matches the input route against a community list, Zen selects one of the predetermined regexes to use in the generated route map and one of its positive examples to use in the generated input route. This allows us to generate complex tests involving regexes (e.g. tests with multiple match and set statements), without modeling and symbolically reasoning about regex matching logic. The Zen model also uses this pool of regexes and examples to generate tests for community deletion.

We have described our regex generation process specifically for community regexes, but we use a similar approach to generate regexes over AS paths and the corresponding positive and negative example AS path values. Note that each component of our hybrid approach (symbolic, random, enumerative) has coverage guarantees, although of different forms; one can increase test coverage by increasing resources.

3.4 Route Aggregation

As shown in Figure 5, we model the route aggregation process in Zen as a function that takes two routes, the aggregate prefix, as well as some aggregation-specific flags, and produces (possibly multiple) output routes. Our Zen aggregation model first checks whether each of the two incoming routes matches the aggregation prefix. Based on this information and the values of various aggregation-specific flags, it then determines the set of routes to return. Aside from the `matching-med-only` flag, another example flag is `summary-only`, which, if set, indicates that only the aggregate route should be advertised. Conversely, if this flag is not set, then both the individual routes and aggregate route should be advertised.

3.5 Testing BGP Dynamics

BGP routers must constantly respond to changes in the environment — a link can go down, a route can be withdrawn, a route policy can be updated, etc. Efficiently handling these kinds of changes is challenging for developers. We have observed that erroneous incremental computation is at the heart of many identified implementation bugs (e.g., [1, 11, 12, 23]). We have developed an approach in

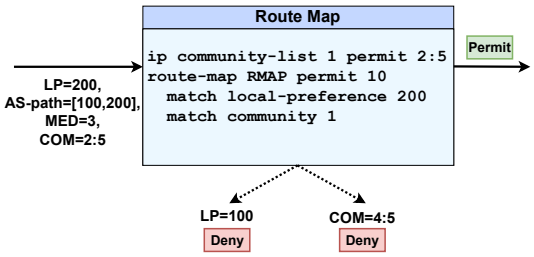

Figure 8: Changes to route map to alter the route-map decision.

MESSI to generate tests to identify incorrect incremental computation. We consider how the route filtering process is affected by a limited form of dynamics: either a modified route map or a modified (withdrawn and newly received) route.[2]

The setup for testing a modified route map is shown in Figure 8. Now our Zen model takes *three* inputs: an original route map, a route to match against, and an updated route map. The goal is to produce test cases whereby the route matches the original route map but not the updated one and vice versa.

Figure 9 similarly shows the setup for testing the case of modified routes. Now a test case consists of a route map, an original route, and an updated route, and we require that the original and updated routes are treated differently by the route map.

The key challenge in generating tests (for even the very limited dynamics we focus on) is the large size of the search space. For example, consider the Zen model for testing route-map changes, which takes as input two route maps and a route. If we perform symbolic execution naively, Zen must explore the space of all possible pairs of route maps that behave differently on the given route. This space is too large, so we will end up exploring a small subset of the space, but with no clear guarantees on coverage.

Instead, we leverage Zen to perform a form of exhaustive *local search*. Specifically, we define a *distance metric* between route maps, which intuitively represents the number of changes to individual stanza actions and match statements that must be made to transform one route map into another akin to Hamming distance. Then we ask our Zen model to only generate pairs of route maps that are within distance k of one another ($k = 1$ in our experiments), which can be done exhaustively in a reasonable amount of time.

For example, consider the following route map:

```
ip community-list 1 permit [1-2]+:[1-2][3]*
route map FILTER_ROUTES permit 10
       match community 1
```

We consider changing the action from `permit` to `deny` as a cost of one. Similarly, changing the regex in community list 1 to another regex is considered a cost of one. Notably, we treat match or set statements on different attributes as being infinitely far apart. For example, if the `match community` statement in our example is changed to a `match metric` statement, then the distance is infinite. In short, our distance metric models the kinds of local changes that network operators often make to route maps, in our experience, and it also makes symbolic test-case generation feasible.

[2]Our use of "small step" BGP dynamics should not be confused with "multi-step" BGP dynamics that manifest in phenomena such as BGP looping and route flap damping.

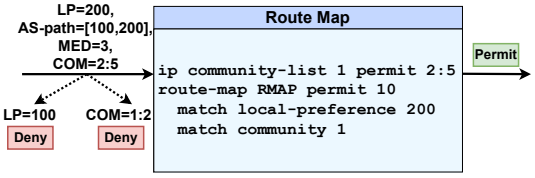

Figure 9: Dynamic Testing by changing route attributes to alter the route-map decision

We use a similar approach to test a modification to the incoming route. Namely, we define a distance metric on route announcements, where each attribute modification has a cost of one, and the Zen model exhaustively explores pairs of routes that have distance at most k ($k = 1$ in our experiments).

We also tested the dynamics of route aggregation similarly, where we allow changes to either the route attributes in one setting or the values of aggregation-specific flags and aggregation prefix (config) in another setting. The distance metric is defined in a similar fashion as above: changing the aggregate prefix, the flags, or a route attribute is considered a cost of one. In our experiments, we constrained this cost to be at most 1.

While supported by our model, increasing distance beyond 1 increases both test cases and generation time exponentially.

3.6 Vendor-specific Translators

BGP is a complicated protocol, and vendors tend to implement their own configuration languages, which differ in syntax, semantics, and features. For instance, both FRR and Quagga support prefix lists, community lists, and AS path lists. However, GoBGP supports *sets* of prefixes, communities, and AS paths, as opposed to lists. The fundamental difference between these two representations is that elements within a list have sequential priority, which allows for overlapping permitted and denied ranges, whereas set members are unordered. We address these differences through special-purpose translators that convert our Zen-generated tests to the input language of a particular implementation.

As an example, suppose Zen generates the following prefix list, denoted in FRR syntax:

```
ip prefix-list PR seq 1 deny 10.1.0.0/15
ip prefix-list PR seq 2 permit 10.0.0.0/8 ge 15 le 18
```

Note that the prefix on line 1 is in the prefix range on line 2. Our GoBGP translator converts this list into the following set of all prefix ranges permitted by line 2 but not denied by line 1:

```
NAME  PREFIX
PR    10.0.0.0/8 16..18
      10.2.0.0/15 15..15
```

Implementation	Language	Description
FRR [27]	C	Used by ISPs, DCs
Quagga [45]	C	Used in Linux
GoBGP [28]	Go	Used by ISPs, research
Batfish [3, 26]	Java	Open-source simulator
Fastplane [40]	C++	Fast BGP simulator
BIRD [13]	C	Dynamic IP routing daemon

Table 1: Implementations tested by MESSI.

```
10.4.0.0/14 15..15
10.8.0.0/13 15..15
10.16.0.0/12 15..15
10.32.0.0/11 15..15
10.64.0.0/10 15..15
10.128.0.0/9 15..15
```

The translator produces prefix sets whose elements are mutually exclusive, which simplifies translation; but it implies we never test GoBGP prefix sets with overlapping elements.

Note that our test cases cover a broad range of settings and attributes, some of which might not be supported by every implementation. For example, GoBGP does not support setting the MED attribute or the removal of AS numbers from an AS path, while both FRR and Quagga do. We also did not find documentation for route aggregation in GoBGP, while both FRR and Quagga have extensive support for it with special flags such as `matching-med-only` and `summary-only`. Similarly, BIRD only offers basic support for regular expressions. Thus while translating test cases for a specific implementation, we ignore inapplicable tests.

3.7 Implementation Testing

We use an automated testing setup that takes as input the translated test cases and sets up a network of Docker containers, one of which runs the implementation under test. The other containers use ExaBGP to send route announcements to this container. We parse the RIB of the implementation under test to obtain the installed routes that result from each test; we consider a test to fail if an implementation's RIB differs from that expected by our model. We group failed tests to reduce the manual effort of debugging. During test generation, our Zen code produces a tag for each test case based on the path it traverses in the code, which roughly indicates the kinds of match and set statements that the test exercises. Failed tests are grouped into *buckets* by tag, and one representative test from each bucket is manually investigated to confirm whether it is a bug or not. This approach is analogous to the bucketing technique from our prior work on test generation for DNS [33].

FRR [27]	
Community list AND/OR matching semantics	Fixed
Filtering routes tagged with Internet Community	Fixed
Delete Community Regex not working	Fixed
Route map permits all routes if internet community permitted	Fixed
Prefix list matches mask greater than or equals	Acked
Changing MED results in incorrect aggregation	Fixed
Route map change from permit any prefix results in incorrect route installation	Fixed
Quagga [45]	
Community list matching AND/OR semantics	Found
Filtering routes tagged with Internet Community	Found
Delete Community Regex not working	Found
Route map permits all internet community routes	Found
Changing MED results in incorrect aggregation	Found
GoBGP [28]	
Prefix set matching with zero masklength but nonzero range	Acked
Withdrawn route is still advertised	Fixed
Route installed in RIB with incorrect MED	Fixed
Batfish [3, 26]	
Space in Community regex giving error	Acked
Internet community route filtering	Found
Fastplane [40]	
Set MED not advertised	Found
AS prepend with 0	Found
AS path regex parsing issue	Found
AS path regex matching issue	Found
Community Regex matching issue	Found

Table 2: Bugs found for different BGP implementations.

Stats	Decision	Filtering	Aggregation	Dynamics
Tests	18	142916	128	11246
Failed	5	457	0	12
Buckets	2	204	0	4
Bugs	1	17	0	4
Acked	1	7	0	3
Fixed	1	4	0	3

Table 3: Experimental results.

4 Results

4.1 System Overview

We generate tests for each module described earlier using the corresponding Zen model. We test implementations using Docker [42] by creating working Docker images of each implementation. For route-map filtering, test generation takes around 2 days; hence, we pipelined test generation by testing the implementations. We used Python scripts for vendor-specific translation (§3.6) and response grouping (§3.7).

4.2 Experiments

We tested six BGP implementations: FRR, Quagga, GoBGP, Batfish, Fastplane, and BIRD. Table 1 shows their source code language and a brief description of their focus or how they are used. Table 2 catalogues all the bugs we found. In Table 3, the first three rows show the number of Zen-generated test cases , failed tests and buckets respectively. The last three rows list the number of bugs found, acknowledged and fixed across all implementations respectively. We found at least 2 bugs in each of the 5 implementations except BIRD, in which we found no bugs. Many of our identified bugs relate to regexes or dynamics. However, BIRD has rudimentary support for regexes, making many of our test cases inapplicable. Further, unlike FRR and Quagga, BIRD restarts the daemon upon config changes, making incremental update errors unlikely.

We next describe the results for each module individually.

Decision Process: For testing the decision process of BGP, we made a model of the BGP decision algorithm in Zen. It generated 18 test cases corresponding to every path (pseudocode shown in Figure 6). We found 1 bug in the decision process of GoBGP and found 4 discrepancies across FRR and Quagga, which revealed some implementation-specific behavior – i.e., for Quagga to choose the route with lower router-id, the bgp `bestpath compare-routerid` flag has to be enabled; by contrast, this is enabled by default for FRR.

Route Filtering: To test BGP route filtering, MESSI simultaneously generates an incoming route along with an inbound routing policy (route map) as a test case. A route map, in general, can have multiple stanzas, with each stanza having its own match and set statements. However, as stated earlier, MESSI generates tests with only one stanza.

We encoded the route map evaluation logic (explained in § 3.3) in Zen for test generation. We also restrict each field of the route and the route map with some validity constraints. The regex testing was discussed earlier in § 3.3.3. MESSI generated 4686 regexes (up to level 3) using enumeration Algorithm 1 and generated positive and negative examples using the coverage algorithm (Figure 7) with a total of 253,958 test cases for regex testing.

In the route map, we only allowed prefix lists to have 3 entries. We did this not only to reduce the search space but also to ensure that we had test cases that could check whether the BGP implementations were following the sequential order while matching entries within the prefix list. IP addresses were limited to unsigned integers within a certain specified range. This is a necessary step so that Zen does not generate invalid IP addresses such as `0.0.25.203` that do not follow CIDR rules.

Second, for ease of computation, we represented subnet masks in their unsigned integer form. The primary reason for doing this is that Zen does not support shift operations like « and ». Thus instead of representing the masks internally as a number in the range 0-32, we converted them to their corresponding unsigned integers. This made it easier to `AND` the subnet masks with the IP addresses to get the prefix value.

```
1  Zen<bool> RMapDynamics(Zen<Route> route,
2      Zen<RouteMap> r1, Zen<RouteMap> r2)
3  {
4      // Limit maximum distance to 1
5      var cond1 = GetDifference(r1, r2) <= 1;
6      // get decisions from the two route maps
7      var dec1 = r1.Match(route);
8      var dec2 = r2.Match(route);
9      // check whether they have different results
10     var cond2 = Not(dec1 == dec2);
11     return And(cond1, cond2);
12 }
```

Figure 10: Constraints for route map dynamics

We also constrained the prefix mask, LE, and GE values to obey the rule `mask <= GE <= LE`. Additionally, to avoid repetition and redundancy, all entries in the prefix list were made unique. AS path and Community lists were limited to 1 entry each. This tactic was again taken to reduce the number of test cases. Finally, local preference and MED values were restricted within a pre-specified range.

This setup generated 142,916 tests, revealing 15 previously unknown bugs across the tested implementations.

Route Aggregation: MESSI uses the model described in §3.4 to simultaneously generate the aggregate prefix, the flags, and two individual prefixes for a test case.

Zen generates 128 test cases to cover all paths in this module. We did not find any bugs in the route aggregation logic across all implementations.

BGP Dynamics: We also tested the dynamics (explained in §3.5) related to route map filtering and route aggregation. The dynamics of route map filtering are further classified into two categories - (1) dynamic changes in the route map and (2) dynamic changes in the incoming route. In the experiments, we use the distance metric (how close route maps or routes are to each other) defined in §3.5.

For (1), we wrote a wrapper method around the route map evaluation module to generate two route maps and an incoming route, subject to the constraints that the route maps should differ by, at most, a unit distance, and one of them should allow the route while the other denies the route. This setup generated 7404 test cases. Figure 10 shows the logic for implementing the wrapper. First Line 5 limits the maximum distance between two route maps to 1. Next, Lines 7 to 8 evaluates the incoming route against the two route maps. Finally, in Line 10, we check to see whether the two decisions are different. For a particular test case, both these conditions must be satisfied.

For (2) we wrote another Zen wrapper to simultaneously generate two routes and one route map. The constraints it is subjected to are: first, the two routes should differ by at most a unit distance; and second, the route map should give different decisions for the two routes. This setup generated 28 test cases.

We also tested route aggregation dynamics, simultaneously generating configs (aggregate prefix + flags) and incoming

routes for two scenarios: changing routes and changing config. Every change is of unit distance. In both scenarios, our constraints ensured the published set of routes are different for the two different settings. In the changing routes scenario, we altered one route keeping a constant config. Conversely, in the changing config scenario, we modified the config while maintaining identical routes. This setup generated 3814 test cases.

Additionally, with each test case, we tested the effect of withdrawal. Dynamic testing revealed 4 bugs across all implementations.

4.3 Example Bugs

In this subsection, we provide more examples of the bugs that MESSI identified, in addition to the ones presented in §2.

Bug #4: The following test was generated by MESSI:

```
Route Map:
ip community-list 1 permit internet
route map FILTER_ROUTES permit 10
       match community 1
Route: 100.10.0.0/16, COM = [0:11]
```

In the above test case, the route map is configured in a way that only permits routes with the Internet community. Although it is expected that only routes containing the internet community (0:0) will be permitted, we observed that FRR and Quagga permit all incoming routes due to a bug in handling the community list. In the source code, there was a code snippet that was setting the decision as `permit` if `internet` community was contained in the community list. The developers in FRR acknowledged and fixed this bug.

Bug #5: MESSI generated the following test case:

```
Route Map:
ip community-list 1 permit 0:0
route map FILTER_ROUTES permit 10
       match community 1
Route: 100.10.0.0/16, COM = [0:0]
```

Here, the route map permits routes tagged with 0:0 (`internet`) community, and the incoming route has community 0:0. Therefore, the expected decision is `PERMIT`. For this particular test case, we observed some discrepancies across different implementations. When we ran this test on FRR, it denied the route, although Fastplane and Batfish allowed it (as expected). If we use `internet` in the route map, and the incoming route has `0:0` as a community, then FRR gives the expected output. But, Batfish doesn't give the expected output. When we filed this issue, FRR developers fixed it by deprecating the `internet` community. We have reported this to the Batfish developers and are waiting for their response.

Bug #6: In another test case generated by MESSI the route map permits all routes that match with default route `0.0.0.0/0` with a mask length greater than or equal to zero. But in FRR, we observed that it does not work. It works as expected in Batfish and Fastplane. However, `ge 1` does work, and it was given as a workaround by the FRR developers.

```
Route Map:
ip prefix-list PFXL seq 5 permit 0.0.0.0/0 ge 0
route-map FILTER_ROUTES permit 10
       match ip address prefix-list PFXL
Route: 100.10.0.0/16...
```

The documentation does not explain why `ge 0` does not work. Neither does FRR throw any error if `ge 0` is provided in the configuration. GoBGP had a similar issue: if there is a prefix set with prefix `0.0.0.0/0` but with a range on mask length, it should match all routes within that mask length range but did not match a prefix generated by MESSI. The GoBGP developers acknowledged this and will hopefully fix this soon. The failed test case generated by MESSI was:

```
prefix-set ps1:
  ip-prefix: 0.0.0.0/0, mask-range: 10..10
policy-definitions:
    conditions:
      match-prefix-set: ps1
Route: 100.10.0.0/10
```

Bug #7: A route map contains a set statement that deletes specific communities from routes that match with some community list. The route communities are not deleted as expected because of an additional `delete` tag that was added to the name of the community list, which made the delete community not work. We found this bug in Quagga and in earlier versions of FRR (fixed later). The failing test case was:

```
Route Map:
ip community-list 101 permit [1-2]+:[3-4]
route map FILTER_ROUTES permit 10
    set coom-list 101 delete
Route: COM = [11:4 0:1]
```

The expected result is the deletion of community `11:4` from the route but it did not delete any communities.

Bug #8: The following test was generated by MESSI:

```
Route Map:
ip prefix-list PFXL seq 5 permit 99.0.0.0/8 le 31
route-map FILTER_ROUTES permit 10
    match ip address prefix-list PFXL
Route: 99.10.11.0/24, MED = 4
```

This test case failed for GoBGP and Fastplane. Here, the incoming route has a `MED 4`, and the inbound route map accepts the route without modifying it. But the installed route had `MED 0`. This error was fixed in GoBGP and is under review by the Fastplane developers. The same test case was also generated by our decision process setup.

Bug #9: Another test case generated by MESSI revealed a bug in Fastplane. The test case is as follows:

```
Route Map:
route-map FILTER_ROUTES permit 10
    set as-path prepend 0 100
Route: AS-path = 200
```

Here the inbound route map has a set statement that should add AS 0 and AS 100 to the AS path of the route, but it only appended AS 100 to the existing AS path. All other implementations appended it correctly. This issue is likely due to a highly compact zero-terminated representation that Fastplane uses to represent the AS path for BGP routes.

5 Limitations

Our model currently focuses on RFC compliance errors only; we do not consider performance bugs or coding bugs such as overflow errors. We do not test route reflectors, confederations, reserved ASNs, and well-known communities, and some regex features such as constraining a route's community set size.

Modular exploration does not test possibly complicated interactions among multiple BGP features. However, the modular approach is more scalable, allowing each feature to be tested extensively; it also allows easily adding support for new features without changing existing portions of the model significantly. We can also use our approach to test the integration of multiple modules when desired.

MESSI automatically configures routers through the command line interface. Certain bugs (e.g., §2.1) would not be revealed by alternative testing approaches, such as directly writing the configuration file. In future work, we could employ multiple test setups to widen the scope of errors found.

6 Related Work

We classify related work into the following categories.

Model-based verification: This approach uses a formal model to verify software. For example, Bagpipe [51] verifies BGP configurations against a formal model of BGP. Verification scales for configurations but not for implementations.

Fuzz testing: Fuzz testing is widely used for software testing in general [2, 14, 37, 52] and specifically for BGP implementations (e.g., [20, 22]). Although fuzzers are effective at finding parser bugs, they are less effective at finding behavioral bugs such as those described in §2.

Symbolic execution: Symbolic execution invokes SMT solvers to generate test cases for as many execution paths of a program as possible (e.g., [17, 29, 30]). BGP implementations contain thousands of lines of low-level code, which makes symbolic execution infeasible in practice.

Model learning: Recent work uses active learning to create abstract models from black-box protocol implementations [25]; errors are detected by comparing the models of different implementations. Compared to our approach, the drawback is the need to learn a model for each protocol implementation; we generate tests for arbitrary protocol implementations from a single reference model. However, their models can be used for purposes other than error detection.

Model-based testing: This general category uses an abstract model of a system to generate tests [16,44,49]. MESSI extends the SCALE model-based testing approach for DNS implementations [33] to account for BGP's complexity and statefulness.

Enumerative Testing: This approach generates inputs up to a given size that meet a specified predicate [15]. We use a form of enumeration to generate regexes for BGP configurations.

7 Future Work and Conclusion

We plan to extend MESSI to test other BGP features such as route redistribution and route reflection. We also aspire to *automatically* derive the constraints from the BGP RFCs, say using large language models like GPT-4, to alleviate the current burden of manually building the model.

The ideas in MESSI should extend to other stateful protocols with complex structures, as well as to other software systems, such as web servers. BGP has a comparatively simple stateful model: if one excludes dynamics, the forwarding of a route depends only on the previous best stored route. However, the state of other protocols may depend on many past messages; this raises the question of efficiently driving such protocols to specified states. MESSI's combination of symbolic, random, and enumerative testing of structures should extend to other protocol structures.

A common slogan when an Internet outage occurs is: "It's always DNS ... except when its BGP" [10] because the majority of major Internet incidents [10] are caused by bugs in DNS (zone files or implementations) or BGP (configurations or implementations). Previous work targets DNS zone file bugs [32], DNS implementation bugs [33], and BGP configuration bugs (e.g., [26]); MESSI fills the remaining gap by targeting BGP implementation bugs.

In terms of ideas, MESSI adds techniques to the repertoire of model-based testing approaches for protocol implementations (e.g., SCALE [33]) to deal with: **1.** complexity (modular exploration), **2.** protocol statefulness (generating both the best route and a new route); **3.** implementation statefulness due to incremental computation (generating two configurations that differ by a single change); **4.** complex structures (combining symbolic with enumerative testing for regexes).

Inspired by the famed soccer player, MESSI tries to attack the goal of BGP testing from many angles, generating over 150K test cases automatically that have already led to the discovery of 22 new bugs in 5 BGP implementations.

Acknowledgements

We thank our shepherd Eric Eide and the anonymous reviewers for their insightful comments. We also thank the BGP developers for their feedback on the bug reports. This work was partially supported by NSF grant CNS-1901510 and by Cisco.

References

[1] About the real-time effect of BGP policy configuration. https://github.com/osrg/gobgp/issues/2164.

[2] American fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[3] Batfish network configuration analyzer. https://github.com/batfish/batfish.

[4] Ryan Beckett. Zen. https://github.com/microsoft/Zen/tree/master.

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

[6] Ryan Beckett and Ratul Mahajan. A general framework for compositional network modeling. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 8–15, 2020.

[7] Ann Bednarz. Global microsoft cloud-service outage traced to rapid bgp router updates. *Network World*, 2023.

[8] BGP 'address-family ipv4' sub-configuration is not shown in the running configuration. https://quickview.cloudapps.cisco.com/quickview/bug/CSCvk45884.

[9] BGP bug bites juniper software. https://www.networkworld.com/article/2289950/bgp-bug-bites-juniper-software.html.

[10] BGP, DNS, and the fragility of our critical systems. https://www.f5.com/labs/articles/cisotociso/bgp-dns-and-the-fragility-of-our-critical-systems.

[11] BGP route-map work incorrectly when prefix-list modify from deny any to permit. https://github.com/FRRouting/frr/issues/13007.

[12] BGPD: aggregate-address with summary-only and matching-med-only does not work when metric is changed. https://github.com/FRRouting/frr/issues/11912.

[13] BIRD routing software. https://bird.network.cz/?get_doc&v=20&f=bird-5.html.

[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[15] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In Phyllis G. Frankl, editor, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, pages 123–133. ACM, 2002.

[16] Josip Bozic, Lina Marsso, Radu Mateescu, and Franz Wotawa. A formal tls handshake model in lnt. *arXiv preprint arXiv:1803.10319*, 2018.

[17] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[18] Tom Strickx Celso Martinho. Understanding how facebook disappeared from the internet. *Cloud Flare*, 2021.

[19] Cisco ios xe software bgp resource public key infrastructure dos vulnerability. https://bst.cisco.com/quickview/bug/CSCvz55292.

[20] Stanislav Dashevskyi. A simple BGP fuzzer based on boofuzz. *Github*, 2023. https://github.com/Forescout/bgp_boofuzzer.

[21] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] Donatas Abraitis et al. Donald Sharp. fuzzing targets and supported fuzzers available in frr. *Github*, 2023. https://docs.frrouting.org/projects/dev-guide/en/latest/fuzzing.html.

[23] eBGP doesn't withdraw routes to other ebgp peers when peer goes down. https://github.com/osrg/gobgp/issues/2208.

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

[25] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. Prognosis: closed-box analysis of network protocol implementations. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 762–774, New York, NY, USA, 2021. Association for Computing Machinery.

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

[27] FRR routing software. https://github.com/FRRouting/frr.

[28] GoBGP routing software. https://github.com/osrg/gobgp.

[29] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

[30] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[31] DAN GOODIN. Google goes down after major bgp mishap routes traffic through china. *ARS Technica*, 2018.

[32] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. Groot: Proactive verification of dns configurations. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 310–328, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. {SCALE}: Automatically finding {RFC} compliance bugs in {DNS} nameservers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 307–323, 2022.

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

[35] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[36] Eduard Kovacs. Bgp flaw can be exploited for prolonged internet outages. *Security Week*, 2023.

[37] Hyojeong Lee, Jeff Seibert, Dylan Fistrovic, Charles Killian, and Cristina Nita-Rotaru. Gatling: Automatic performance attack discovery in large-scale distributed systems. *ACM Trans. Inf. Syst. Secur.*, 17(4), apr 2015.

[38] Jonathan Lee and Jeffrey Shallit. Enumerating regular expressions and their languages. In *Proceedings of the 9th International Conference on Implementation and Application of Automata*, CIAA'04, page 2–22, Berlin, Heidelberg, 2004. Springer-Verlag.

[39] Hongqiang Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *SOSP '17 Proceedings of the 26th Symposium on Operating Systems Principles*, pages 599–613. ACM, October 2017.

[40] Nuno Lopes and Andrey Rybalchenko. Fast bgp simulation of large datacenters. In *VMCAI: Verification, Model Checking, and Abstract Interpretation*, January 2019.

[41] Robert McMillan. Youtube outage underscores big internet problem. *Info World*, 2008.

[42] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.

[43] Sebastian Moss. Verizon bgp route leak causes cloudflare customer outages, aws issues. *Data Center Dynamics*, 2019.

[44] Javier Paris and Thomas Arts. Automatic testing of tcp/ip implementations using quickcheck. In *Proceedings of the 8th ACM SIGPLAN Workshop on Erlang*, pages 83–92, 2009.

[45] Quagga routing software. https://www.nongnu.org/quagga/.

[46] Regular expressions. https://microsoft.github.io/z3guide/docs/theories/Regular%20Expressions/.

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

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

[49] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive

systems with spec explorer. *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, pages 39–76, 2008.

[50] Brandon Vigliarolo. Faa grounds all us departures after notam goes down. *The Register*, 2023.

[51] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Bagpipe: Verified bgp configuration checking. In *Proc. OOPSLA*, 2016.

[52] zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf.