



Guided Exploration of Control Plane Routing States

Conference Paper**Author(s):**

Schneider, Tibor ; Mégret, Jean Bernard David ; Vanbever, Laurent

Publication date:

2025

Permanent link:

<https://doi.org/https://doi.org/10.3929/ethz-c-000783996>

Rights / license:

[In Copyright - Non-Commercial Use Permitted](#)

Funding acknowledgement:

- From Network Verification to Synthesis: Breaking New Ground in Network Automation ()

Guided Exploration of Control-Plane Routing States

Tibor Schneider
ETH Zürich
sctibor@ethz.ch

Jean Mégret
ETH Zürich
megretj@ethz.ch

Laurent Vanbever
ETH Zürich
lvanbever@ethz.ch

Abstract—In recent years, significant progress has been made towards scalable network control-plane verification. Yet, operators are still hesitant to deploy such systems. We argue that this reluctance is in part due to a semantic gap between operators reasoning about routing states and verifiers exploring the space of environments. Indeed, operators express the specification in terms of behavior of routing states, while verifiers usually rely on solvers to find specific environments that violate the specification. This semantic gap prevents users from *guiding* these solvers to directly explore routing states that violate the specification, or to search for states that are most relevant or likely.

In this paper, we present a new approach for flexible control-plane verification. Instead of relying on rigid off-the-shelf solvers, we design a novel backtracking algorithm to directly explore the space of routing states. This enables users to *guide* the exploration according to the specification and domain-specific knowledge from operators. This algorithm paves the way for novel use cases, ranging from finding *relevant* (e.g., likely) counterexamples to performing verification of probabilistic specifications.

Index Terms—Routing protocols, Formal verification

I. INTRODUCTION

Network verifiers have come a long way in the past two decades, greatly improving in terms of the protocols they support, the types of properties they can verify, and the scale of the networks they can practically analyze. While early verifiers could “only” verify reachability properties against static forwarding planes [1] or specific configuration aspects [2], recent verifiers are capable of analyzing both advanced control and data plane properties under arbitrary network environments, i.e., failures and routing inputs [3]–[6].

In contrast to this rapid progress, the actual deployment of control-plane verification has been comparatively slower. While multiple reasons explain this slow adoption, we attribute it to the fact that verification systems remain difficult to use [7]. Some related works blame this on verifiers returning *single* counterexamples that (i) contain little context about the cause of the problem [8], [9], and (ii) often describe unlikely corner cases, even in the presence of a critical misconfiguration [7]. But this only scratches the surface.

Fundamentally, there is a semantic gap between verifiers exploring the space of environments and operators reasoning about the network’s routing state. These two viewpoints essentially correspond to the inputs and outputs of the distributed computation of the network: the convergence process turns environments into routing states. This convergence process separates the verifiers’ explorations from the user’s intent, i.e., the specification expressed as undesired routing behaviors.

Thus, verifiers cannot find those counterexamples with the most practical relevance to the users—they would benefit from *guiding* the exploration process themselves to find the relevant, e.g., most likely, counterexamples.

Most existing control-plane verifiers rely on the use of generic SMT-based solvers to explore the space of network environments [3], [4], [10]. They generate SMT constraints that relate symbolic variables describing the environments with others describing the routing state. These verifiers leave it up to the solver to explore the joint space of environments and routing states, which often finds inactionable counterexamples [8]. While SMT solvers like Z3 offer sophisticated languages to declare and combine search tactics to influence their exploration [11], designing the right tactics is notoriously challenging [12]–[14]. It is, thus, infeasible for operators to guide the solver’s exploration by designing custom tactics.

Some recently proposed network verifiers symbolically simulate the route propagation and selection with Binary Decision Diagrams (BDDs) [5], [15]. In doing so, they operate on the *entire* space of environments to describe *all* routing states at the same time. Thus, they can yield the entire space of counterexamples. However, this space is defined in terms of the environments instead of routing states. This mismatch still leaves users guessing at what routing states to focus on first. Moreover, any attempt at guiding the symbolic execution of these verifiers might yield incorrect results.

In short, the semantic gap between operators reasoning about routing states and verifiers operating on the space of environments prevents users from *guiding* the exploration according to their high-level intent. Yet, as we show, doing so can significantly improve the usability of these verifiers, enabling them to: (i) find interesting, e.g., likely, counterexamples first; and to (ii) prune parts of the search space that provably do not contain any counterexamples.

To bridge this gap, we propose a technique to directly explore the space of routing states, that is, which routes are selected in the network. Specifically, we introduce the *BGP State Iterator*, a novel backtracking algorithm that explores the space of routing states, guided directly by the specification, domain-specific knowledge, or a probabilistic model. Unlike existing approaches, the *BGP State Iterator* can: (i) find the entire set of environments that trigger a violation; (ii) explore the important, e.g. likely, routing states first; and (iii) perform specification-guided exploration.

Turning the problem around by directly exploring the large space of routing states is challenging though, mainly because

the vast majority of states are simply incoherent. From the large space of environments, the network will, in fact, only converge to a tiny subspace of all routing states. The problem now becomes synthesizing the set of environments that result in each state, and if this set is empty, then the state is unreachable. To that end, we explore the space of routing states hierarchically. The *BGP State Iterator* defines a tree of *partial* routing states, that is, states in which the selected routes of some routers are yet undefined. We find that, in most cases, only a few selected routes suffice to prove that the network cannot converge to selecting those routes. Our backtracking algorithm thus relies on sufficient conditions on partial routing states to be unreachable. As we show, these conditions are (i) effective in pruning the search space; (ii) efficient to test, using simple set operations; and (iii) become both necessary and sufficient conditions once the algorithm reaches a leaf. Our algorithm is thus sound and complete, finding *all* reachable routing states, including a description of the entire space of environments that result in the corresponding state.

Contributions: We fully implement the *BGP State Iterator*, evaluate it on realistic configurations, and release it under a GPLv2 license. In our evaluation, we demonstrate how network verification benefits from specification-guided exploration. In particular, we find the space of all counterexamples in less than a second, when SMT-based verifiers such as [10] would timeout after a day. Beyond this, we outline how, given a model that describes the likelihood of observing specific routing inputs and failures, the *BGP State Iterator* can explore the most likely routing states first. In doing so, we lay the foundations for probabilistic control-plane verification—verifying that properties are satisfied with at least probability p —on failures *and* routing inputs, something which was previously only possible under node and link failures [16], [17].

II. OVERVIEW

The *BGP State Iterator* explores the space of BGP routing states using a backtracking algorithm. It finds all stable routing states of the network and yields them one by one. The iterator also generates conditions on the environment, i.e., on advertised routes and failures, for the network to converge to that state. These conditions describe a space of concrete environments and thus form an equivalence class.

We define a *partial* routing state as an assignment of selected routes to a subset of routers. Partial routing states thus describe a set of *concrete* routing states, which specify the selected routes of all routers. The *BGP State Iterator* explores the tree of such partial routing states; the root contains no selected routes, while all leaves are concrete states.

Concrete routing states are either *stable*, i.e., there exists an environment that leads the network to converge to that state, or *unstable*. We call a partial state *unstable* if it only describes unstable leaves. By detecting unstable states early during the exploration, we massively reduce the size of the tree that we traverse. To that end, we define a simple yet expressive language for describing route maps. This language enables us to devise efficient heuristics for checking whether a

(partial) routing state is unstable using efficient set operations. These heuristics are necessary but insufficient to discriminate unstable states, but they become sufficient as the iterator reaches a leaf—we only yield stable states.

Checking for the existence of a stable state in BGP-driven networks is proven *NP-Complete* [18]. Yet, it can be done using BDDs, and only becomes challenging for complex route maps on iBGP sessions, which is generally discouraged [19]. For most realistic networks, it stays easy to check.

Tree Exploration and Use Cases: By designing our own backtracking algorithm, we allow users to guide the exploration and efficiently solve a wide range of network verification problems. In fact, the *BGP State Iterator* allows users to specify branches that should be explored first and others to skip entirely. This benefits users in the following ways:

We can improve the scalability of traditional network verification by guiding the exploration according to the specification, that is, only explore states that could lead to a violation. For instance, to ensure a customer’s route is preferred over the one from a provider, we only explore the states in which that customer advertises a route, but some routers still select the route from the provider. In Section VI, we demonstrate how the *BGP State Iterator* can find *all* violations of a specification within one second, while *Minesweeper* times out after one day.

By exploring the tree of routing states in a depth-first manner, we minimize the memory usage and the time to yield the first state. Alternatively, we can employ more general guided search techniques, exploring those branches that the user is most interested in first. For instance, the operator might want to explore the routing inputs that differ least from the ones currently observed first. Likewise, given a probabilistic model that describes the likelihood of observing routing inputs satisfying some constraints. The *BGP State Iterator* could prioritize the branches with the highest likelihood and find the most likely counterexamples first. This allows operators to ensure service level agreements (SLAs) are met (e.g. reachability is satisfied for 99.999% of the cases). They may run the *BGP State Iterator* until the probability of the yielded states satisfying the specification covers the required probability.

Finding Stable Routing States: The size of the routing state tree grows exponentially in the number of routers. In a network where the n routers may choose among r routes, the number of routing states grows $\mathcal{O}(r^n)$. Yet, only a small subset of these states is *stable*. As the *BGP State Iterator* traverses the tree, it uses effective heuristics to prune unstable (partial) routing states. More precisely, a router selecting a route implies that:

- (i) *Propagation:* the neighbor advertises that route,
- (ii) *Advertisement:* the route contains the correct attributes,
- (iii) *Preference:* the route is the most preferred route available.

As a result, the selection of a route massively restricts the stable choices of other routers. For instance, all routers along a propagation path select the same route (Propagation). Likewise, any other neighboring router cannot advertise a more preferred route, preventing that neighbor from selecting it in the first place (Preference).

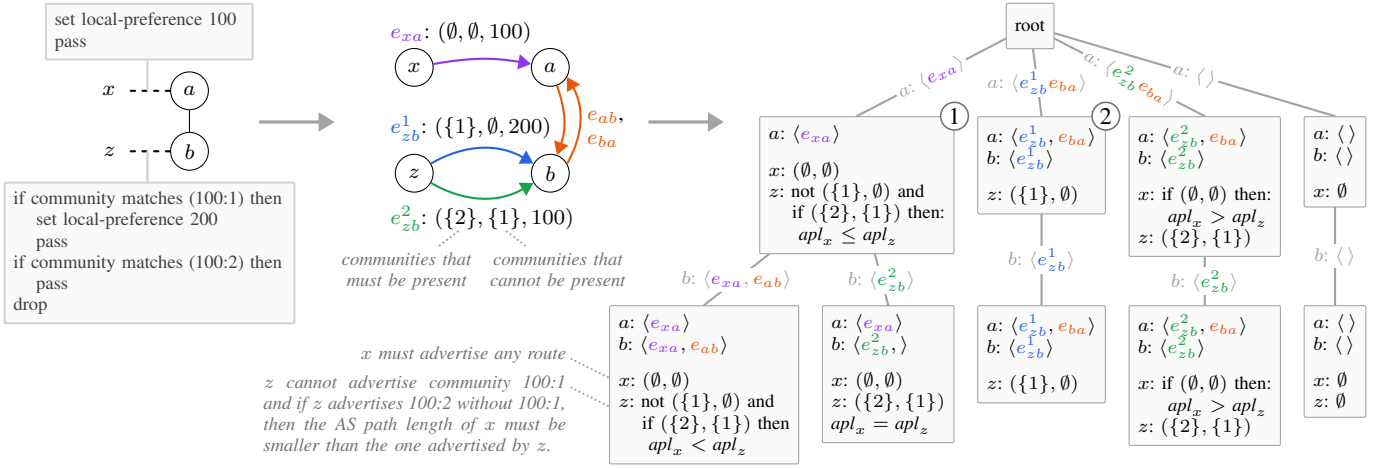


Fig. 1: The *BGP State Iterator* first transforms a BGP configuration (on the top left) to a directed multi-graph (center-left). Each edge represents one possible execution path of the configured route maps. The iterator then builds and explores the tree of routing states (on the right). The first level corresponds to all possible route choices of router a , and the second corresponds to all remaining options for b . We only show states that are stable, i.e., which the *BGP State Iterator* actually explores. Each state contains both selected paths and conditions on the environment, i.e., on the advertised attributes from x and z .

Example: Fig. 1 depicts an example network consisting of two routers, a and b connected to two external networks, x and z . The *BGP State Iterator* first constructs the BGP graph to capture the BGP configuration. This BGP graph is a multi-graph in which each edge corresponds to one execution path of a route map. Consider the route map on the session $z \rightarrow b$. A route with community 100:1 is accepted with local-preference 200. We capture this as an edge e_{zb}^1 , labelled with $(\{1\}, \emptyset, 200)$. Here, the first set $\{1\}$ describes all communities (omitting the 100: prefix) that must be present, while the second set describes those that cannot be present (none in this case). A route without community 100:1 but with 100:2 is accepted without changing the local-preference, resulting in edge $e_{zb}^2: (\{2\}, \{1\}, 100)$. All other routes that z advertises to b are discarded. The route map from x to a results in a single edge $e_{xa}: (\emptyset, \emptyset, 100)$. The iBGP session between a and b simply accepts all routes without transformations.

We then construct the tree of routing states, shown on the right side of Fig. 1. The first level corresponds to all possible routes selected by router a . We capture those routes in terms of paths in the BGP graph: a can choose a path that starts at an external network and ends in a , that is, $\langle e_{xa} \rangle$, $\langle e_{zb}^1, e_{ba} \rangle$, $\langle e_{zb}^2, e_{ba} \rangle$, and the empty route $\langle \rangle$. As the iterator explores one such path, it constructs conditions for a to select that route in three steps: *Propagation*, *Advertisement*, and *Preference*.

During *Propagation*, we ensure all routers along the path select the same path. Consider state ②. If a selects the path $\langle e_{zb}^1, e_{ba} \rangle$, then b must also select $\langle e_{zb}^1 \rangle$ to propagate $\langle e_{zb}^1, e_{ba} \rangle$ to a . In contrast, for state ①, a selecting $\langle e_{xa} \rangle$ does not yet restrict the selected route of b as it is learnt directly from x .

During *Advertisement*, we ensure that the external networks advertise the correct attributes, i.e., communities. Again, consider state ①. The path $\langle e_{xa} \rangle$ has no configured route maps, and thus, the only condition is that x advertises any route with

arbitrary attributes. To select $\langle e_{zb}^1, e_{ba} \rangle$ in ②, z must advertise a route with community 100:1. We write this condition as a tuple $(\{1\}, \emptyset)$; the first set describes all communities that must be present, and the second contains those that must be absent.

During the last *Preference* step, we ensure any other available route is less preferred than the selected one. State ② does not restrict advertisements from x ; the path $\langle e_{zb}^1, e_{ba} \rangle$ is always preferred due to its higher local-preference. In contrast, for a to select $\langle e_{xa} \rangle$ in state ①, it cannot receive a route from b that is more preferred. To that end, the *BGP State Iterator* generates two conditions. First, it requires that $\langle e_{zb}^1, e_{ba} \rangle$ is not available. Second, if $\langle e_{zb}^2, e_{ba} \rangle$ is available, then the AS path length of x is smaller or equal to the one from z , i.e., $apl_x \leq apl_z$. We allow equality as routes learned from eBGP are preferred over those from iBGP with equal path lengths.

After deciding on a route selected by a , the iterator explores all remaining options for b . From state ①, b can either select $\langle e_{xa}, e_{ab} \rangle$, but only if $apl_x < apl_z$, or $\langle e_{zb}^2 \rangle$, which requires that the two AS path lengths are equal. The other options for b result in unstable states. Selecting $\langle e_{zb}^1 \rangle$ is prohibited by the conditions inherited from ①, while selecting no route, i.e., $\langle \rangle$, fails the Preference condition as b receives $\langle e_{xa}, e_{ab} \rangle$.

With this example in mind, the remainder of this paper will be structured as follows. We first introduce a model of BGP routes, route maps, and routing states, all of which greatly impact the semantics of the iterator (Section III). We then describe the backtracking algorithm, including our heuristics of determining unstable routing states (Section IV). Next, we discuss compatible specifications, while acknowledging properties that the *BGP State Iterator* cannot verify (Section V). Finally, we evaluate the runtime of the *BGP State Iterator* and investigate the impact of guiding the exploration based on a specification (Section VI).

III. MODEL OF BGP

We model the BGP propagation graph as a directional multi-graph $G_{BGP} = (N, E_{BGP})$ to capture how routes propagate over BGP sessions. The BGP propagation graph is an overlay signaling graph of the underlying physical topology, so BGP sessions might not correspond to physical links. This graph describes both BGP sessions and all transformations applied to propagated routes, i.e., route maps and implicit rules from the BGP protocol itself. We use a multi-graph to express each possible execution path of a route map as a distinct edge. We further separate $N = N_i \cup N_x$ into internal routers N_i for which the configuration is known, and external networks N_x that may advertise an arbitrary route for the destination d .

In this work, we focus on three important BGP attributes: the *AS path length*, the *local-preference*, and the *community set*. While BGP routes carry additional attributes, these three are representatives of all other attributes. Our model can be extended to support remaining attributes, either by following similar ideas or by encoding them using special communities. An example of the former would be to include MED values akin to how the AS path lengths are stored and processed, however with a few additional semantics. For the latter, if some routes are denied due to the presence of certain AS numbers on the AS path, or because of some IP prefix access control list, we may encode the presence of such AS numbers or IP prefix ranges through special communities.

Route Maps and Transfer Functions: We introduce a simple and canonical language for expressing route maps and other transformations that are implied by the BGP protocol itself. The language captures all possible execution paths of these transformations. We call each such execution path a *transfer function* and use them as edge labels in the BGP graph G_{BGP} .

Let $tf_e = (m_e^\pm, s_e^\pm, lp_e)$ be the transfer function of edge $e \in E_{BGP}$. It encodes match conditions, i.e., $m_e^\pm = (m_e^+, m_e^-)$, and actions, i.e., $s_e^\pm = (s_e^+, s_e^-)$ and lp_e . A statement matches a route if it contains all communities in the set m_e^+ and none of m_e^- . If it does match, the communities in the set s_e^+ are added, while those in s_e^- are removed, and the local-preference lp_i is assigned (or left unchanged if $lp_i = \emptyset$). We require all edges from one router to another to have disjoint transfer functions, i.e., a route can only propagate over one edge from any router to another. More formally, for any two distinct edges a, b from router u to v , we require that

$$(m_a^+ \cap m_b^-) \cup (m_a^- \cap m_b^+) \neq \emptyset.$$

Consequently, the order of transfer functions is irrelevant. If no statement matches, the route is denied.

Let a path $p = \langle e_0, e_1, \dots, e_{k-1}, e_k \rangle \in \mathcal{P}$ be a propagation path in graph G_{BGP} that propagates a route from external network $src(p) = src(e_0)$ to internal router $dst(p) = dst(e_k)$. We call a path *valid* if (i) it complies with BGP propagation rules, e.g., routers do not propagate routes learned from an iBGP peer to another one, and (ii) there exists a set of route-attributes that would match all transfer functions consecutively. We write $tf_p = tf_{e_k} \circ \dots \circ tf_{e_1} \circ tf_{e_0}$ as the composition

of all transfer functions along path p . Finally, we denote $pre(p) = \langle e_0, \dots, e_{k-1} \rangle$ as the prefix of path p without e_k .

Let $p = \langle a, b \rangle$ be such a path. This path is invalid if the action s_a^\pm of a contradicts with the match m_b^\mp of b , that is, if $s_a^+ \cap m_b^- \neq \emptyset$ or $s_a^- \cap m_b^+ \neq \emptyset$. If the path is valid, the resulting transfer function $tf_p = tf_b \circ tf_a$ of path p is:

$$\begin{aligned} m_p^+ &= m_a^+ \cup (m_b^+ \setminus s_a^+), & m_p^- &= m_a^- \cup (m_b^- \setminus s_a^-), \\ s_p^+ &= s_a^+ \setminus s_b^- \cup s_b^+, & s_p^- &= s_a^- \setminus s_b^+ \cup s_b^-, & lp_p &= lp_b. \end{aligned}$$

BGP Routes and Attributes: Any external network $x \in N_x$ may advertise a route for destination d . We model the attributes of these routes using a symbolic representation, describing a set of possible attributes at the same time. We write the attributes that x advertises as $att(x) = (c_x^\pm, c_x^-, apl_x)$. It is a tuple containing two sets of communities: c^+ those that must be present, c^- those that must be absent, as well as the AS path length apl_x . Any community outside $c^+ \cup c^-$ is not constrained and may or may not be present without influencing any routing decision. Therefore c_x^\pm is symbolic and describes a set of community sets. If x does not advertise a route for destination d , we write $att(x) = \emptyset$.

We write $c_x^\pm \models m_p^\pm$ if all potential community sets described by c_x^\pm match the conditions m_p^\mp of p 's transfer function tf_p . Likewise, we write $c_x^\pm \models \neg m_p^\pm$ if no community set described by c_x^\pm matches m_p^\pm . Otherwise, some but not all community sets match m_p^\pm , which we denote as $c_x^\pm \diamond m_p^\pm$:

$$\begin{aligned} c_x^\pm \models m_p^\pm &\Leftrightarrow \underbrace{c_x^+ \cap m_p^- = c_x^- \cap m_p^+ = \emptyset}_{\text{no contradiction}} \wedge \underbrace{m_p^+ \subseteq c_x^+ \wedge m_p^- \subseteq c_x^-}_{c_x^\pm \text{ captures } m_p^\pm} \\ c_x^\pm \diamond m_p^\pm &\Leftrightarrow \underbrace{c_x^+ \cap m_p^- = c_x^- \cap m_p^+ = \emptyset}_{\text{no contradiction}} \wedge \underbrace{m_p^+ \not\subseteq c_x^+ \vee m_p^- \not\subseteq c_x^-}_{c_x^\pm \text{ does not capture } m_p^\pm} \\ c_x^\pm \models \neg m_p^\pm &\Leftrightarrow \underbrace{c_x^+ \cap m_p^- \neq \emptyset \vee c_x^- \cap m_p^+ \neq \emptyset}_{c_x^\pm \text{ contradicts } m_p^\pm} \end{aligned}$$

In contrast to external advertisements, we do not represent the attributes of routes selected within the network. Instead, we identify them as a propagation path p . We say node $v = dst(p)$ selects path p learnt from neighbor $u = dst(pre(p))$ and advertised by the external network $x = src(p)$. To reconstruct the v 's selected attributes, we apply tf_p on $att(x)$.

IV. BACKTRACKING ALGORITHM

The core of the *BGP State Iterator* is a backtracking algorithm. The following explains the construction and traversal of the search tree and heuristics to prune it.

A. Routing State Tree

The *BGP State Iterator* explores a tree of routing states; leaves are concrete states and all others are partial states. We define a concrete (or partial) routing state $S : N_i \rightarrow \mathcal{P}$ as a (partial) function that maps routers to their selected route, that is, a selected propagation path in G_{BGP} . In this paper, we write $S(v) = p$ and $p \in S$ interchangeably. For any router $v \in N_i$, we define $\mathcal{P}(v)$ as the set of paths in G_{BGP} that start at any

external network N_x and end in v . We also add the empty path $\langle \rangle \in \mathcal{P}(v)$ to indicate the absence of a route.

We construct a tree of (partial) routing states with a depth of $|N_i| + 1$ as follows: Given an ordering $[v_0, v_1, \dots]$ of routers, a partial state at depth k contains a routing state mapping S that is defined for all nodes up to, and including, v_{k-1} . The root at level 0 has an empty state mapping, while leaves at level $|N_i|$ are concrete routing states with fully defined state mappings. We provide more details on how the router order may affect the algorithm's efficiency in Section IV-E.

B. Stable Routing States

The constructed tree of routing states is enormous, growing exponentially with the size of the BGP propagation graph. Yet, we only need to explore *stable* states—the network only converges to a small subset of possible states. We define three conditions to capture stable routing states: *Propagation*, *Advertisement*, and *Preference*. Propagation requires the neighbor to propagate the selected path. Advertisement ensures the existence of matching routing inputs (attributes). Preference enforces each selected route to be preferred over all that are available. Formally, for path $p \in S$ to be selected:

$$\text{Propagation } pre(p) \in S \quad (1)$$

$$\text{Advertisement } c_x^\pm \models m_p^\pm \quad (2)$$

$$\text{Preference } \forall q \neq p : pre(q) \in S \wedge c_x^\pm \models m_q^\pm \rightarrow p \prec q \quad (3)$$

$p \prec q$ in Equation (3) requires p to be preferred over any other available path q . The total order relation \prec follows BGP's route-selection algorithm [19]. Our prototype implementation compares the local-preference, AS path length, and IGP cost (breaking ties based on neighboring router ID).

The three conditions in Equations (1) to (3) are necessary and sufficient for any concrete routing state to be stable, i.e., such that there exists a set of attributes att in which the network may converge to that state [18].

C. Heuristics on Partial States

Equations (1) to (3) can be applied on partial states as well by relaxing (ignoring) the conditions on undefined states. The remaining relaxed conditions become necessary but not sufficient for a partial routing state to contain at least one stable leaf, i.e., not to be unstable. Partial states that violate the relaxed conditions are proven unstable and can be skipped during exploration, massively reducing the search space.

For each state that the *BGP State Iterator* explores, it performs three steps which correspond to the Propagate, Advertise, and Prefer conditions from Section IV-B. In doing so, it accumulates a set of restrictions associated with each partial state and all its descendants. There are three kinds of restrictions: (i) AS path length relations, (ii) routes that cannot be selected, and (iii) routes whose selection imply AS path length relations. Once the iterator reaches a leaf, the accumulated restrictions are equivalent to the conditions from Equations (1) to (3). Therefore, the iterator finds *all stable* routing states, making it both sound and complete.

In the following, we consider exploring the state in which v is assigned to select route $p = \langle e_0, e_1, \dots, e_k \rangle$ from external network $x = src(p)$, advertised by router $u = src(e_k)$.

Propagation: For each sub-path $p_i = \langle e_0, e_1, \dots, e_i \rangle$ of path p with $0 < i \leq k$, we assign $S(n) \leftarrow p_i$ where $n = dst(p_i)$. We then ensure that $S(n)$ is not already set to a different route, and that all restrictions collected before are satisfied. If any restriction for any p_i is violated, the state is unstable and skipped. The additional restrictions are constructed in the subsequent steps.

Advertisement: In the second phase, we ensure there exists a set of attributes $att(x)$ advertised by x , which results in the route p . First, we extend the communities advertised by x to match tf_p : $c_x^+ \leftarrow c_x^+ \cup m_p^+$ and $c_x^- \leftarrow c_x^- \cup m_p^-$. We then check that $c_x^+ \cap c_x^- = \emptyset$. If the two sets overlap, then no such community set can exist, and v cannot select p .

Preference: For each path $q = \langle e'_0, \dots, e'_m \rangle \in \mathcal{P}(v) \setminus p$ learnt from a different neighbor $w = src(e'_m) \neq u$, we ensure that the route is either not received, or that it has a lower preference, i.e., $p \prec q$. We distinguish two cases. (i) If the information in the partial routing state is sufficient to know that v cannot receive q , we can safely ignore route q . Concretely, router v cannot receive q if either $S(w)$ is defined and $S(w) \neq pre(q)$, or if $c_y^\pm \models \neg m_q^\pm$, where $y = src(q)$ is the external network. (ii) Otherwise, we must ensure that $p \prec q$ when v receives q to ensure v selects p over q . Depending on the local-preferences of p and q , we generate different constraints:

- $lp_p > lp_q$ implies v prefers p over q , so we can ignore q .
- $lp_p < lp_q$ implies that v prefers q over p . We must ensure that v will not receive q , thus adding the restriction: $pre(q) \in S \implies c_y^\pm \models \neg m_q^\pm$.
- $lp_p = lp_q$ implies a relation on the AS path length of the routes from x and y . Thus, we restrict $pre(q) \in S \wedge c_y^\pm \models m_q^\pm \implies apl_x < apl_y$. We allow equality if the IGP cost of p is smaller than the one of q (including the tie-break).

When considering link failures, the *BGP State Iterator* will further fork the state based on the comparison of IGP costs between p and q , see Section IV-D. Note, that we do not propagate the selection condition to all sub-paths p_i of p . These conditions will be checked when exploring $dst(e_i)$.

As soon as $pre(q) \in S$ is selected in this or in any subsequent routing state, we add all restrictions to the model associated with selecting $pre(q)$. This either forbids a certain set of communities to be advertised, i.e., $c_y^\pm \models \neg m_q^\pm$, or force the advertisement of such communities to yield a relation of the AS path lengths. For the latter, we fork the current state into two, one in which $apl_x < apl_y$, i.e., following the AS path relation, and one in which $apl_x \geq apl_y \wedge c_y^\pm \models \neg m_q^\pm$, i.e., where v does not receive q .

The *BGP State Iterator* stores AS path relations as a directed graph, with edges from one path length to another that must be larger. The presence of a cycle indicates an unstable state, unless all relations on the cycle permit equality.

Likewise, the *BGP State Iterator* stores all match restrictions m_q^\pm that advertised attributes cannot match. The currently

known advertised communities c_y^\pm relates to such a restriction in three different ways:

- $c_y^\pm \models m_q^\pm$, i.e., the current communities already match m_q^\pm . This implies the current partial state is unstable.
- $c_y^\pm \models \neg m_q^\pm$, i.e., the communities already contradict the match m_q^\pm . We thus ignore this restrictions.
- $c_y^\pm \diamond m_q^\pm$, i.e., c_y^\pm might match m_q^\pm . We collect all such restrictions m_q^\pm and use a BDD to check if they “cover” the entire space of c_y^\pm , i.e., is there a concrete community set in c_x^\pm that matches none of m_q^\pm .

Therefore, in theory, the *BGP State Iterator* must solve a SAT problem for each explored partial state. This problem of checking whether a neighboring AS can advertise a route with attributes distinct from already rejected ones (m_q^\pm) scales with the complexity of the configured route maps. Fortunately, these are usually either well structured and redundant or inexistent, as most operators do not configure route maps on iBGP sessions, i.e., $tf_q = tf_{pre(q)}$. Thus, $pre(q) \in S$ usually implies $c_y^\pm \models m_q^\pm$, reducing most restrictions to simple and efficient set operations.

D. Link Failures

Link failures affect the BGP routing state indirectly in two ways. First, failures can partition the network and disconnect BGP neighbors. Second, failures impact the IGP cost, and thus, the route selection process of BGP.

Prior to the exploration, we compute the set of all failure scenarios with up to k link failures. Similar to related works, we only consider the failures that affect IGP costs and paths [16]. During the exploration, the *BGP State Iterator* updates this set of failure scenarios to contain only scenarios that could lead to the current state. When selecting a new route in the Propagation step, we remove all failure scenarios that would disconnect the two endpoints of any edge in the path. During the Preference step, when generating the relations on the AS path length, we further check if there exists failure scenarios that make path p have a smaller and larger IGP cost than q . If both exist, we fork the current state into two; one containing all failures in which p has a smaller IGP cost than q , and one in which the opposite holds.

E. Exploration Order

We build the routing state tree by defining an order of routers $[v_0, v_1, \dots]$; Router v_{k-1} is considered when exploring a state at depth k . This order impacts the *BGP State Iterator*’s performance. Our heuristics for pruning unstable partial states, i.e., for checking that the partial state only contains unstable leaves, are sufficient but not necessary. The iterator might explore unstable partial states until it eventually proves them unstable. A suboptimal order of routers can cause more or less unnecessary explorations of such unstable partial states.

Our heuristics are not necessary due to the restrictions generated during the Preference step: they only check the selected route of the direct neighbor of v on a path p , say u , but not all other nodes along p . Extending the restrictions

to check the selection on all other nodes would be far from trivial, as it introduces a long chain of logical implications.

Counterintuitively, we find that we increase such situations when exploring routers from the network’s edge inwards. Instead, it is beneficial to explore “central” routers, e.g., route reflectors, first. This is because if a node $m \in p$, say a border router, is processed rather than the direct neighbor u , say a route reflector, then our restrictions for v to select p only apply to u , which is not yet constrained.

V. SUPPORTED SPECIFICATIONS

The flexibility of our backtracking algorithm enables us to reason about a wide range of networking properties and specifications. In the following, we describe properties that the *BGP State Iterator* can verify and outline how it can do so. We also acknowledge properties that our backtracking algorithm cannot reason about. Finally, we describe how our algorithm enables probabilistic verification of control-plane properties. We split the space of specifications into properties on the routing and on forwarding states.

Routing Properties: These properties relate to the routing state of the network. We follow ideas from Tang et al. [3] who distinguish *safety* and *liveness* properties. Intuitively, safety properties require that “bad” can never be selected, while liveness properties require that “good” routes will eventually be selected. Safety properties, such as ensuring that no route from one provider is propagated to another provider, must hold in all states, including *transient* states during BGP’s convergence process. The *BGP State Iterator* verifies safety properties by building the BGP graph and exploring propagation paths. Indeed, if there exists a valid propagation path from one provider to another, there does exist a sequence of BGP messages exchanged that causes a route leak.

In contrast, liveness properties pertain to selected routes in the stable state. For instance, ensuring that routes from a customer are preferred over those from a provider entails proving that no stable state exists in which the provider’s route is selected while the customer advertises a route for the same prefix. To prove such liveness properties (or to find counterexamples), the *BGP State Iterator* can explore only the partial states in which the border router adjacent to the provider selects the provider’s route, while the customer advertises one. By further picking a beneficial router order to explore the two border routers first, the *BGP State Iterator* only explores a tiny space of the tree and quickly finds all counterexamples, as we describe in Section VI-C.

Forwarding Properties: Such properties apply to all possible forwarding path. The *BGP State Iterator* can check properties on forwarding paths by iterating over all stable states, and simulating the network in all such states. Our backtracking algorithm can only be guided by routing properties—forwarding properties often depend on the selected route of many (if not all) routers in the network. Yet, even for large networks and complex configurations, we show in Section VI-B that the number of stable states remains manageable, especially for recent and efficient network simulators [20]–[22].

The backtracking algorithm explores *all* stable states. It cannot find routing inputs that cannot converge or reason about *transient* states during convergence. While the former has been extensively studied in the literature [18], [23], the latter remains an open research problem. Some systems measure BGP convergence and how this affects packet loss and other QoS metrics [24]–[27]. Others perform planned reconfigurations without disturbing traffic during convergence [28]–[31].

Similar to most recent control-plane verifiers (including [3], [10], [15], [32], [33]), the *BGP State Iterator* is limited to analyzing routes one prefix at a time. However, doing so can overlook misconfigurations caused by longest-prefix matching (LPM), where some router along a forwarding path uses a more or less specific prefix. *Expresso* [5] verifies such situations by considering 33 distinct routes simultaneously, one for each IPv4 prefix length. Iterating over multiple selected routes, that is, for each prefix, simply causes the search space to explode [5], [16]. Yet, we notice that, fundamentally, such problems only arise when a route can be propagated only to a few routers, causing all others to use a less specific route. In such a case, a router along a path might drop a packet or create a forwarding loop. Therefore, to detect potential issues caused by LPM and instead of considering multiple prefixes simultaneously, the *BGP State Iterator* checks for all stable states in which only a subset of routers select a route.

Likewise, some systems verify link load properties in the network [6], [34], [35]. Doing so, however, requires to reason about all routes simultaneously—BGP currently distributes routes for over 1 million different prefixes. While the *BGP State Iterator* can reason about local forwarding properties such as load balancing, it cannot guarantee that the network remains free of congestion given an interdomain traffic matrix.

Probabilistic Verification: Probabilistic verification enables the relaxation of specifications, e.g., to maintain guarantees in 95% of scenarios according to SLAs. Our backtracking algorithm opens the door for verifying such properties for both failures and routing inputs. Such specifications involve proving that certain properties are satisfied for at least some probability p . Given a probabilistic model of the environment, the iterator can yield states ordered by their likelihood. We then explore the tree until we observed either positive examples that cover p or negative ones that cover $1 - p$.

VI. EVALUATION

In this section, we evaluate the scalability of the *BGP State Iterator*. First, we investigate how its running time relates to the network size, number of external networks, and configuration—we find *all* stable states of large networks and complex configurations *within hours* at most (recall that existing verifiers only aim at finding *one* of these). Next, we analyze the effectiveness of specification-guided exploration of the state space—we find all counterexamples in *under a second*, exploring less than 0.1% of the space compared to full exploration, while *Minesweeper* [10] times out after 1 day. Finally, we examine how changing the router ordering affects

the effectiveness of our heuristics—exploring route reflectors first reduces the number of states explored by up to $10\times$.

Implementation: We implement a prototype of the *BGP State Iterator* in $\approx 7\,000$ lines of Rust code, including the graph representation of the BGP configuration and the backtracking algorithm available at <https://github.com/nsg-ethz/BGP-state-iterator>. We run all experiments on a server with 96 CPU cores and 256 GB of memory, using a single core per experiment.

Methodology: We use 160 different topologies from Topology Zoo [36], ranging from 20 to 197 routers, with up to 243 links. For each topology, we randomly connect external networks to routers, randomly split into customers, peers, and providers. Though our implementation includes parsing and translation functions for real-world configurations, we only evaluate on synthetic configurations. Specifically, we configure the network to implement the common Gao-Rexford routing policies [37] to make routes from customers preferred over those from peers and providers. By default, we configure the network in an iBGP full-mesh. We use this configuration as a starting point for all experiments, while introducing more complex policies in §VI-B.

We then measure the time to explore the entire tree of routing states to find *all* stable states. A full exploration constitutes the worst-case running time of the algorithm, as finding a few counterexamples is usually sufficient. We also count the number of partial states that the iterator explores. We repeat each experiment 100 times and report the median, as well as the 5th and 95th percentiles.

A. Network Size

The network topology influences the *BGP State Iterator*’s running time by affecting the size of the routing state tree that it explores. The number of routers in the network increases the depth of the search tree. Adding more external routers increases the number of possible routes, and thus the number of leaves of each node and the width of the tree. Finally, the number of links impacts the number of failure scenarios to consider, and thus, how expensive each exploration step is.

Number of Routers: In the first set of experiments, we measure the effect of the number of routers on the algorithm’s running time by enumerating all stable routing states of all 160 topologies, each with 10 external networks. We show the results in Fig. 2 (notice the logarithmic y -axis). Even for large networks of almost 200 routers, the *BGP State Iterator* can find *all* stable routing states in less than five seconds (as shown in the top plot).

The number of states explored is shown in the lower part of Fig. 2 and remains below 10^6 . To put this number into perspective, let us consider the theoretical maximum: the tree of routing states may have as much as $\sum_{k=0}^n r^k$ nodes, for n routers and r possible routes. For a network with 100 routers and 10 external networks, the tree already becomes larger than 10^{100} . Yet, this is a gross overapproximation. Once we reach a partial state in which all border routers have selected a route, all others have only one valid choice of a selected route. An

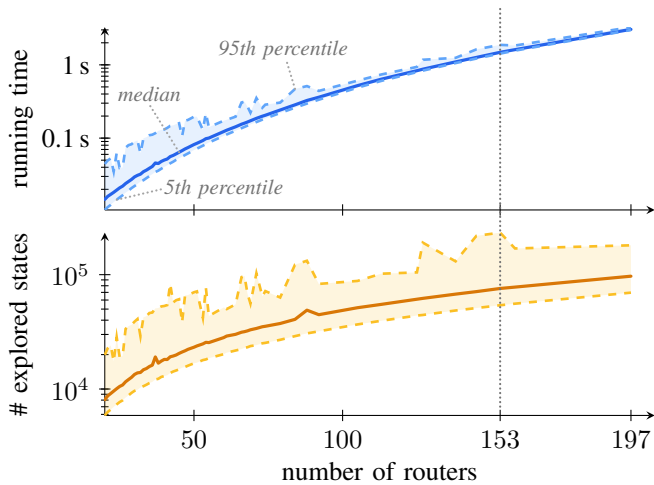


Fig. 2: Larger networks make the *BGP State Iterator* explore more states, and thus, increase its running time. The blue line in the top plot shows the running time, while the orange line in the bottom plot counts the number of states explored. The plot shows the median and the 5th and 95th percentiles.

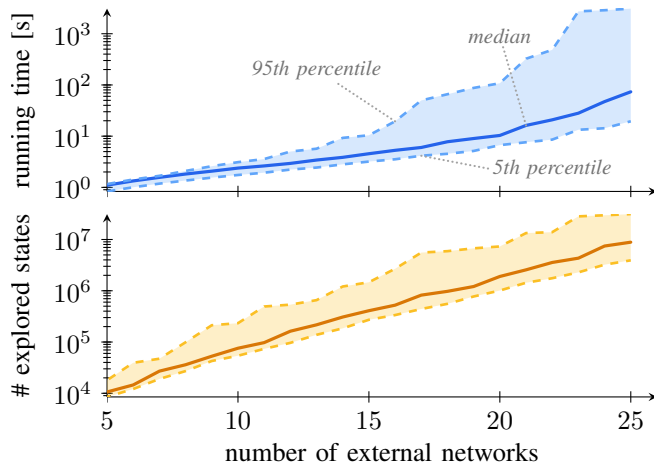


Fig. 3: The running time of the *BGP State Iterator* significantly depends on the number of external networks. The blue line at the top shows the running time and the orange line at the bottom counts all explored states.

upper bound for the effective tree size is $(n-b) \cdot r^b + \sum_{k=0}^b r^k$, in which all b border routers have r possible routes to choose from, while all $n-b$ remaining ones have only one. This yields a tree size of over 10^{12} nodes for networks with 100 routers and 10 external peers, far above the 10^5 states that we explore.

Number of External Networks: We take a single topology, *Colt*, with 153 routers and 177 bidirectional links, and vary the number of external networks from 5 to 25. Figure 3 shows the *BGP State Iterator*'s running time (above) and number of states explored (below) using a logarithmic y -axis.

Both the running time and the number of explored states grow exponentially in the number of external networks. Indeed, introducing more eBGP sessions with new neighbors

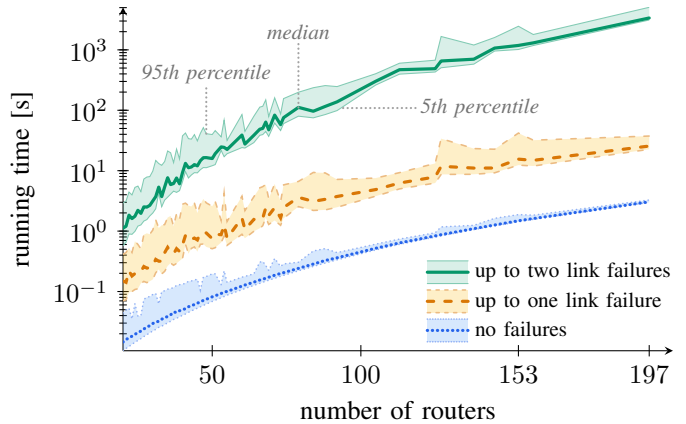


Fig. 4: The *BGP State Iterator* can find all states with up to two link failures within one hour. The green solid line shows the running time when considering up to two simultaneous link failures. The orange dashed line shows the same for a single failure, and the blue dotted line does so for no failures.

increases the number of possible routes and egress combinations in stable states. Despite that, we usually find *all* stable states within a couple of minutes. In the worst case, it takes the iterator up to 10 hours to explore the entire space.

The large variance can be attributed to the random assignment of the roles of the external networks (customer, peer, or provider). Routes from networks of the same role have the same local-preference, and hence, a stable state can select routes from any combination of networks of the same role. An even distribution of the roles results in fewer stable states, while a more skewed distribution yields many more.

Link Failures: Exploring the space of routing inputs and failure scenarios has multiple effects on the running time of the *BGP State Iterator*. First, link failures add more stable states, as internal routers can choose different routes depending on the IGP cost. Consequently, it also affects the number of partial states that we explore. Moreover, comparing any two routes, which happens very frequently, becomes more complex: the iterator must consider all remaining failure scenarios.

In Fig. 4, we measure the time to find all stable routing states of all 160 networks for up to one (dashed orange line) and up to two (solid green line) simultaneous link failures, and compare these times with when no failures are considered (dotted blue line). Even for large networks, the *BGP State Iterator* finds all stable routing states with up to two link failures within one hour. Notice that the increase in compute time is much less than if we had considered failure scenarios independently. For instance, when considering up to two simultaneous failures, around 40k distinct failure scenarios have to be considered for the largest network. Yet, the increase in compute time is a factor of less than 2k.

Takeaways: The *BGP State Iterator* scales to large networks, both in terms of number of routers (we evaluate networks of up to 200 routers) and external networks (we evaluate up to 25 of them), finding *all* states within hours

Configuration	Time	# states explored	# states found
(i) Gao-Rexford policies	2.30 s	75 k	42
(ii) + Route reflection	2.65 s	86 k	42
(iii) + No-Advertise	170.60 s	47 086 k	3 624
(iv) + Route filters	223.45 s	47 086 k	3 624
(v) + Adjust local-pref	986.10 s	222 476 k	13 247
Verification (correct)	0.67 s	6 k	0
Verification (with bug)	0.88 s	178 k	9
<i>Minesweeper</i> ⁻ (correct)	<i>Timeout after 1 day</i>		
<i>Minesweeper</i> ⁻ (with bug)	<i>Timeout after 1 day</i>		

Table I: The *BGP State Iterator* finds *all* states for complex configurations in under an hour. It verifies such complex configurations within a second, exploring less than 0.1% of states compared to the full exploratoin. *Minesweeper*⁻ fails to solve the same problem within one day.

at most. Likewise, our backtracking algorithm can explore up to two simultaneous link failures within hours.

B. Configuration complexity

In the next set of experiments, we increase the complexity of the configuration by adding common BGP policies. We configure all 153 routers of the *Colt* network and 10 external networks with various policies and find all stable states when considering no link failures. The upper half of Table I compares the median running time as well as the number of states explored and found across five configurations, each progressively increasing the complexity of the previous one.

We start with the basic *Gao-Rexford policies* [37] that set a high local-preference for customer routes, a low one for provider routes, and one in between for routes from peers. The policy also tags routes so they can be filtered out on outgoing eBGP sessions. The *BGP State Iterator* finds all 42 states in a few seconds.

Next, we introduce three randomly placed *Route Reflectors* in the network. Each other router is a client of all three reflectors, while the reflectors themselves are connected in a full-mesh. This iBGP topology is common as it maintains reachability even if some reflectors fail [38]. Changing the iBGP topology does not increase the number of stable states, but it slightly increases the number of explored states. This is because the propagation paths become longer, making our restrictions in the Preference step less effective, c.f., §IV-E.

On top of this, we consider the well-known community *No-Advertise* that allows a route to be selected, but prevents it from being propagated further [39]. Doing so increases the running time from seconds to minutes, and the number of stable states and states explored by two and three orders of magnitude, respectively. Indeed, handling this community (*i*) effectively duplicates all available routes, and (*ii*) allows customer and provider routes to be selected at the same time.

Route filters are common in BGP configurations, e.g., to ignore routes with certain ASes along the path. We encode

this as a special community—we configure the network to ignore routes advertising that community. Introducing route filters only changes the conditions on the environment, but not the number of explored or stable states. The *BGP State Iterator* must thus check more constraints during the exploration, which slightly increases its running time to around 4 minutes.

Finally, we allow peers to *adjust the local-preference* by advertising a special community. Again, doing so increases the number of available routes available and stable states in the network. Yet, we find all 13 k stable states in around 20 minutes by exploring around $2 \cdot 10^8$ partial states.

Takeaways: These experiments demonstrate that the *BGP State Iterator* can find *all* stable states even for complex configurations in reasonable time.

C. Specification-Guided Exploration for Verification

In all previous experiments, we explored the entire tree of routing states. However, for most applications, this is superfluous. Specifically, for verifying a network we only need to traverse the parts of the tree that contains counterexamples. As a demonstration, we use the *BGP State Iterator* to verify that valid routes from customers are always preferred over routes from peers and providers.

To that end, we configure the *Colt* network with all five configuration options and policies mentioned in Section VI-B. We then traverse only the branches in which a customer’s route is not preferred. In essence, we traverse the tree multiple times, once for each pair of a customer and a peer/provider. In each iteration, we require the border router adjacent to the peer/provider to select its local route, while the customer advertises a valid one (without the No-Advertise community). Doing so will only explore the parts of the tree that can contain violations, see Section V.

The results are shown in Table I. The *BGP State Iterator* explores only 6 k states in less than a second to prove that the specification is satisfied. Recall that exploring the entire tree necessitates searching through over $10\,000\times$ more states.

Next, we introduce a bug in the configuration to evaluate the behavior of our algorithm when there exists counterexamples. We do so by adding a new route map on an iBGP session that increases a route’s local-preference if a specific community is present. On this modified configuration, *BGP State Iterator* still finds all 9 counterexamples in less than a second.

We compare the performance of *BGP State Iterator* with *Minesweeper* [10], a popular open-source control-plane verifier based on an SMT solver. *Minesweeper* supports many protocols and features not needed in our experiments. To achieve a fairer comparison, we implement *Minesweeper*⁻ by expressing only the parts pertaining to BGP in SMT constraints, while keeping all optimization techniques of the original paper. Despite that, *Minesweeper*⁻ still cannot solve the same verification task as it times-out after one day.

Takeaways: Specification-guided exploration reduces the number of states to explore to less than 0.1% compared to full exploration. We find all counterexamples in less than a second, while *Minesweeper*⁻ times out after one day.

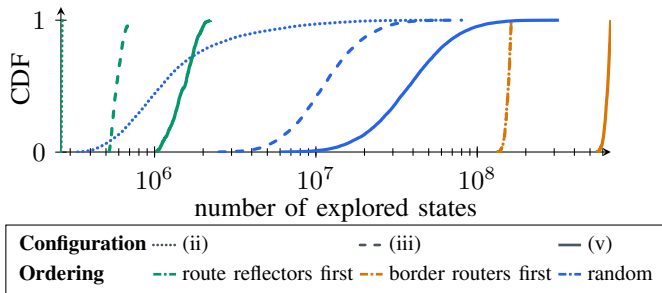


Fig. 5: Choosing a different ordering can change the number of explored states by up to three orders of magnitude in the median case. Exploring route reflectors first and border routers last consistently improves the iterator’s performance across a range of configurations presented in Table I.

D. Router Orderings

Finally, we compare the impact of different router orderings on the number of explored states and, consequently, on the running time of the *BGP State Iterator*. Determining which routers are considered first affects the effectiveness of the restrictions generated in the Preference step of our backtracking algorithm. To measure this effect, we compare the distribution of the number of explored states by repeatedly searching for all stable states of configurations ii, iii and v from Table I on the *Colt* network, while varying the router order.

Fig. 5 demonstrates that the router order can impact the number of states explored by up to three orders of magnitude. It also confirms our theoretical predictions from Section IV-E: exploring the border routers first causes the algorithm to explore partial states that do not contain any reachable states.

VII. RELATED WORK

The field of network-verification has been studied extensively in the past two decades. On the one hand, many tools focus on verifying (programmable) data planes. Such systems assume a fixed control-plane state and take forwarding rules as an input to verify properties on propagation paths or the absence thereof [1], [21], [40]–[45].

Control-plane verifiers, on the other hand, reason about how (distributed) routing protocols converge on forwarding paths. These systems can verify properties on the control-plane, e.g., whether certain routes are propagated to a peer while others are not, and/or the data-plane, e.g., whether a certain waypoint policy is satisfied. Systems like *Arc* [46], *Tiramisu* [47], *NetDice* [16], *Plankton* [48], and *SRE* [15] can verify such properties when considering arbitrary link failures, but they expect a fixed set of routing advertisements as input.

Several systems have been proposed to verify properties on networks under arbitrary failures and routing inputs. *Minesweeper* [10], *NV* [49], and *Acorn* [32] rely on SMT solvers to solve the stable-state equation, similar to those we present in Section IV-B. While SMT solvers allow users to

customize the solving strategy, using those to guide the exploration for control-plane verification is still an open problem.

The most similar work to ours is *Expresso* [5], which relies on BDDs to symbolically execute the routing and forwarding planes, resulting in the set of all forwarding paths along with conditions on the environment to realize them. Like the *BGP State Iterator*, *Expresso* finds an entire space of environments that result in a violation. However, *Expresso* must perform the entire symbolic execution to verify most control-plane properties such as route preference, whereas our backtracking algorithm only explores the relevant parts of the search space.

Complementary Works: Many control-plane verification systems achieve scalability by introducing equivalence classes on the environment [5], [10]. These are sets of environments that will be treated identically by the network, and thus reduce the overall search space. Some go further—*Bonsai* [50] proposes to find similar routers that can be combined together. The *BGP State Iterator* inherently computes equivalence classes on the environment through the constraints on a stable state. Yet, the ideas of *Bonsai* apply to our algorithm as well, further improving its scalability.

Recent systems such as *Lightyear* [3], *Timepiece* [4], and *Kirigami* [33] propose to split up the verification problem by partitioning the topology into smaller components. They define interfaces between them and invoke an SMT solver to verify each component, allowing to scale not in terms of the network size but in the size of the largest component. Our contributions complement those from modular network verifiers.

Probabilistic Verification: We are aware of three systems able to reason about probabilistic network properties. However, they all focus on probabilistic failures. *ProbNetKAT* [51] is a data-plane programming language to formally reason about properties like congestion and fault tolerance. Similarly, *ProbNV* [17] presents a language to express control-plane programs and reason about their behavior. *NetDice* [16] investigates probabilistic forwarding properties on ISP networks. However, no system yet allows the use of a probabilistic model of routing inputs to find the most likely counterexamples, or to verify that a property holds with some probability p .

VIII. CONCLUSION

From the postulate that current control-plane verifiers lack usability, we pinpoint their weakness in the loss of networking context that is essential to *guide* the exploration of the search space. With this in mind, we proposed the *BGP State Iterator*, a versatile algorithm that scales in multiple dimensions thanks to its specification-guided exploration. Beyond traditional control- and data-plane policies on stable states, we open the door for a new class of probabilistic specifications, enabling operators to reason about high-level requirements from SLAs.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful feedback. The research leading to these results was supported by an ERC Starting Grant (SyNET) 851809.

REFERENCES

- [1] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *9th USENIX Symposium on Networked Systems Design and Implementation*, 2012.
- [2] 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*, 2005.
- [3] A. Tang, R. Beckett, S. Benaloh, K. Jayaraman, T. Patil, T. Millstein, and G. Varghese, "Lightyear: Using modularity to scale bgp control plane verification," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023.
- [4] T. Alberdingk Thijm, R. Beckett, A. Gupta, and D. Walker, "Modular control plane verification via temporal invariants," *Proceedings of the ACM on Programming Languages*, 2023.
- [5] D. Wang, P. Zhang, and A. Gember-Jacobson, "Expresso: Comprehensively Reasoning About External Routes Using Symbolic Simulation," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024.
- [6] T. Schneider, S. Vissicchio, and L. Vanbever, "Verifying maximum link loads in a changing world," in *22th USENIX Symposium on Networked Systems Design and Implementation*, 2025.
- [7] M. Brown, A. Fogel, D. Halperin, V. Heorhiadi, R. Mahajan, and T. Millstein, "Lessons from the evolution of the batfish configuration analysis tool," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023.
- [8] A. Tang, S. K. R. Kakarla, R. Beckett, E. Zhai, M. Brown, T. Millstein, Y. Tamir, and G. Varghese, "Campion: debugging router configuration differences," in *Proceedings of the ACM SIGCOMM 2021 Conference*, 2021.
- [9] M. T. Arashloo, R. Beckett, and R. Agarwal, "Formal methods for network performance analysis," in *20th USENIX Symposium on Networked Systems Design and Implementation*, 2023.
- [10] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the ACM SIGCOMM 2017 Conference*, 2017.
- [11] L. De Moura and N. Björner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
- [12] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: portfolio-based algorithm selection for sat," *Journal of artificial intelligence research*, vol. 32, 2008.
- [13] N. G. Ramírez, Y. Hamadi, E. Monfroy, and F. Saubion, "Evolving smt strategies," in *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2016.
- [14] M. Balunovic, P. Bielik, and M. Vechev, "Learning to solve smt formulas," in *Advances in Neural Information Processing Systems*, 2018.
- [15] P. Zhang, D. Wang, and A. Gember-Jacobson, "Symbolic router execution," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022.
- [16] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev, "Probabilistic verification of network configurations," in *Proceedings of the ACM SIGCOMM 2020 Conference*, 2020.
- [17] N. Giannarakis, A. Silva, and D. Walker, "Probnv: probabilistic verification of network control planes," *Proceedings of the ACM on Programming Languages*, 2021.
- [18] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "The stable paths problem and interdomain routing," *IEEE/ACM Transactions On Networking*, vol. 10, no. 2, 2002.
- [19] Y. Rekhter, T. Li, and S. Hares, "RFC 4271: A border gateway protocol 4 (BGP-4)," Tech. Rep., 2006.
- [20] B. Quoitin and S. Uhlig, "Modeling the routing of an autonomous system with c-bgp," *IEEE network*, vol. 19, no. 6, 2005.
- [21] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A General Approach to Network Configuration Analysis," in *12th USENIX Symposium on Networked Systems Design and Implementation*, 2015.
- [22] T. Schneider, R. Birkner, and L. Vanbever, "Snowcap: Synthesizing network-wide configuration updates," in *Proceedings of the ACM SIGCOMM 2021 Conference*, 2021.
- [23] L. Cittadini, M. Rimondini, S. Vissicchio, M. Corea, and G. Di Battista, "From theory to practice: Efficiently checking bgp configurations for guaranteed convergence," *IEEE Transactions on Network and Service Management*, vol. 8, no. 4, 2011.
- [24] R. Bush, T. G. Griffin, J. Li, Z. M. Mao, E. Purpus, and D. Stutzbach, "Happy packets to you!" in *Proceedings of the IEEE 24th International Conference on Computer Communications (INFOCOM)*, 2005.
- [25] F. Wang, Z. M. Mao, J. Wang, L. Gao, and R. Bush, "A measurement study on the impact of routing events on end-to-end internet path performance," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, 2006.
- [26] P. Merindol, P. David, J.-J. Pansiot, F. Clad, and S. Vissicchio, "A fine-grained multi-source measurement platform correlating routing transitions with packet losses," *Computer Communications*, vol. 129, 2018.
- [27] R. Schmid, T. Schneider, G. Fragkoulis, and L. Vanbever, "Transient forwarding anomalies and how to find them," in *Proceedings of the 21th ACM International Conference on Emerging Networking Experiments and Technologies*, 2025.
- [28] P. Francois and O. Bonaventure, "Avoiding transient loops during the convergence of link-state routing protocols," *IEEE/ACM Transactions on Networking*, vol. 15, no. 6, 2007.
- [29] S. Vissicchio, L. Vanbever, C. Pelsser, L. Cittadini, P. Francois, and O. Bonaventure, "Improving network agility with seamless bgp reconfigurations," *IEEE/ACM Transactions on Networking*, vol. 21, no. 3, 2012.
- [30] L. Vanbever, "Methods and techniques for disruption-free network reconfiguration," Ph.D. dissertation, 2012.
- [31] T. Schneider, R. Schmid, S. Vissicchio, and L. Vanbever, "Taming the transient while reconfiguring bgp," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023.
- [32] D. Raghunathan, R. Beckett, A. Gupta, and D. Walker, "Acorn network control plane abstraction using route nondeterminism," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2022.
- [33] T. A. Thijm, R. Beckett, A. Gupta, and D. Walker, "Kirigami, the verifiable art of network cutting," *IEEE/ACM Transactions on Networking*, vol. 32, no. 3, 2024.
- [34] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella, "Detecting network load violations for distributed control planes," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.
- [35] R. Li, Y. Yuan, F. Ye, M. Liu, R. Yang, Y. Yu, T. Guo, Q. Ma, X. Zeng, C. Xu *et al.*, "A general and efficient approach to verifying traffic load properties under arbitrary k failures," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024.
- [36] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, 2011.
- [37] L. Gao and J. Rexford, "Stable internet routing without global coordination," *IEEE/ACM Transactions On Networking*, vol. 9, no. 6, 2001.
- [38] T. Bates, E. Chen, and R. Chandra, "RFC 4456: Bgp route reflection: An alternative to full mesh internal bgp (ibgp)," Tech. Rep., 2006.
- [39] J. Borkenhagen, R. Bush, R. Bonica, and S. Bayraktar, "RFC 8642: Policy Behavior for Well-Known BGP Communities," Tech. Rep., 2019.
- [40] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," 2011.
- [41] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.
- [42] A. Horn, A. Kheradmand, and M. Prasad, "Delta-net: Real-time network verification using atoms," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.
- [43] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caçaval, N. McKeown, and N. Foster, "P4v: Practical verification for programmable data planes," in *Proceedings of the ACM SIGCOMM 2018 Conference*, 2018.
- [44] R. Wang, X.-J. Wu, and J. Kittler, "Symnet: A simple symmetric positive definite manifold deep learning method for image set classification," *IEEE/ACM Transactions on Networking*, vol. 33, no. 5, 2021.
- [45] M. Moeller, J. Jacobs, O. S. Belanger, D. Darais, C. Schlesinger, S. Smolka, N. Foster, and A. Silva, "Katch: A fast symbolic verifier for netkat," *Proceedings of the ACM on Programming Languages*, 2024.
- [46] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proceedings of the ACM SIGCOMM 2016 Conference*, 2016.

- [47] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *17th USENIX Symposium on Networked Systems Design and Implementation*, 2020.
- [48] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *17th USENIX Symposium on Networked Systems Design and Implementation*, 2020.
- [49] N. Giannarakis, D. Loehr, R. Beckett, and D. Walker, "Nv: An intermediate language for verification of network control planes," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.
- [50] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Bonsai: Control plane compression," in *Proceedings of the ACM SIGCOMM 2018 Conference*, 2018.
- [51] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva, "Probabilistic netkat," in *Proceedings of the 25th European Symposium on Programming, ESOP 2016*, 2016.