LIGHTYEAR: Using Modularity to Scale BGP Control Plane Verification

Alan Tang
UCLA

Ryan Beckett
Microsoft

Steven Benaloh
Microsoft

Karthick Jayaraman
Microsoft

Tejas Patil
Microsoft

Todd Millstein
UCLA

George Varghese
UCLA

ABSTRACT

Current network control plane verification tools cannot scale to large networks because of the complexity of jointly reasoning about the behaviors of all network nodes. We present a *modular* approach to control plane verification, where end-to-end network properties are verified via a set of purely *local* checks on individual nodes and edges. The approach targets verification of reachability properties for BGP configurations, and provides guarantees in the face of arbitrary external route announcements and, for some properties, arbitrary node/link failures. We have proven the approach correct and implemented it in a tool LIGHTYEAR. Experimentally we show LIGHTYEAR scales dramatically better than prior control plane verifiers. Further, LIGHTYEAR has been used for six months to verify properties of a major cloud provider network containing hundreds of routers and tens of thousands of edges, finding and fixing bugs in the process. To our knowledge no prior control-plane verification tool has been shown to scale to that size and complexity. Our modular approach also makes it easy to localize configuration errors and enables incremental re-verification.

CCS CONCEPTS

• **Networks** → **Network reliability**;

KEYWORDS

Network Verification, BGP, Modular Reasoning

ACM Reference Format:
Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. 2023. LIGHTYEAR: Using Modularity to Scale BGP Control Plane Verification. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3603269.3604842

1 INTRODUCTION

Routing in networks today is controlled using low-level configuration on individual routers, which often leads to errors, potentially causing a network outage. Many earlier techniques try to remedy this by verifying configurations against specified end-to-end network behavior. For instance, Minesweeper [4] models network behavior using SMT constraints, ARC [10] and Tiramisu [1] use graphs, and Plankton [22] uses explicit-state model checking.

These techniques provide strong guarantees, frequently reasoning about network behavior over all possible external announcements and/or link failures. However, a key open problem is to scale these techniques to large networks. While these approaches attempt to scale through various means, they are not efficient enough to be used today on large real-world networks such as the wide-area networks of hyperscalers.

This lack of scalability is fundamentally caused by a shared limitation of earlier approaches: they model and reason about network behavior *monolithically*. They analyze the network configuration and routing processes as a whole, exhaustively exploring all possible control-plane behaviors induced by the complex interactions among all configuration directives and protocols. As the size of the network grows, the number of possible network states grows exponentially, limiting their ability to scale. By contrast, verification has scaled to large systems in other domains, like software or hardware, through *modular* checking. In this style, subsystems (e.g., a software function or hardware module) are verified independently to meet *local* specifications (e.g., a precondition/postcondition pair) that together imply a desired global property [11, 14, 21]. Prior work has used modularity to scale data-plane analysis [12], but modularizing control-plane verification is more challenging due to complex routing protocols and policies.

This paper presents a modular approach to network control plane verification. Like prior verifiers, LIGHTYEAR takes as input a network's configuration and a *global* property to verify. To ensure the property, LIGHTYEAR additionally requires the user to provide local constraints that should hold on individual routers and edges. LIGHTYEAR then automatically produces a set of *local* checks on individual nodes and edges that (1) verify the user's local constraints and (2) ensure that these constraints imply the given end-to-end property.

We focus on BGP since it is ubiquitous and in many networks is the most complex process that impacts the data plane's forwarding behavior. Our approach targets two common classes of BGP reachability properties. First, *safety* properties on individual routers intuitively ensure that "bad" routes never reach a particular node. This includes common properties like filtering bogons, preventing transit between peers, and ensuring isolation. Second, we target

ACM SIGCOMM '23, September 10, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09.
https://doi.org/10.1145/3603269.3604842

liveness properties, which intuitively ensure that a "good" route will eventually be accepted or forwarded at a particular location. This includes many control-plane reachability queries, for example that a route received from one neighbor will be sent to another.

Modularizing control-plane verification is challenging. Control plane behavior depends on the interaction of complex configurations with BGP, a distributed message-passing protocol. A classical way to reason modularly about protocols is through invariants indexed by time [2], and/or employ temporal logic [17]. This requires significant effort and expertise. Instead, we demonstrate that in practice a wide range of desired properties can be modularly verified without making time explicit. Reasoning modularly about liveness properties is particularly challenging; it requires that the modular checks together imply an end-to-end path through the network. We describe a natural approach to ensure this using two kinds of constraints: *path constraints* that ensure the feasibility of a "good" path, and *no-interference invariants* that ensure good paths cannot be prevented.

We have formalized our approach to modular control plane verification, proved its correctness, and built a tool called Lightyear based on it. Lightyear's approach offers several advantages over the prior work, as summarized in Table 1:

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

Strong Guarantees: If all of Lightyear's local checks are satisfied, then the specified network property is guaranteed to hold for all possible external route announcements from neighbors. Further, for safety properties our guarantees hold even in the presence of arbitrary node or link failures, though this is not true in general for liveness properties. As shown in the first two rows of the table, of the prior work only Minesweeper [4] supports reasoning about both external route announcements and failures.

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

Lightyear's main tradeoff is that users must specify local constraints. However, for networks designed in a modular and structured fashion, only a few simple constraints are required for any desired end-to-end property. For example, network nodes are commonly partitioned into *roles*, such as border or core, each with its own responsibilities; nodes in the same role will typically have the same local constraints.

In addition, the scalability and localization properties of our tool make it easy for users to hypothesize an initial set of local constraints and then refine them iteratively based on feedback. We used this approach to produce the local constraints in our real-world experiments (see below), having brief discussions with the network operators based on Lightyear's feedback in order to either determine that an identified issue was a real configuration error or to update our local invariants appropriately.

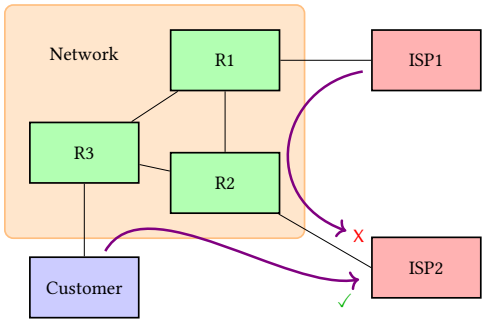


Figure 1: Example network with safety and liveness properties (shown intuitively by the purple arrows). Routes from ISP1 should not be sent to ISP2 (safety). Routes from Customer should reach ISP2 (liveness).

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

In summary, we make the following contributions:

(1) **Modularity:** A novel solution to scaling control plane verification by checking individual routers locally.
(2) **Formalization:** A formal model of BGP routing that we use to prove correctness of the modular approach.
(3) **System:** A tool Lightyear built using our approach, which has been running in a hyperscaler for six months.
(4) **Evaluation:** A demonstration of Lightyear's ability to scale to very large networks experimentally.

This work does not raise any ethical issues.

2 APPROACH OVERVIEW

In this section we show how Lightyear works with the example in Figure 1. In the example network, each edge represents a connection between BGP speakers. The network contains three BGP routers: R1, R2, and R3. R1 and R2 each have an ISP as an external neighbor. R3 is connected to an external neighbor that is a customer. The network satisfies two properties. First, it satisfies the standard no-transit property that routes originating from ISP1 should not be advertised to ISP2, and second, it satisfies the property that routes from Customer, with appropriate prefixes, should eventually be sent to ISP2. The former is a *safety property*, holding when a certain event never occurs, and the latter is a *liveness property*, holding when a event must eventually occur. Both are network-wide policies in that they depend on the interaction of multiple routers to achieve the correct result.

Existing control-plane verifiers [1, 4, 10, 22, 24] would verify these properties by creating a representation of the possible data planes that can result from the *entire network's configuration* and then searching this representation for counterexamples. This joint representation of all network node behaviors has inherent scalability limitations.

However, we observe that network configurations are highly structured and modular by design. Each router contains *route maps*

Tool Feature	**Minesweeper** [4]	**BagPipe** [24]	**Plankton** [22] **Tiramisu** [1]	**ARC** [10] **Hoyan** [26]	**Lightyear**
Analyzes all peer BGP routes	●	●	○	○	●
Analyzes failures	●	○	●	●	◐
Checks safety and liveness properties	●	◐	●	●	●
Verification is fully automatic	●	●	●	●	◐
Near linear scaling with network size	○	○	○	○	●
Localizes bugs in configurations	○	○	○	○	●

Table 1: Comparison of prior verification tools with Lightyear.

(also called *route policies* and *route filters*), which define an import and export policy on each BGP peering session, determining which routes are rejected, which are accepted, and how accepted routes are transformed. Each route map plays a particular role in the assurance of desired global properties. For example, in the network from Figure 1 the no-transit property can be ensured using a common approach based on communities: (1) R1's import policy marks received routes from ISP1 with a BGP community (a simple 32-bit tag) with value `100:1` (2) R2's export policy filters routes tagged with `100:1` when advertising to ISP2, and (3) no other import or export policy strips community `100:1` from routes that it advertises.

Note that each of the above behaviors is *node-local* and pertains to an individual BGP route map. Unlike Lightyear, prior control plane verification tools are not aware of this modular structure and so cannot leverage it. Alternatively, one could envision making a tool that simply performs a set of user-specified local checks like the ones above. However, in that case there is no guarantee that together they imply the desired end-to-end property. Even in this simple example, the fact that it is necessary to check the third condition above is subtle and easily missed.

Figure 2 shows the architecture of Lightyear. Like prior control-plane verifiers, it takes as input the network configuration and an end-to-end property to verify. However, Lightyear requires the user to provide additional local constraints that capture the modular structure of the configurations. From these inputs Lightyear generates a set of local checks on individual nodes in the network and uses a constraint solver to verify each one. If all of these local checks succeed, then the end-to-end property is guaranteed to hold, for all possible external route announcements from neighbors and, for safety properties, for all possible link and node failures in the network. Otherwise, Lightyear provides concrete counterexamples for each failed local check.

In the rest of this section, we show how Lightyear modularly verifies the two properties for the network in Figure 1.

2.1 Safety Properties

End-to-end Property: For safety properties, the end-to-end property of interest is specified as a pair of a particular location in the network and a predicate on all routes reaching that location. Many network policies fall into this class of properties, for example bogon filtering; ensuring that a network only advertises routes to its own destinations; and forms of isolation between nodes or groups of nodes. Such properties can also express complex constraints among BGP attributes, for example that prefixes in a specific range always have a particular local preference or MED value.

As shown in the first line of Table 2, the no-transit property specifies that no route transmitted over the edge from R2 to ISP2 should originate at ISP1. To enable the expression of rich properties, Lightyear allows users to define *ghost attributes* that conceptually update message headers with additional fields. This is a common technique in software verification, where additional variables are introduced that do not affect the computation but allow for easier property specification [9]. In the table, `FromISP1`(r) is a boolean ghost variable that is defined by the user to be false in all originated routes, set to true by the import filter on R1 from ISP1, and left unchanged by all other filters.

Network Invariants: Users must also specify invariants that are true for routes at locations within the network. While in principle the user could specify a different invariant for each network location, many locations play the same role in the network and have the same behavior with respect to the desired end-to-end property. In our example, there are only three network invariants, shown in Table 2, which correspond exactly to the three node-local behaviors described earlier that ensure the no-transit property. First, no assumption is made about the routes coming from ISP1 to R1, so the associated predicate is True. Second, routes coming from R2 to ISP2 should not come from ISP1. Note that this invariant is identical to the end-to-end property, which is common but need not be the case. Third, all other locations in the network should satisfy the key correctness invariant: routes from ISP1 must be tagged with the community `100:1`.

For many safety properties, like in the example above, invariants follow a straightforward three-part structure. First, very little is assumed about routes coming from outside the network (so the associated local invariant is `True` or similarly nonrestrictive). Second, the desired global property should hold at the corresponding location in the network (the edge from R2 to ISP2 in the above example). Third, there is a key invariant that holds in the rest of the network, which intuitively describes *how* the network ensures the global property. In our example above, the invariant specifies the fact that the network uses the community `100:1` to keep track of the routes that came from ISP1. In general this invariant restricts the routes that can flow through the network to be of a limited kind, for example a specific set of prefixes or containing specific attribute values such as the MED, local preference or communities. Notably, this three-part decomposition is analogous to the modular verification of software [11], which typically involves a *precondition* that is assumed to hold initially, a *postcondition* to be proven, and one or more *inductive invariants* that hold throughout each execution and are sufficient to imply the postcondition.

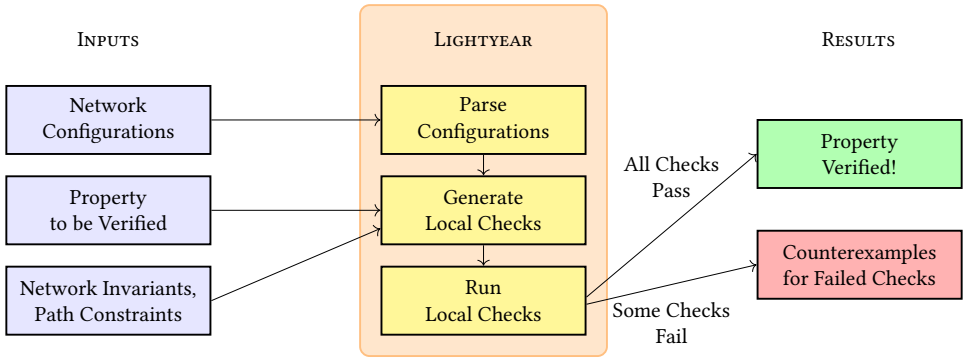


Figure 2: The architecture of Lightyear.

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

Table 2: Using Lightyear to prove the no-transit property from Figure 1. The user-provided global property and local invariants are show in blue. Lightyear-generated local verification checks are shown in yellow.

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

Table 3: Using Lightyear to prove the liveness property from Figure 1. The user-provided global property, and path constraints are show in blue. The propagation checks are shown in yellow for the path is Customer → R3 → R2 → ISP2. The no-interference checks are safety properties proven using their own invariants (not shown).

Importantly, these local invariants are far from complete specifications of the network's routing behavior. Rather, local invariants need only describe the constraints on routes that are necessary to ensure the particular global property of interest. For example, suppose that our example network uses a route's local preference value to choose among multiple routes to a destination. Since the local preference values don't pertain to our no-transit property, this behavior need not be specified.

Generated Checks: Given the invariants provided from the user, LIGHTYEAR automatically generates local checks to validate the given network invariants. Importantly, each local check pertains to a single BGP filter on a single network router, applied to messages from a specific neighbor. Together these checks implement a form of *assume-guarantee reasoning* [14, 21]: each location's network invariant is proven under the assumption that the network invariants of its directly connected locations hold. As we prove later, together these checks imply that all local invariants in the network are respected.

Table 2 shows the local checks that LIGHTYEAR automatically generates for our running example. The first check ensures that the import filter at R1 on the edge from ISP1 to R1 establishes the key invariant $\texttt{FromISP1}(r) \Rightarrow 100{:}1 \in \text{Comm}(r)$. Since that filter tags all routes with community `100:1`, the check is easily provable by a constraint solver. The second check ensures that the key invariant is sufficient to ensure that routes from ISP1 are not exported on the edge from R2 to ISP2. Since the export filter at R2 on that edge drops all routes that are tagged with `100:1`, the check passes. The third set of checks ensure that the key invariant is preserved by all other import and export filters in the network. Since these filters never strip community `100:1` from a route, the checks pass.[1] Lastly (not shown in the table), LIGHTYEAR must check that the invariant on the edge from R2 to ISP2 implies the end-to-end property. This check is trivial since the two properties are identical.

Output: If the configuration contains errors, LIGHTYEAR returns a counterexample for each local check that did not pass. In our example, suppose that R1's import filter accidentally does not add the community 100:1 for some routes received from ISP1. In that case, the first generated check in Table 2 would fail, producing a counterexample consisting of a concrete route that is accepted by R1 but does not get the community 100:1 added to it. This counterexample directly indicates the route policy that is responsible for the error and concretely illustrates the specific local property that was violated. Counterexamples from LIGHTYEAR are also be helpful in refining local invariants that are not precisely known in the beginning. For example, a user might write a local invariant for some network location but forget to account for a specific corner case. In that case LIGHTYEAR will identify an "error" due to a failed local check, and the associated counterexample informs the user how to refine that local invariant to more closely match the network location's behavior.

2.2 Liveness Properties

End-to-end Property: For liveness properties, the end-to-end property of interest is also a pair of a particular location in the network and predicate. However, here the predicate indicates that a route satisfying the property will eventually reach that location. The property in Table 3 shows that a route with a customer prefix will eventually be sent from R2 to ISP2. If the routes of interest come from a neighbor, as in this case, then the property will only be provable under the assumption that the neighbor advertises such a route. Users can optionally specify such an assumption, as shown in the table.

Path and Constraints: As with safety properties, users need to provide a set of local constraints on individual network locations, but they take a different form for liveness properties. Users must provide a *path* through the network that the desired route can take to reach the destination from the source, along with local constraints for each edge and node along the path. The path does not need to be unique. Intuitively, each local constraint indicates the properties of the "good" routes that will reach that particular location, and together they constitute a witness that a "good" route will eventually reach its intended destination. As shown in Table 3, our example has two path constraints: at locations R3, R2, and R3 $\rightarrow$ *R2* there will eventually be a route with the customer prefix that does not have the community `100:1`, and at R2 $\rightarrow$ ISP2 there will eventually be a route with the customer prefix. It is important that routes from Customer do not have the community `100:1`, or else they will be dropped at R2, due to the way that the earlier no-transit property is ensured. As described earlier, the local and concrete feedback from LIGHTYEAR can be used iteratively to identify these conditions.

Propagation Checks: In order to prove the liveness property two types of checks need to be performed. First, there are local checks that together imply that a route will in fact traverse the given path, in the absence of interference from other possible paths. These checks are analogous to the generated checks for safety properties shown earlier. Notably, in order for the first propagation check in Table 3 to be satisfied, the import policy at R3 must not accept any routes tagged with community `100:1`. One way to ensure this is for the policy to strip communities from all accepted routes. The other two checks are straightforward.

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

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

Output: As in the previous example, Lightyear returns a concrete counterexample for each failed propagation check and no-interference check. For example, if R3's import policy does not properly strip the community `100:1` from accepted routes, then Lightyear will produce a concrete example illustrating this fact, allowing the user to easily understand and localize the error.

In summary, Lightyear's approach to control-plane verification leverages the modular structure that is already present in the network configurations. By requiring the user to make this structure explicit through a set of local invariants at each location, Lightyear soundly reduces checking an end-to-end network property to a set of checks that each pertain to a single BGP import or export filter. We formalize our approach and prove its correctness in Sections 4 and 5.

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

3 FORMAL MODEL OF BGP

In this section we define a model of BGP in terms of traces and axioms on traces. This model is used in the next two sections to make Lightyear's approach precise and to prove its correctness.

3.1 BGP Topologies and Policies

We model a network's BGP configuration as consisting of two parts: a *topology* and a *policy*. A BGP network topology is a tuple of the form (Routers, Externals, Edges), where:

(1) Routers is the set of routers for which the user provides configurations.
(2) Externals is the set of external routers. That is, there is no provided configuration, but each such router is an eBGP or iBGP peer with at least one router in Routers.
(3) Edges is the set of directed edges corresponding to BGP peering sessions.

The network topology forms a graph with Routers ∪ Externals as the set of nodes and Edges as the set of edges. We will use the notation $A \to B$ to refer the directional edge (A, B) in the topology.

A BGP *route* (or *route advertisement*) is modeled as a tuple

(Prefix, ASPath, NextHop, LocalPref, MED, Comm)

where:

(1) NextHop, LocalPref, and MED are integer values
(2) Prefix is a pair consisting of an IP address and a length, both of which are integer values
(3) ASPath and Comm are lists of integer values representing the BGP path and the community tags, respectively.

Let Routes denote the set of all routes. We will use $\text{Comm}(r)$ to refer to the Comm field of the route r, $\text{Prefix}(r)$ to refer the prefix of r, and so on. Real BGP messages contain a few other attributes as well, which could be incorporated into this model. Routes can also be extended with additional "ghost" attributes, such as the `FromISP1` attribute from Section 2. This is described in Section 4.4.

We model the BGP network policy as consisting of three functions, which can be derived from the BGP and route-map configurations of each router:

(1) $\texttt{Import} : \text{Edges} \times \text{Routes} \to \text{Routes} \cup \{\text{Reject}\}$
(2) $\texttt{Export} : \text{Edges} \times \text{Routes} \to \text{Routes} \cup \{\text{Reject}\}$
(3) $\texttt{Originate} : \text{Edges} \to \mathbb{P}(\text{Routes})$

The first two correspond to the import and export route maps which are defined in the router configurations. The third models the router's ability to advertise static routes or routes from other protocols into BGP. For an edge $A \to B$ and a route r, $\texttt{Import}(A \to B, r)$ either returns the route produced when applying the import filter at B to the route r sent from A or returns Reject if the import filter rejects the route. $\texttt{Export}(A \to B, r)$ either returns the route produced when applying the export filter at A to the route r sent to B or returns Reject if the export filter rejects the route. $\texttt{Originate}(A \to B)$ returns the set of routes that are originated at A and sent to B.

3.2 BGP Traces

We model the semantics of BGP as a set of allowed *traces*. Our semantics is a variant of that from the Bagpipe tool [24].

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

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

We denote the set of all traces as Traces.

A *valid* trace is one that could occur for a given topology and policy, according to the BGP semantics. We formalize the notion of trace validity as a set Valid ⊆ Traces of traces that satisfy specific properties. We consider a trace to be valid, and hence part of the set Valid, if it satisfies a set of *safety axioms*, and a set of *liveness axioms*. These axioms are stated in Appendix A. The safety axioms are used to prove the correctness of safety checks and state necessary conditions for an event to be in the trace. For example, if a `slct` event is in the trace, then there must be a `recv` event earlier and `Import` must have transformed the received route into the selected route. The liveness axioms are used to prove the correctness of liveness checks and state sufficient conditions to show that an event occurs later in the trace. For example, if a `slct` event occurs, then the result of `Export` applied to the selected route will be used in a `fwrd` event.

In our model, external neighbors can send different announcements in different traces, and events at different locations can occur in any order.

4 SAFETY VERIFICATION IN LIGHTYEAR

In this section, we describe LIGHTYEAR's approach for modularly verifying safety properties and prove its correctness.

4.1 Inputs for Safety Checks

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

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

$$(\ell, P) \in (\text{ROUTERS} \cup \text{EDGES}) \times \mathbb{P}(\text{ROUTES})$$

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

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

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

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

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

$$I \subseteq (\text{ROUTERS} \cup \text{EDGES}) \times \mathbb{P}(\text{ROUTES})$$

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

We require that there exist exactly one pair in I per location in the given network, and we use the notation I_ℓ to denote the set P of routes associated with location ℓ in I. We also require that $I_{R \to N} = \text{ROUTES}$ for each edge $R \to N$ where $R \in \text{EXTERNALS}$. In other words, we make no assumption about routes coming from external neighbors but rather assume that any route may be advertised.

4.2 Local Checks

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

1. **Import:** For all $r, r' \in \text{ROUTES}$, if $r = \texttt{Import}(A \to B, r')$ and $r' \in I_{A \to B}$, then $r = \text{REJECT} \vee r \in I_B$.
2. **Export:** For all $r, r' \in \text{ROUTES}$, if $r = \texttt{Export}(A \to B, r')$ and $r' \in I_A$, then $r = \text{REJECT} \vee r \in I_{A \to B}$.
3. **Originate:** For all $r \in \text{ROUTES}$, if $r \in \texttt{Originate}(A \to B)$, then $r \in I_{A \to B}$.

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

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

4.3 Correctness

We have proven the correctness of our approach to modular safety verification.

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

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

Proof: See Appendix B.

4.4 Ghost Attributes

To increase LIGHTYEAR's expressiveness, users can define *ghost attributes*, which conceptually extend each route with additional fields. For example, the $\texttt{FromISP1}(r)$ ghost attribute from Section 2 is used to indicate whether r originated from ISP1. A ghost attribute is defined by specifying the set of values that the attribute can take, along with updates to the `Import`, `Export`, and `Originate` functions that make up the given network's policy (Section 3.1).

In the case of $\texttt{FromISP1}(r)$ from Figure 1, it can be defined as a boolean attribute with the following behavior:

- the import filter on ISP1 → R1 sets `FromISP1` to true
- the import filters on ISP2 → R2 and Customer → R3 set `FromISP1` to false
- other filters leave `FromISP1` unchanged
- all originated routes have `FromISP1` set to false

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

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

4.5 Fault Tolerance for Safety Properties

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

Lightyear soundly reasons about failures because of our over-approximate notion of trace validity (Section 3.2 and Appendix A). Specifically, any trace that is feasible according to the given BGP topology and passes the import and export filters along the corresponding path is considered valid. Hence, every trace that can occur under any failure scenario is already considered valid. By our correctness theorem, all of these traces satisfy the property (ℓ, P).

5 LIVENESS VERIFICATION IN LIGHTYEAR

We now describe how Lightyear checks liveness properties modularly. Proving liveness properties modularly is more difficult than proving safety properties, since it requires showing both that "good" routes are allowed and that interfering routes are not.

5.1 Inputs for Liveness Checks

The inputs for a liveness check consist of the following:

(1) The network configurations
(2) A liveness property $(\ell, P) \in \{\text{Routers} \cup \text{Edges}\} \times \mathbb{P}(\text{Routes})$
(3) A path $(\ell_1, \ldots, \ell_n = \ell)$ where $\ell_i \in \{\text{Routers} \cup \text{Edges}\}$
(4) A constraint $C_1 \ldots C_n$ for each location in the path, where $C_i \in \mathbb{P}(\text{Routes})$

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

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

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

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

5.2 Local Checks

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

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

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

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

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

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

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

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

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

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

These properties can be proven using our existing approach for proving safety properties (Section 4), given appropriate local invariants.

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

5.3 Correctness

We have proven the correctness of our approach to modular safety verification.

Theorem: Given the following:

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

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

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

then there exists r' such that either:

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

Proof: See Appendix C.

Notably, the correctness only depends on there being no link failures along the given path, so the property holds even if there are failures elsewhere.

6 EVALUATION

6.1 Cloud WAN

We used Lightyear to modularly verify properties of the wide-area network (WAN) of a major cloud provider, containing hundreds of routers and tens of thousands of peering sessions. In doing so, we show that: (1) important behavioral properties in real-world networks can be expressed in Lightyear; (2) these properties can be proven through a combination of modular checks; (3) this approach scales, allowing properties to be verified quickly; and (4) if a local check does not succeed, it produces actionable information,

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

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

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

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

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

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

Table 4: End-to-end properties and network invariants for three use cases in the WAN.

indicating a bug in either a specific route map or a specific local invariant. To our knowledge no prior tool that verifies properties of all possible external announcements from neighbors has been demonstrated to scale to such a size.

We used LIGHTYEAR to verify two classes of properties that the wide-area network must satisfy. In all cases we determined the intended network behavior by inspecting the configurations and talking with the network operators, and the local constraints were written based on that intent. This process was typically iterative. That is, we would write an initial property specification and its set of local invariants based on our current understanding of how the network operates. If LIGHTYEAR reported violations of local checks, we would inspect the counterexamples and discuss with operators, either determining that the bugs are real errors or identifying special cases that led to refined local invariants and (sometimes) refined end-to-end property specifications.

Implementation: We implemented LIGHTYEAR as a tool in C#. The tool parses and extracts the BGP policy along with import and export route maps from each configuration, while supporting common attributes of BGP routes such as communities, AS path, MED, local preference, along with common route map features, like matching on and setting attributes. The tool allows users to provide local invariants written as a C# function using the Zen constraint solving library [28], and to specify the routers and policies of interest. The Zen library translates the functions into SMT formulas that are solved by Z3 [7]. For each local check that fails, the tool returns a counterexample consisting of a specific route map and a concrete input route that leads to a violation.

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

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

with different properties $Q(r)$. These include properties like not accepting bogons or routes with invalid AS paths. An example of the invariants for the no-bogons property is shown in Table 4a. The network has a set of Internet *edge routers*, that peer with Internet service providers, other cloud providers, and customers, and so act as gateways between the cloud provider and the Internet. The wide-area network ensures that "bad" routes are not admitted by filtering them at all of the Internet edge routers.

As mentioned earlier, running LIGHTYEAR to check these properties is an iterative process, which involves refining the local constraints based on operator feedback. In the end, through this process

Lightyear identified 11 actual configuration errors. These included cases where a route map denied more traffic than intended, and inconsistencies between the filters of edge routers that are intended to have similar behavior. For example, in one case, among the hundreds of similarly defined peering sessions, it was discovered that a handful had ad-hoc policies that filtered AS paths differently. All of the findings were latent bugs that did not have an immediate impact, but could become impactful in the presence of failures or changes in the external announcements received from neighbors. Further, because Lightyear is sound the operators can be sure that these are the *only* violations of the desired end-to-end properties. As of this writing, all identified errors are prioritized for fixing by network engineers.

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

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

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

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

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

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

The WAN enforces these properties by tagging routes for reused IP addresses with a region-specific community C when they are received from data centers. Routers in the same region then accept routes tagged with that community, while routers in other regions reject them. The local constraints we used to verify the safety and liveness properties are shown in Table 4b and 4c respectively. One subtlety is that routes to reused IP addresses in the region of interest must not only have the community C, but they also must not be tagged with any other region's community. Otherwise, these routes could be accidentally accepted by other regions. The local constraints validate this property, and the WAN enforces it by deleting all communities on routes coming from the data centers, before adding the community C.

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

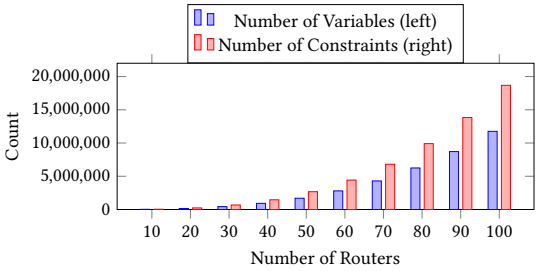
6.2 Scaling Experiments

To illustrate the scaling benefits of modular checking, we compared Lightyear with Minesweeper [4] on synthetic test cases. For a fair comparison, we created an implementation of Lightyear that is built on top of the same parser and constraint generation system as Minesweeper. This is a different implementation from the one used on the cloud network. We use a BGP full mesh where each router is connected to one external neighbor through eBGP and all other routers through iBGP. This leads to a total of N^2 edges in a network of size N. The network's configuration is relatively simple, with each eBGP connection using only prefix and community filters. We checked a no-transit safety property, similar to the example in Figure 1.

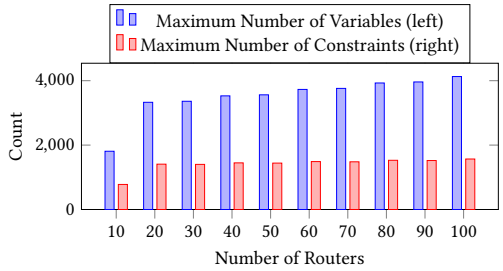
Figure 3 provides details on these results by comparing the number of SMT variables and constraints generated by each tool, as well as the amount of time used to solve the SMT constraints compared to the total computation time. As the network size increases, Minesweeper requires several orders of magnitude more SMT variables and constraints than the maximum number required by Lightyear for any local check (compare Figures 3a and 3b). As a result, SMT solving time dominates the run time of Minesweeper and is the limiting factor on its ability to scale, while for Lightyear the solving time is a relatively small portion of the total time (compare Figures 3c and 3d). Minesweeper does not terminate within two hours when run on a network of size 40, while Lightyear verifies a network of size 100 in 5.5 minutes.

7 RELATED WORK

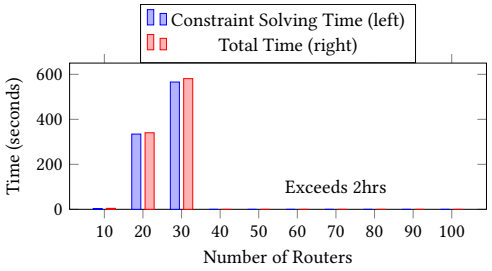
Control Plane Verification: State-of-the-art approaches to network control-plane verification were summarized in Table 1. Unlike Lightyear, these approaches are all *monolithic* — they require joint analysis of the configurations of all nodes — which dramatically limits scalability. Compared to Lightyear, Minesweeper's worst case complexity is exponential in the network size. Other improvements not only reduce generality but are at least quadratic in the network size even when using specialized algorithms. Most approaches make tradeoffs in expressiveness, for example giving up the ability to reason about all possible BGP announcements from neighbors [1, 10, 22, 27]. In contrast, Lightyear's modular approach only requires reasoning about individual BGP route maps in isolation and so is highly scalable. Lightyear also provides guarantees across all possible external announcements and, for safety properties, arbitrary failures.



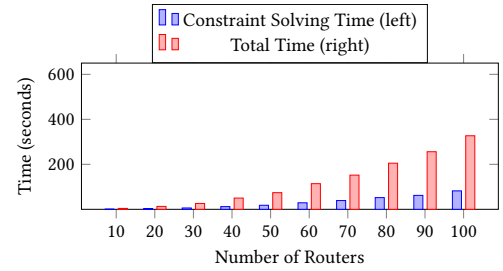
(a) Number of variables and constraints generated by Minesweeper for synthetic networks.



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



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



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

Figure 3: Comparing LIGHTYEAR and Minesweeper on synthetic networks of various sizes.

`rcc` [8] validates important properties of BGP configurations, largely through local checks on individual configuration. However, `rcc` is limited to specific "best practice" policies, and there is no guarantee that the local checks together ensure the desired end-to-end properties.

Closest to our work are recent techniques for modular control-plane verification, Kirigami [23] and Timepiece [2], which also use assume-guarantee reasoning for the control plane via local invariants. However, each approach makes a different set of tradeoffs than LIGHTYEAR. Kirigami's local invariants require the exact routes that will arrive on a particular edge. Because these invariants are fully concrete, Kirigami cannot reason about arbitrary route announcements from neighbors or give guarantees in the presence of failures.

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

Another line of work has improved scalability of control-plane verification through forms of *abstraction* [5, 6]: the full network is analyzed monolithically, but irrelevant or redundant configuration information is abstracted away to simplify the analysis. Our work is orthogonal to this line of work; the two approaches could be combined.

Data Plane Verification: Other tools check properties of forwarding state, rather than network configurations [3, 13, 15, 16, 18, 19, 25]. These approaches generally require joint reasoning about the entire network. A recent exception is RCDC [12], which modularly verifies global reachability contracts in a data center via local checks. However, RCDC is specific to one data center design and does not provide a general framework for decomposing global property checks into local checks. Another approach [20] exploits abstraction, such as symmetries, to scale data-plane verification.

Modular Verification: Assume-guarantee reasoning [14, 21] enables modular verification in other domains. A global property is modularized by providing each system component with local invariants that it must satisfy, assuming other components satisfy their invariants. LIGHTYEAR applies this methodology to networks to generate the local checks that each BGP policy must satisfy.

Verification often requires identifying *inductive invariants*, properties that hold over some unbounded space of system states, such as the iterations of a loop [11]. Such invariants arise naturally in networks and enable many locations to use the same local invariant. Typically, a small set of nodes establishes an inductive invariant (e.g., by attaching a community), and this invariant holds through the network as long as other nodes "do no harm" (e.g., never remove communities).

8 CONCLUSION

Exploiting symmetries in network verification [20] is natural because of hardware design patterns such as fat trees. Similarly, exploiting modularity in control plane verification is natural because of design patterns in the way configurations are written and maintained in well engineered networks. We have confirmed this hypothesis in six months of deployment at a major cloud vendor. Further, Lightyear finesses the need to reason about time to prove safety and liveness, offering a sweet spot between expressiveness and complexity that has worked well for many desired properties in our network.

In Lightyear, users must provide local network constraints. While in our experience it has been easy to determine these constraints, we believe it is possible to instead *learn* local invariants automatically from configurations in the future, for example when properties are enforced via communities.

REFERENCES

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 201–219. https://www.usenix.org/conference/nsdi20/presentation/abhashkumar

[2] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. 2023. Modular Control Plane Verification via Temporal Invariants. *Proc. ACM Program. Lang.* 7, PLDI, Article 108 (jun 2023), 26 pages. https://doi.org/10.1145/3591222

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2535838.2535862

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 155–168. https://doi.org/10.1145/3098822.3098834

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control Plane Compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 476–489. https://doi.org/10.1145/3230543.3230583

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2019. Abstract Interpretation of Distributed Network Control Planes. *Proc. ACM Program. Lang.* 4, POPL, Article 42 (dec 2019), 27 pages. https://doi.org/10.1145/3371110

[7] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[8] Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP Configuration Faults with Static Analysis. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05)*. USENIX Association, Berkeley, CA, USA, 43–56. http://dl.acm.org/citation.cfm?id=1251203.1251207

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

[10] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 300–313. https://doi.org/10.1145/2934872.2934876

[11] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[12] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019. Validating Datacenters at Scale. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 200–213. https://doi.org/10.1145/3341302.3342094

[13] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. 2014. *Automated Analysis and Debugging of Network Connectivity Policies.* Technical Report MSR-TR-2014-102. Microsoft.

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

[15] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=2228298.2228311

[16] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2012. Veriflow: Verifying Network-wide Invariants in Real Time. *SIGCOMM Comput. Commun. Rev.* 42, 4 (Sept. 2012), 467–472. https://doi.org/10.1145/2377677.2377766

[17] Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems* 16, 3 (1994), 872–923.

[18] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 499–512. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes

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

[20] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. 2016. Scaling network verification using symmetry and surgery. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 69–83. https://doi.org/10.1145/2837614.2837657

[21] Amir Pnueli. 1984. In Transition From Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems (NATO ASI Series)*, Krzysztof R. Apt (Ed.), Vol. 13. Springer, 123–144.

[22] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. (2020).

[23] Tim Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. 2022. Kirigami, the Verifiable Art of Network Cutting. (2022). https://doi.org/10.48550/ARXIV.2202.06098

[24] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. *SIGPLAN Not.* 51, 10 (Oct. 2016), 765–780. https://doi.org/10.1145/3022671.2984012

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

[26] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, 599–614. https://doi.org/10.1145/3387514.3406217

[27] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 599–614. https://doi.org/10.1145/3387514.3406217

[28] Zen [n. d.]. Zen. https://github.com/microsoft/Zen. ([n. d.]).

Appendices are supporting material that has not been peer-reviewed.

A BGP TRACE AXIOMS

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

(1) If $A_k = \texttt{recv}(N \rightarrow R, r)$, then either:
   (a) $N \in$ Externals, or
   (b) there exists $j < k$ such that $A_j = \texttt{frwd}(N \rightarrow R, r)$

(2) If $A_k = \texttt{slct}(R, r)$, then there exists $j < k$, $r' \in$ Routes, and $N \in$ Routers$\cup$Externals such that $A_j = \texttt{recv}(N \rightarrow R, r')$ and $r = \texttt{Import}(N \rightarrow R, r')$

(3) If $A_k = \texttt{frwd}(R \rightarrow N, r)$, then either:
   (a) $r \in \texttt{Originate}(R \rightarrow N)$, or

(b) there exists $j < k$ and $r' \in$ ROUTES such that $A_j = \texttt{slct}(R, r')$ and $r = \texttt{Export}(R \to N, r')$

The liveness axioms depend on the BGP route preference relation, which selects routes to the same prefix by comparing their local preference, AS paths, and other attributes. We say that $r_1 > r_2$ if r_1 is preferred over r_2. The liveness axioms consist of the following properties, for all $1 \le k \le n$:

(1) If all of the following are true:
- $A_k = \texttt{slct}(R, r) \in T$
- $r' = \texttt{Export}(R \to N, r)$ with $r' \neq$ REJECT

then there exists $j > k$ such that $A_j = \texttt{frwd}(R \to N, r')$

(2) If $r \in \texttt{Originate}(R \to N)$ then there exists $j > k$ such that $A_j = \texttt{frwd}(R \to N, r)$

(3) If $A_k = \texttt{frwd}(R \to N, r)$ and there is no link failure along $R \to N$, then there exists $j > k$ such that $A_j = \texttt{recv}(N \to R, r)$

(4) If all of the following are true:
- $A_k = \texttt{recv}(N \to R, r)$
- $r' = \texttt{Import}(N \to R, r)$ with $r' \neq$ REJECT
- For all neighbors $N' \neq N$ and routes r'': if $\texttt{Prefix}(r) = \texttt{Prefix}(r'')$ and $\texttt{recv}(N' \to R, r'') \in T$, then $r' > \texttt{Import}(N \to R, r'')$

then there exists $j > k$ such that $A_j = \texttt{slct}(R, r') \in T$.

B CORRECTNESS PROOF FOR SAFETY

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

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

Lemma: Given a BGP topology and policy as well as network invariants I, let C be the set of Import, Export, and Originate checks that LIGHTYEAR generates. If all checks in C pass, then for all $T \in$ VALID, $r \in$ ROUTES, $R, N \in$ ROUTERS:
- if $\texttt{slct}(R, r) \in T$, then $r \in I_R$
- if $\texttt{frwd}(R \to N, r) \in T \vee \texttt{recv}(R \to N, r) \in T$, then $r \in I_{R \to N}$

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

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

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

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

(1) $N \in$ EXTERNALS. In this case we know that $I_{N \to R} =$ ROUTES, so $r \in I_{N \to R}$.

(2) There exists $j < k+1$ such that $A_j = \texttt{frwd}(N \to R, r)$. Then by the inductive hypothesis we have that $r \in I_{N \to R}$.

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

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

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

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

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

Theorem: Given a BGP topology and policy, a network property (ℓ, P), and network invariants I, let C be the set of Import, Export, and Originate checks that LIGHTYEAR generates. If all checks in C pass and $I_\ell \subseteq P$, then for all $T \in$ VALID, $r \in$ ROUTES, $R, N \in$ ROUTERS:
- if $\ell = R$ and $\texttt{slct}(R, r) \in T$, then $r \in P$
- if $\ell = R \to N$ and $\texttt{frwd}(R \to N, r) \in T \vee \texttt{recv}(R \to N, r) \in T$, then $r \in P$

Proof: There are two cases:

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

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

Note that our reasoning does not depend on BGP converging as traces can be infinite.

C CORRECTNESS PROOF FOR LIVENESS

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

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

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

(1) all checks (propagation, no interference) pass

(2) there exists r such that $\texttt{recv}(\ell_1, r) \in T \wedge C_1(r)$

(3) for all r, $C_n(r) \implies P(r)$

(4) there are no link failures along the path

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

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

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

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

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

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

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

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

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

From this, we know that at ℓ_n, there exists a route r_n such that $C_n(r_n)$ and either $\texttt{frwd}(\ell_n, r_n) \in T$ or $\texttt{slct}(\ell_n, r_n) \in T$. $C_n(r_n) \implies P(r_n)$, which is what we wanted to prove. Again, note that our reasoning does not depend on BGP converging as traces can be infinite.