Expresso: Comprehensively Reasoning About External Routes Using Symbolic Simulation

Dan Wang
Xi'an Jiaotong University

Peng Zhang
Xi'an Jiaotong University

Aaron Gember-Jacobson
Colgate University

Abstract

Existing network verifiers can efficiently identify failure-induced bugs. However, an equally-important concern is identification of external-routes-induced bugs, which has not been well addressed. Comprehensively reasoning about external routes is challenging, since each external neighbor can advertise an arbitrary set of routes, which is quite a huge space. This paper introduces a new network verifier, Expresso, which uses symbolic simulation to explore the equivalences in the space of external routes. We evaluate the effectiveness and scalability of Expresso on the WAN of a large cloud service provider and Internet2. Expresso found various property violations, some of which have already been confirmed by the operators. To the best of our knowledge, Expresso is the only verifier that can check the correctness of WANs amidst arbitrary external routes in a tractable amount of time, while other verifiers time-out after 1 day.

CCS Concepts

• **Computer systems organization** → *Reliability*.

Keywords

network verification, external route, equivalence classes, symbolic simulation

ACM Reference Format:
Dan Wang, Peng Zhang, and Aaron Gember-Jacobson. 2024. Expresso: Comprehensively Reasoning About External Routes Using Symbolic Simulation. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3651890.3672220

1 Introduction

Network verification has the power to detect latent failure-induced bugs in configurations, by analyzing the forwarding behaviors under all potential *environments or scenarios* of failures. For example, network verifiers can check whether a routing/forwarding property holds under $\leq k$ arbitrary node/link failures [12, 25, 29, 30]. However, an equally important but less explored dimension of environment is *external routes*: i.e., whether a routing/forwarding property holds assuming any neighbor can advertise an arbitrary set of routes.

Many network outages are triggered when routes for certain prefixes are advertised in an unexpected way. For example, operators of a WAN informed of us a blackhole that occurred *inside* the WAN when one ISP accidentally advertised an internal prefix owned by the WAN—a scenario which the WAN operators had not anticipated. Similarly, when a large CDN mistakenly advertised more specific routes received from its peer, a peering link was overwhelmed and massive packet drops caused the peer to be inaccessible from *external* networks [6]. (See §2.1 for details.)

Since neighbors can theoretically advertise an arbitrary set of routes, *reasoning about the external route environment requires exploring a colossal space*. Firstly, the number of possible prefixes a neighbor may advertise is quite large: $2^{33} - 1$ for IPv4 prefixes[1]. Secondly, a neighbor typically advertises multiple prefixes simultaneously: e.g., a neighbor may advertise distinct prefixes for each of its customers, or a neighbor may advertise a prefix (e.g., /16) and several sub-prefixes (e.g., /24) for traffic engineering purposes. Different combinations of prefixes can result in different forwarding behaviors due to the longest prefix match semantics of the data plane and, if configured, route aggregation in the control plane. Thus, the space of combinations to explore for a single neighbor can be as large as $2^{(2^{33}-1)}$. Thirdly, many networks have tens or hundreds of neighbors that each advertises different combinations of prefixes: e.g., one neighbor may advertise a less-specific (e.g., /16) prefix and another neighbor may advertise more-specific (e.g., /24) sub-prefixes. Checking policies such as egress preferences requires considering different combinations of route advertisements from different combinations of neighbors. This space can be as large as $(2^{(2^{33}-1)})^n$ for a network with n neighbors. Finally, route advertisements may contain different attributes—e.g., AS path, communities—that can influence how a route advertisement is treated, which further grows the space to consider.

Many existing verifiers are ill-suited for reasoning about arbitrary external route environments, because the verifier requires a concrete set of route advertisements [10, 13, 15, 17, 22, 29, 30], and enumerating the aforementioned space is intractable. Some verifiers allow external neighbors to advertise an arbitrary set of routes by making routing advertisements symbolic [2, 12, 19, 27], but the underlying solvers cannot efficiently accommodate the large space of variables this requires. (See §2.3 for details.)

Fortunately, there exist equivalencies in the space of external routes. *(1) Prefix equivalence:* prefixes that match the same (or equivalent) route policies throughout the network will be treated the same during route computation. For example, if a route policy is defined for 10.0.0.0/16 ≥24, which encompasses all advertisements for sub-prefixes of 10.0.0.0/16 with a prefix length ≥ 24, then advertisements for 10.0.1.0/24 and 10.0.2.0/24 will be treated the same.

[1]For each prefix length i, there are 2^i prefixes. therefore, the total number of IPv4 prefixes is $\Sigma_{i=0}^{32} 2^i = 2^{33} - 1$.

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672220

(2) Advertiser equivalence: for a specific prefix, whether or not a neighbor advertises the prefix does not impact the best routes. For example, if the network sets a high local preference for advertisements for 10.0.0.0/8 received from one neighbor, then this route will always be the best route, regardless of whether or not other neighbors advertise the prefix.

This paper introduces a new network verifier, called Expresso, which exploits these equivalences to scale the analysis of routing/forwarding properties to all external route environments. Expresso uses a symbolic simulation approach inspired by SRE [30]. SRE makes link up/down state symbolic using boolean variables (one per link), and simulates the control and data plane to discover failure scenarios that result in the same best routes and/or forwarding paths. However, instead of introducing "dummy" nodes representing prefixes advertised by external neighbors and using $(2^{33}-1)\times n$ "link" variables (details in §3.1), Expresso makes *prefixes and external neighbors' advertising of those prefixes* symbolic.

In the control plane, the computation for different prefixes is (mostly) independent,[2] so $38+n$ boolean variables is sufficient: 32 for the (IPv4) address, 6 for the (IPv4) prefix length, and a single variable for each neighbor indicating whether or not the neighbor advertises the implied prefix. The Expresso Path Vector Protocol (EPVP) operates on route advertisements with symbolic prefix and advertiser variables—as well as symbolic AS paths and communities—to simulate route computation, and solve the canonical stable paths problem [20], for arbitrary external route advertisements.

In the data plane, longest prefix match semantics create dependencies between related prefixes—e.g., a packet will match a forwarding rule with a /24 prefix if a forwarding rule with a more specific /26 prefix does not exist—so using a single boolean variable for each external neighbor is insufficient. Thus, Expresso introduces a boolean variable for each prefix length for each neighbor—a total of $38 + 32n$ variables—and converts symbolic routes into symbolic forwarding rules that account for longest prefix match semantics.

We implement Expresso and use it to check the configurations from a large cloud provider's WAN and Internet2. Expresso finds 187 misconfigurations that may cause route leaks, route hijacks, and traffic hijacks when specific neighbors advertise certain routes; 42 of the misconfigurations have already been confirmed and fixed by the operators.

Contributions. In sum, our contributions are:

- We show the necessity of reasoning about arbitrary external routes, when checking network configurations.
- We design and implement Expresso, a verifier that can comprehensively reason about external routes in a scalable way, with symbolic simulation.
- We use Expresso to check the WAN of a large cloud service provider (CSP), and find misconfigurations that may cause route leaks, route/traffic hijacks.
- We evaluate the performance of Expresso on the WAN of CSP and Internet2, and show that Expresso is the only verifier that can finish (within a hour), while others time-out after one day.

[2]Route aggregation introduces limited dependencies (details in §3.1).

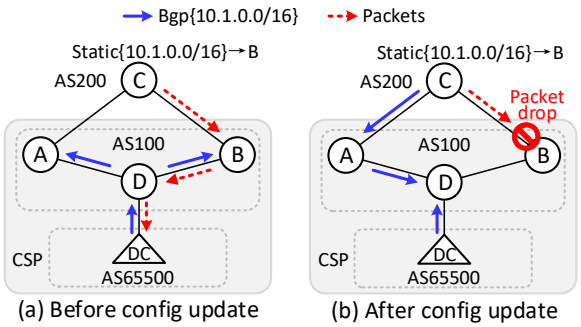


Figure 1: A blackhole in a large cloud service provider.

2 Motivation

In this section, we motivate the need to check the correctness of configurations against *arbitrary* external routes, using two real outages caused by unexpected external routes. Then, we define verification tasks that can detect those outages, and discuss the limitations of existing verifiers for those tasks.

2.1 Why Arbitrary External Routes?

Many issues like route leaks only manifest upon receipt of some unexpected route advertisements from neighbors. To show this, we use two real incidents, one from a cloud service provider, and the other from a content deliver network.

Case 1: An internal blackhole due to unexpected external route advertisements. Figure 1 shows a real incident of a large cloud service provider (CSP). The CSP has tens of datacenters across the world, which are connected by its own wide area network (WAN) with AS number 100. The WAN peers with hundreds of ISPs at different Points of Presence (PoPs). At one PoP, there are two routers A and B connecting to ISPs, and a router D connecting to the datacenter (DC, with private AS number 65500). D advertises a prefix 10.1.0.0/16 received from the datacenter to A and B. A peers with a router of ISP C using BGP; for B, however, C configures a static route for the prefix 10.1.0.0/16 with B as the next hop. Initially, router A is configured to only advertise the default route to router D, with a configuration command "advertise-default". One day, the operators remove this command to let A advertise all routes received from the Internet to datacenters (D in this example), so that datacenters can flexibly choose their respective best PoP to reach the Internet. However, after the change, the prefix 10.1.0.0/16 becomes unreachable from the Internet.

After checking the routing tables, the operator found that router A receives a route for 10.1.0.0/16 from router C, who isn't supposed to advertise this route. This route propagates to router D through BGP. Since this BGP route has a higher priority than the route received from the datacenter, it becomes the best route at D. Then, D stopped advertising the route for 10.1.0.0/16 to B, since A, B and D are iBGP neighbors and BGP prevents an iBGP peer from re-advertising that route [18]. This leads to a blackhole at router B: all traffic from the Internet to the internal prefix 10.1.0.0/16 are dropped at B.

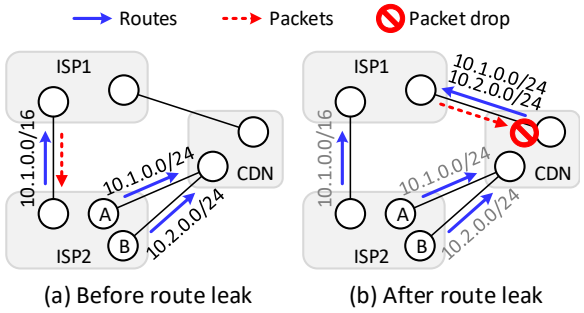


Figure 2: A real route leak on the Internet.

Case 2: A route leak on the Internet. Figure 2 shows a typical route leak on the Internet [6], simplified for illustrative purposes. A national ISP2 hosts millions of customers with a block of IP addresses 10.1.0.0/16. and advertises this /16 prefix to another ISP1. Since ISP2 exchanges a massive volume of traffic with a content delivery network (CDN), the operator de-aggregates the /16 prefix into multiple more specific /24 prefixes, and advertises different /24 routes to the CDN at different PoPs (router *A* and *B*). Normally, the CDN should not export those specific routes to its peers. However, one day operators of the CDN made a misconfiguration, advertising the /24 prefixes to its neighbors, including ISP1. As a result, ISP1 began to choose the CDN instead of ISP2 as the next hop, and a large volume of traffic from ISP1 to ISP2 congested at the CDN, disconnecting millions of customers of ISP2 from the Internet.

Lessons learnt. (1) External route advertisements can impact the reachability of internal networks, and may cause violations of reachability properties inside the network. Verifiers that focus on checking reachability inside a closed environment may miss those violations. (2) It is insufficient to assume that neighbors advertise a specific set of routes, since some violations only manifest upon receiving some unexpected routes. Therefore, operators from one network need to consider arbitrary external routes, so that they do not pay for the mistakes made by operators of other networks.

2.2 Verification Tasks

Detecting outages caused by external routes requires checking two categories of end-to-end network behaviors:

- *Routing properties* specify how routes satisfying certain constraints should propagate inside the network, e.g., routes advertised by a neighbor of router A should never be exported by another router B to any of its neighbors. Checking routing properties requires modeling the route computation behaviors (control plane).
- *Forwarding properties* specify how packets satisfying certain constraints are forwarded inside the network, e.g., all internal prefixes should be reachable from the internal network and the Internet. Checking forwarding properties requires modeling the packet forwarding behaviors (data plane).

It is essential to model the routing and forwarding behaviors in an end-to-end manner, because routes and packets may traverse multiple hops inside a network, where each hop performs a subset of operations.

2.3 Related Work

Control plane verification (CPV) reasons about the correctness of router configurations against specific routing or forwarding properties, under some potential *environment states*. Considering environment state enables CPV to detect latent bugs that only manifest under specific environments, e.g., two links fail simultaneously, a neighbor advertises a specific set of routes. Modeling link failures has been extensively studied [12, 19, 23, 25, 29, 30]. Here, we discuss how existing verifiers handle the impact of external routes.

No external routes. Some verifiers [10, 22, 29, 30] have not explicitly discussed how to handle the impact of external routes. While these verifiers can add "dummy" nodes that advertise specific sets of prefixes, this simple workaround does not allow those verifiers to detect latent bugs due to arbitrary, unexpected external routes.

Concrete external routes. A few verifiers consider concrete sets of external routes. Batfish [17] lets users specify a list of route advertisements received from each neighbor. ERA [15] uses a BDD to represent route advertisements received from each neighbor, so as to model a flexible set of advertisements—e.g., users can let a neighbor simultaneously advertise all prefixes using the BDD `True`, instead of providing a concrete list of all IPv4 prefixes. However, ERA still considers one concrete environment (each neighbor advertises a *specific* set of routes) at a time, and needs to enumerate environments to cover all cases where each neighbor advertises an *arbitrary* set of routes. Similarly, ShapeShifter [13] can consider a batch of external routes by abstracting away some route attributes (e.g., AS Path). However, this abstraction incurs some loss of precision, and, like ERA, still entails a concrete assumption about external routes.

Symbolic external routes. Some verifiers [2, 12, 19, 27] allow each neighbor to advertise an arbitrary set of external routes by making the route advertisement of the neighbor symbolic. Bagpipe [27] models the BGP route computation and the properties to check with SMT constraints, and uses off-the-shelf solvers (e.g., Z3) to prove whether the properties hold. Minesweeper [12] and NV [19] also use SMT, but their models are more general, including interior gateway protocols like OSPF. Even thought these SMT-based verifiers avoid enumerating all possible external routes, their scalability is still limited due to the large number of SMT constraints. For example, Minesweeper generates over 1.6 million SMT constraints—which takes hundreds of seconds to solve—for a single region of a cloud service provider's WAN (10 nodes, 40 neighbors), and Minesweeper times out after a day when applied to the full WAN (30 nodes, 100 neighbors). The latest version of Batfish [2] provides a question named `SearchRoutePolicies`, which uses BDDs to search for routes that exhibit a particular behavior (permit or deny) for a specific route policy. However, such a unit test on route policies can miss bugs caused by mistakes in configurations beyond route policies (e.g., iBGP sessions [16]). For example, a missing command in BGP peer configuration can also lead to a route leak, even though the routing policies are correct (§3.2).

Monitoring and auditing systems. Authoritative organizations (e.g., RIPE NCC [9]) have been encouraging ASes to register their BGP information (e.g., prefixes and routing policies) for years. Given that information, ASes can use monitoring or auditing systems (e.g.,

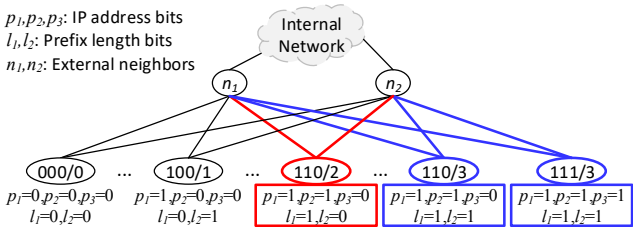


Figure 3: Illustration of the key insights.

[1, 3, 4]) to detect route leaks and/or hijacks. However, this approach is hard to promote as ASes are reluctant to register their information since it is usually private, and it can only detect routing anomalies passively instead of preventing them proactively. We believe that enabling a single AS to proactively reason about external routes without global coordination is more promising.

3 Overview

We now discuss the key insights of Expresso and show the workflow of Expresso with an example network. For simplicity of illustration in this section, we assume an IP address has only three bits, denoted by p_1, p_2, p_3. Then, the prefix length can be represented with two bits l_1, l_2 (e.g., "11" means a prefix length of 3).

3.1 Key Insights

Expresso is inspired by SRE [30], a network verifier which can reason about a large space of failure scenarios in a scalable way. SRE achieves the scalability by using symbolic simulation (execution), where equivalent failure scenarios are compressed by making link state (up or down) symbolic.

An intuitive approach for reasoning about arbitrary external routes is to make the external route environment symbolic. Specifically, suppose a network has n external neighbors, each represented by a node, as shown in Figure 3. For each of the $2^4 - 1$ prefixes possible with 3-bit IP addresses, we include a virtual node and link it to each neighbor node. The link between a neighbor and a prefix node is said to be up/down, if and only if the neighbor advertises/does not advertise that prefix. In this way, the up/down state of all $(2^4 - 1) \times n$ links represents an external route environment. Using SRE in this manner with IPv4 addresses requires $(2^{33} - 1) \times n$ links and corresponding boolean variables. Such a large number of variables is infeasible for any known BDD library or SMT solver.

Expresso is based on the key observation that prefixes are *mostly* independent of each other in the control plane, and have *limited* dependency in the data plane, so we can use far fewer boolean variables for the symbolic route environment.

Independency in the control plane. In the control plane, the computations of routes for different prefixes do not interfere with each other—except in the case of route aggregation, which we discuss later[3]. Due to such independency, we only need to reason about the advertisement of one prefix (i.e., one prefix node's "links") at a time, so a single variable is enough for each neighbor. Specifically, we can use 5 variables to identify the prefix (3 for the address, 2 for the length) we are reasoning about, plus another n variables to control whether or not each neighbor announces this prefix. (For IPv4, $38 + n$ variables is enough!) For example, as shown in Figure 3, we can first "select" the prefix 110/2 (the red node) by constraining the boolean variables $p_1 = 1, p_2 = 1, p_3 = 0, l_1 = 1$, and $l_2 = 0$. Then, only two links (red) remain, and we can use another two variables n_1 and n_2 to specify whether or not each of the two neighbors advertises the prefix 110/2.

Route aggregation may introduce dependency among some prefixes with common first bits, but it can be supported with a few more variables. The upper bound of needed additional variables is determined by the shortest prefix length of all aggregated prefixes, since each aggregated prefix depends on all longer prefixes having common first bits with it. For example, if there's two aggregation rules for IPv4 prefixes, one generates a /20 prefix and the other generates a /24 prefix, then we need $12 \times n$ more variables (in the worst case).

Limited dependency in the data plane. In the data plane, prefixes become dependent due to the longest prefix matching (LPM) principle: a route matched by a packet is used for forwarding only if there are no other routes that are also matched and have longer prefix lengths. For example, in Figure 3, whether routes to prefix 110/2 (red node) are used for packet forwarding depends on whether routes to prefix 110/3 and 111/3 (blue nodes) exist. Due to the dependency, we cannot "select" the prefix first. However, such dependency is limited to those prefixes with common first bits but different prefix lengths. Thus, we can account for such dependency using a few more variables. Specifically, we show that 32 more variables per neighbor are enough for IPV4 (in the worst case), and in two real networks we study, each neighbor only needs 8 and 11 more variables on average (see §5).

3.2 Workflow of Expresso

We walk through the 3 steps of Expresso with an example.

Example network. Figure 4 shows an example network with two routers (PR_1 and PR_2) peering with two ISPs (ISP_1 and ISP_2). Both PR_1 and PR_2 permit two prefixes (100/2 and 110/2) from the ISPs, and PR_1 sets the local preference of routes from ISP_1 to 200 (the default is 100), in order to prefer ISP_1 to ISP_2 as the exit to reach the Internet. PR_2 announces an internal prefix 000/2.

The operators of the network configure import/export routing policies on the two *PR*s following best practices: attach a specific community (i.e., 300:100) to incoming routes from ISPs, and deny outgoing routes with that specific community towards ISPs. Additionally, the session property `advertise-community` are configured between *PR*s to include communities in all exported routes towards each other. However, the operator made a misconfiguration: forgetting the `advertise-community` command when configuring PR_1, which results in a route leak from ISP_1 to ISP_2.

(1) Symbolic Route Computation (SRC) takes configurations and topology as input, and computes symbolic RIBs. Unlike concrete RIBs, which contain concrete routes under a specific external route environment, a symbolic RIB contains symbolic routes that can materialize into different concrete routes under different external route environments.

In Expresso, *a symbolic route is a set of concrete routes*, represented as a tuple of $(\mathbf{pln}, asp, comm, lp, nh, o)$, where $\mathbf{p} = p_1p_2p_3$

[3]Other dependencies such as BGP conditional route advertisement are discussed in §8.

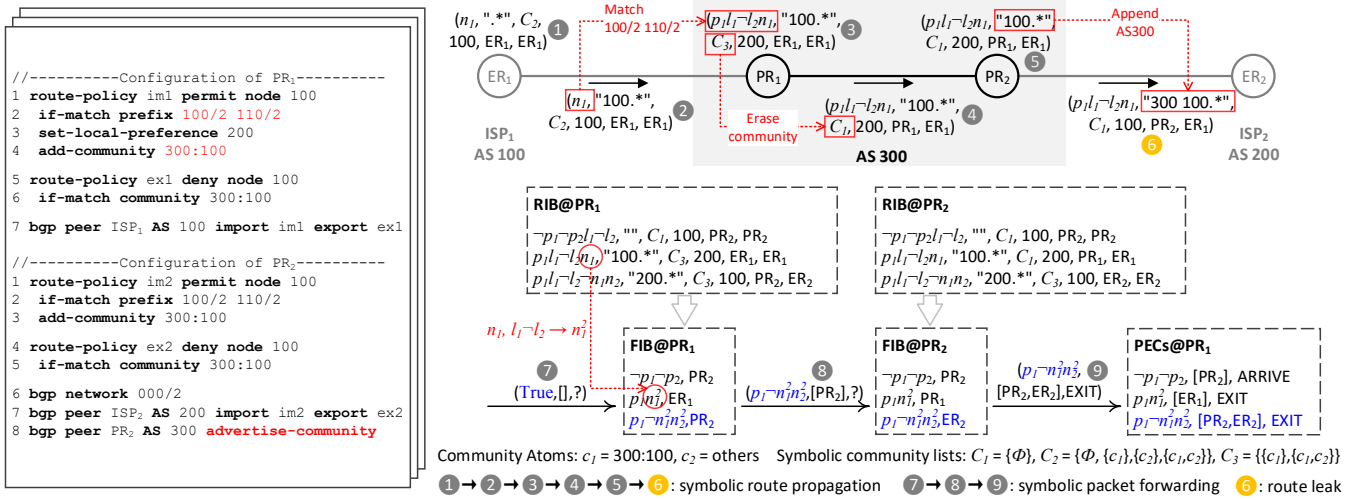


Figure 4: An example network with configurations and Expresso's workflow.

is an IP address ($[000, 111]$), $\mathbf{l} = l_1 l_2$ is a prefix length ($[0, 3]$), $\mathbf{n} = n_1 n_2$ is an advertiser condition, *asp* is a regular expression for AS path, *comm* is a set of community lists; *lp*, *nh* and *o* is the local preference, next-hop and originator[4], respectively. Returning to the example in Figure 4, the second RIB entry of PR_1 ($p_1 l_1 \neg l_2 n_1$, "100.*", C_3, 200, ER_1, ER_1) represents a set of concrete routes whose prefixes can either be 100/2 ($\mathbf{p} = 100, \mathbf{l} = 10$) or 110/2 ($\mathbf{p} = 110, \mathbf{l} = 10$), AS paths start with 100, community lists that contains c_1 (i.e., 300:100, since $c_1 = \bigcap_{l \in C_3} l$), local preference is 100, and next-hop and originator are both ER_1. Finally, the advertiser condition n_1 indicates that any of these routes materialize when the corresponding prefix is advertised by ISP_1.

Expresso computes symbolic routes by executing the *Expresso path vector protocol (EPVP)*, a symbolic variant of the Simple Path Vector Protocol (SPVP) [20], which works as follows. First, for each external neighbor i, EPVP initializes an *wildcard* symbolic route (standing for the universe of all possible routes), with advertiser condition n_i, and all other symbolic fields wildcarded , e.g., $\mathbf{pl} \leftarrow$ `True`, $asp \leftarrow$ ".*", $comm \leftarrow 2^{\{c_1, c_2\}}$. Then, EPVP advertises the symbolic routes to the network, and iteratively calculates the best (symbolic) routes for each router, until reaching a fixed point. Since EPVP works on symbolic routes, it must use symbolic versions of `transfer` (for transforming or filtering routes) and `merge` (for selecting the best routes) functions. Details are in §4.3.

(2) Symbolic Packet Forwarding (SPF) takes the symbolic RIBs as input, forwards symbolic packets through the network, and generates a set of packet equivalence classes (PECs). Each PEC consists of all concrete packets, each associated with a specific external route environment, that have the same forwarding path in the network.

Figure 4 shows three PECs whose forwarding path begins at PR_1. The third PEC ($p_1 \neg n_1^2 n_2^2$, $[PR_2, ER_2]$, EXIT) represents a set of packets following the path $PR_1 \rightarrow PR_2 \rightarrow ER_2$, with destIP matching $1**$ and ISP_2 (but not ISP_1) announcing the /2 prefixes—i.e., 100/2, or 110/2, or both ($\neg n_1^2 n_2^2$). Note here we replace the n_i variable with n_i^j variables, where j represents the prefix length. This is to account for the LPM semantics during packet forwarding (see §5 for details).

To generate PECs, Expresso converts the symbolic RIBs into symbolic FIBs, where each FIB entry includes a predicate for the variables of **pl**, **n**, and the output port. As discussed in §3.1, in data plane, LPM introduces dependency among prefixes, but the dependency is limited and prefixes with the same length are still independent. Therefore, we extract all prefix lengths (at most 33) from a RIB entry to generate corresponding FIB entries. The conversion contains two steps. Firstly, we convert each RIB entry into several FIB entries, one for each prefix length. This is done by iteratively extracting all prefixes of a specific length (longest to shortest) from the RIB entry. Then, we combine the length and advertiser condition to form data plane advertiser condition (e.g., from $p_1 l_1 \neg l_2 n_1$ to $p_1 n_1^2$). After generating FIBs, SPF follows the same symbolic forwarding procedure as SRE, i.e., augments each packet with a symbolic header, i.e., the predicate `True` (encoding all destinations and advertiser conditions), and injects it at each router in the network.

(3) Property Analysis of various routing and forwarding properties can be done based on output of the above two stages. First, Expresso can analyze routing properties, e.g., `RouteLeakFree` and `RouteHijackFree`, by checking the existence or absence of specific routes right after the first stage SRC. Take `RouteLeakFree` for example, Expresso checks for each neighbor, whether it receives any route that is originated by another neighbor. As shown in Figure 4, ER_2 receives a route whose `originator` is ER_1 of ISP_1, indicating this is a route of ISP_1 leaked by the network to ISP_2. Forwarding properties, e.g., `BlackHoleFree`, `LoopFree`, can be analyzed after the second stage SPF is done, as the analyses require forwarding paths or final states (e.g., ARRIVE and EXIT in Figure 4) of PECs.

[4]The `originator` of a symbolic route should be distinguished from the `origin` field of a concrete route. For each symbolic route R, Expresso records its propagation path, and `originator` ($R.o$) is the first hop of the path.

Take `LoopFree` for example: Expresso checks whether there are PECs whose final state is LOOP because of visiting a node twice during symbolic packet forwarding. As shown in Figure 4, packets starting from PR_1 don't encounter any loop.

4 Symbolic Route Computation

In this section, we introduce routing algebra and simple path vector protocol (§4.1), which serve as a framework for SRC. Then we show how to make routes symbolic (§4.2), and how to operate on symbolic routes (§4.3).

4.1 Routing Algebra and Simple Path Vector Protocol

Routing Algebras [24] define the basic semantics of route computation in the control plane. $A(\Sigma, \oplus, F, 0, \infty)$ is an algebra where:

- Σ is a set of *signatures* that describes characteristics of a route. A route has the form $r = (d, \sigma)$, where d is the destination (i.e., a prefix) and $\sigma \in \Sigma$ is the signature. Taking BGP as an example, a signature includes local preference, AS path, communities, etc.
- $\oplus : 2^{\Sigma} \rightarrow 2^{\Sigma}$ is the *merge function*, which takes a set of signatures as input, and selects the most preferred signatures, considering Equal Cost Multi-Path (ECMP).
- F is a family of *transfer functions*, with f_{uv} denoting the transfer function of link (u, v), which processes the signatures traversing from u to v. For example, in BGP, when traversing from u to v, a signature $\sigma \in \Sigma$ goes through the export policy (e_u) of u first, then the import policy i_v of v, which means $f_{uv} = i_v \circ e_u$.
- 0 is the initial signature.
- ∞ is the invalid signature, used to denote the absence of a route to the destination.

Simple Path Vector Protocol (SPVP) [20] defines a fix-point algorithm (Algorithm 1) that takes a network $G(V, E)$ (V is the vertex set and E is the edge set) as input, and iteratively compute the routes to a destination (i.e., prefix) using the merge and transfer functions defined by a routing algebra. For each vertex $u \in V$, SPVP uses $Best_u$ to track the most preferred (i.e., best) routes, and $Received_u$ to track the routes received from peers. Before the iterations start, SPVP initializes the $Best_u$ of each node (line 1) to either an empty set (i.e., $Best_u \leftarrow \{\}$) or a set containing the default signature (i.e., $Best_u \leftarrow \{0\}$), depending on whether u holds the prefix. Then, during each iteration, for each vertex $u \in V$, SPVP first collects candidate routes from its neighbors (line 11):

$$\forall (u, v) \in E, Recv_u \leftarrow Recv_u \cup \{f_{uv}(\sigma) | \sigma \in Best_v)\},$$

then merges them with the current best routes (line 12):

$$Best_u \leftarrow \oplus(Best_u, Recv_u).$$

SPVP converges when $Best_u$ of every vertex u remains unchanged after an iteration (line 14).

Limitations of SPVP. Although SPVP is widely used by verifiers as the control plane engine [10, 13, 17, 22, 29, 30], Expresso cannot use SPVP for the following reasons. Firstly, SPVP is designed to compute routes for one prefix at a time, so we need to run SPVP $(2^{33} - 1)$ times to cover all prefixes. Secondly, SPVP is designed to compute routes for one environment at a time, by including the external nodes that advertises destination d in G, so we need to run SPVP 2^n times to consider the uncertainty of whether each external neighbor n_i advertises d or not.

Therefore, we introduce Expresso Path Vector Protocol (EPVP), a symbolic variant of SPVP. EPVP operates on symbolic routes (§4.2) with symbolic versions of initialize, merge and transfer functions (§4.3).

4.2 Symbolic routes

For BGP, a route advertisement includes prefix, AS path, communities, local preference, MED, etc. To consider external environment, we also include another dimension of *environment*, a bit vector indicating whether a neighbor advertises the prefix or not. Note the environment dimension is used purely for analysis rather than modeling route computation in real networks.

In Expresso, a symbolic route R is defined as a set of *prefix-environment pairs*, a set of AS paths, and a set of community lists, and the shared route attributes:

$$\begin{aligned} R &= (\{(d_0, e_0), (d_1, e_1), \cdots\}, \langle\{a_0, a_1, \cdots\}, \{c_0, c_1, \cdots\}, attr\rangle \\ &= (\mathcal{D}, \langle asp, comm, attr\rangle). \end{aligned} \tag{1}$$

It collectively represents a set of concrete routes, which we call the *unfolding* of R and denote as $\overline{R}$:

$$\overline{R} = \{(d, e, a, c, attr) | (d, e) \in \mathcal{D}, a \in asp, c \in comm\}. \tag{2}$$

A symbolic route is a route whose prefixes and attributes are symbolic values (variables). Every symbolic route encodes an equivalence class of concrete routes, i.e., routes that are treated the same by routers in the network. But note that routes that are treated the same may be in different symbolic routes due to such an encoding. For example, suppose there are two concrete routes, whose AS path and community list are "[100], {300:100}" and "[200], {300:200}", respectively. Even if they are treated the same by a route filter, they can't be represented by a single symbolic route with a symbolic AS path and a symbolic community list.

In the following, we show how Expresso encodes each part of a symbolic route.

IP prefix. An IP prefix is a ternary bit (0, 1, and $*$) vector and a prefix length (0-32). It can be represented as a predicate (formula) over a set of boolean variables. Similar to previous verifiers, Expresso uses a Binary Decision Diagram (BDD) [11], a canonical representation for boolean formulas, to encode a symbolic IP prefix. Specifically, Expresso uses 38 bits (BDD variables) for the symbolic IP prefix—32 bits for the prefix and 6 bits for the prefix length.

Advertiser condition. The advertiser condition specifies the external route environments under which routes are computed. It is similar to the failure condition introduced by SRE, which specifies the failure scenarios (whether each node or link in the network is up or down) under which routes are computed. The difference is that failure condition is a global condition for all prefixes, while advertiser condition is local, i.e., with respect to a specific prefix in the symbolic IP prefix. As a result, different IP prefixes can multiplex the advertiser condition of a neighbor.

Suppose the network has n external neighbors, with each peering with some routers of the network. For the ith neighbor, we use a boolean variable (bit) n_i to represent whether the neighbor advertises a specific route or not. Therefore, the advertiser condition

is a bit vector of fixed length, and can be encoded with BDD, in a similar way to prefix.

Community list. To encode symbolic community lists, we take inspiration from Batfish's `SearchRoutePolicies` feature. In particular, we pre-compute a set of atomic predicates (atoms) over all communities that appear in the configurations. This way, a symbolic community list can be encoded by a set, where each item is an atom (integer) set that represents a concrete community list. Operations on community lists are converted to set operations.

For example, suppose there are two communities, 300:100 and 300:[1-9]00, in the configurations. Then there are three community atoms, i.e., c_1 = 300:100, c_2 = 300:[2-9]00, and c_3 represents all other communities. The symbolic community list representing any concrete community lists is, $C = 2^{\{c_1,c_2,c_3\}}$. Adding community 300:100 to C equals adding atom c_1 to every member set of C, i.e.,

$$C \leftarrow C \times (2^{\{c_1\}}/\emptyset) = \{\{c_1\}, \{c_1, c_2\}, \{c_1, c_3\}, \{c_1, c_2, c_3\}\}.$$

AS path. While Batfish's `SearchRoutePolicies` also pre-computes atomic predicates for AS path, we found it really inefficient[5], since element order matters in AS paths and regex matching is applied to the whole AS path[6]. Therefore, Expresso uses automaton (a form equivalent to regexes) to represent symbolic AS paths. Operations on AS paths are converted to automaton operations.

For example, the symbolic AS path representing any concrete AS path is A = ".*" (i.e., any string). When appending AS number 100, A is updated by the concatenation of the automaton of string "100" and A, i.e., $A \leftarrow str(\text{"100"}).\text{concat}(A)$. When processed by an AS path filter matching ".*400" , A is updated by the intersection of the automaton of regex ".*400" and A, i.e., $A \leftarrow re(\text{".*400"}).\text{intersect}(A)$.

Other attributes. Expresso uses concrete (default) values for other attributes (e.g., local preference, origin, and MED). For non-transitive attributes like local preference, they will be erased when importing from neighbors, and therefore do not affect our analysis. For transitive attributes like MED and origin, although their values may be non-default in external routes, we choose to make them concrete, in order to have a deterministic result when comparing two symbolic routes.

4.3 Processing symbolic routes

Expresso uses EPVP, a variant of SPVP, to compute symbolic routes. The procedure of EPVP is similar to SPVP, but with a different initialization process, transfer function, and merge function.

***Initialize* symbolic routes.** Instead of initializing concrete routes only for internal routers, EPVP initializes symbolic routes for both internal routers and external neighbors as follows.

(1) For each internal router u, $Best_u$ is initialized as a symbolic route with $\mathcal{D} = \bigvee_{d \in D_u} d$ (D_u is the prefixes originated by u), asp = "" (empty string), $comm = \{\emptyset\}$ (no communities) and $attr$ having default values. Note that the environment for every $d \in D_u$ is `True`, since it will always be advertised no matter in what environment.

(2) For each external neighbor v, $Best_v$ is initialized as a symbolic route with $\mathcal{D} = n_v$, asp = ".*" (any string), $comm = 2^{AP}$ (the power set of community atoms AP) and $attr$ with default values. The above indicates the neighbor advertises an arbitrary set of prefixes, and for each advertised prefix, the AS Path and community list can be arbitrary.

For example, in Figure 4, the symbolic route of internal node PR_2 is initialized to $(\neg p_1 \neg p_2 l_1 \neg l_2, \langle\text{""}, \{\emptyset\}, 100, PR_2, PR_2\rangle)$, and that of external ER_i $(i = 1, 2)$ is initialized to $(n_i, \langle\text{".*"}, 2^{\{c_1,c_2\}}, 100, ER_i, ER_i\rangle)$.

***Transfer* symbolic routes.** Unlike SPVP where a transfer function (i.e., a node's route policies) deterministically transforms a (concrete) route, the transfer function in EPVP may transform a symbolic route ambiguously. The reason is that a symbolic route consists of a set of concrete routes, which may be transformed differently by the same transfer function. For example, suppose there is a symbolic route R with community list $\{\{c_1\}, \{c_1, c_2\}\}$, and a transfer function f which (1) matches $\{c_1\}$ and sets local preference to 200, and (2) matches $\{c_1, c_2\}$ and sets local preference to 300. Applying f on R will result in two symbolic routes.

Therefore, instead of defining a single transfer function as an if-then-else program in SPVP [20], we model a transfer function of EPVP as a set of $(predicate, function)$ pairs, each of which ambiguously transform symbolic routes satisfying the predicate[7]:

$$f = \{(\alpha_1, f_1), (\alpha_2, f_2), \cdots, (\alpha_n, f_n)\}, \tag{3}$$

where α_i is a predicate over a symbolic route, satisfying $\bigvee_{i \in [1,n]} \alpha_i =$ `True`, and f_i is a transfer function for routes satisfying α_i. Then, applying the transfer function f on a symbolic route R results in

$$f(R) = \{f_i(\alpha_i \wedge R) | i \in [1, n]\}, \tag{4}$$

where $\alpha_i \wedge R$ means to constrain R with the predicate α_i. In the above example, suppose $R = (p_1 l_1 \neg l_2, \langle\{\{c_1\}, \{c_1, c_2\}\}, 100\rangle)$ (for route attributes, we show only symbolic community list and local preference for simplicity). Then, we have:

$$f(R) = \{(p_1 l_1 \neg l_2, \langle\{\{c_1\}\}, 200\rangle), (p_1 l_1 \neg l_2, \langle\{\{c_1, c_2\}\}, 300\rangle)\}$$

***Merge* symbolic routes.** Since a symbolic route is essentially a set of concrete routes, merging two symbolic routes R_1 and R_2 is actually merging every $r_1 \in \overline{R_1}$ and every $r_2 \in \overline{R_2}$ pairwise, and we have to consider two cases: (1) r_1 and r_2 have the same prefix and environment, and (2) r_1 and r_2 have different prefixes and/or different environments. For the first case, r_1 and r_2 are comparable, so only the more preferred one stays after merging R_1 and R_2. For the second case, r_1 and r_2 are incomparable, so they both stay after merging R_1 and R_2. Therefore, when merging two symbolic routes, we can't select a single best symbolic route. Instead, we drop the concrete routes in R_1 and R_2 that cannot be selected as best routes. Formally, dropping the concrete routes in R_2 with respect to R_1 is defined as ($attrs = \langle asp, comm, attr\rangle$, ρ represents the preference of $attrs$, defined by routing protocols):

$$R_2 - R_1 = \begin{cases} (\mathcal{D}_2, attrs_2), & \text{if } \rho(attrs_1) \geq \rho(attrs_2) \\ (\mathcal{D}_2 \wedge \neg\mathcal{D}_1, attrs_2), & \text{if } \rho(attrs_1) < \rho(attrs_2) \end{cases}.$$

This way, merging R_1 and R_2 is defined as:

$$R_1 \oplus R_2 = \{R_1 - R_2, R_2 - R_1\}. \tag{5}$$

One may wonder how the function ρ can compare the preference of two symbolic routes, whose attributes (i.e., asp and $comm$) are variables. Firstly, community lists are not used for best route selection, and therefore do not affect the comparison. Secondly, as

[5] Computing atomic predicates for AS path times out in 1 hour on our datasets (§7.2).
[6] On the contrary, element order doesn't matter in community list, and regex matching applies to separate elements in a community list.

[7] How to compute these pairs is shown in Appendix B.

for AS paths, BGP prefers routes with shorter AS path length. To support this, we choose the path with the shortest length from the symbolic path as the representative, and use the length of this path for comparison, e.g., a symbolic AS path "100.*" has a length of 1 during comparison. For example, suppose we have two symbolic routes (for route attributes, we show only symbolic AS path and symbolic community list for simplicity, other attributes of R_1 and R_2 are identical):

$$R_1 = (p_1 l_1 \neg l_2 n_1, \langle \text{"100.*"}, C_1 \rangle)$$
$$R_2 = (p_1 l_1 \neg l_2 n_2, \langle \text{"200,200.*"}, C_3 \rangle)$$

R_1 is more preferred than R_2 since it has "shorter" symbolic AS path. Merging them results in:

$R_1 \oplus R_2 = \{(p_1 l_1 \neg l_2 n_1, \langle \text{"100.*"}, C_1 \rangle), (p_1 l_1 \neg l_2 \neg n_1 n_2, \langle \text{"200,200.*"}, C_3 \rangle)\}$.

The first part of the result represents that all concrete routes in R_1 are selected as best routes, since their attributes have higher preference. The second part of the result represents that the concrete routes in R_2 that have no counter-parts (i.e., having the same prefix and environment) in R_1 can be selected as best routes, because of being the only candidate.

Correctness of EPVP. We prove the correctness of EPVP in Appendix §D.

5 Symbolic Packet Forwarding

5.1 Generating Symbolic FIBs

For each router, Expresso generates a symbolic FIB, an ordered list of forwarding rules $(match, port)$, where $match$ is a predicate (boolean formula) over both packet headers and *advertiser conditions*. The advertiser condition specifies "when" (some neighbors advertise a specific set of routes) the rule will materialize. Unlike SRE, Expresso cannot directly convert symbolic RIBs into symbolic FIBs, since the routes (e.g., prefixes) are symbolic.[8]

Taking a symbolic route $p_1 l_2 n_1$ for example, it contains prefixes with two lengths (i.e., 100/1, 100/3, 101/3, 110/3, and 111/3). When a packet with destIP 101 comes, it will firstly match the 101/3 prefix if it exists ($n_1 = 1$); otherwise, if the rule does not exist ($n_1 = 0$), the packet will match 100/1 if it exists ($n_1 = 1$, contradicting with $n_1 = 0$). Therefore, using a single variable n_1 to represent the condition ISP1 advertises the route is not enough any more.

To solve this problem, we need more variables for advertiser condition. Rather than using one variable for each prefix ($(2^{33} - 1)$ variables in total), we observe that one variable for each prefix length (33 variables in total) is sufficient: if a symbolic route has multiple prefix lengths, then we split it into multiple routes, one for each possible prefix length. In addition, if the advertiser condition of the original route has a variable n_i then for each of these prefix length j, create a new advertiser condition n_i^j to replace n_i. Therefore, in practice, we need less than 32 additional variables: in two real network snapshots we study, each neighbor only needs 8 and 11 more variables on average. Returning to the previous example, the original symbolic route is split to two symbolic routes, one for each prefix length—$p_1 \neg l_1 l_2 n_1$ and $p_1 l_1 l_2 n_1$. Then, we replace the advertiser condition n_1 with new advertiser conditions, i.e., $p_1 n_1^1$ and $p_1 n_1^3$. Note that the prefix length bit l_i is not needed anymore since it become concrete, as well as n_1^0 and n_1^2.

[8] SRE also terms the route as symbolic, since it carries a topology condition, while the routes themselves (prefixes and attributes) are concrete. In contrast, the routes in Expresso are symbolic.

5.2 Computing PECs

After generating symbolic FIBs, Expresso follows a similar process to compute packet equivalence classes (PECs). In the following, we briefly show the process and more details can be found in the paper of SRE.

Computing port predicates. Before simulating the forwarding of packets, Expresso first pre-computes *port predicates*, in a similar way with [28, 30]. A port predicate is a boolean formula encoding the set of packets forwarded to that port (forwarding predicates), or allowed by a specific port (ACL predicates). Such an approach makes the simulation more efficient, as will be seen later.

Forwarding symbolic packets. Then, Expresso augments each packet header with an *advertiser condition*, a predicate over boolean variables n_i^j, indicating whether the neighbor i advertises a prefix of length j or not. Initially, Expresso injects a symbolic packet at each router (either internal or external), The symbolic packet matches all packet headers and advertiser conditions, by setting the packet header and advertiser variables to `True`. When the packet reaches a port of a router, the router creates a new replica of the packet, and computes the conjunction of it with the predicate of the port, and lets the replica traverse the port.

Obtaining PECs. This forwarding of symbolic packets continues until reaching a port which (1) has been visited twice with the same symbolic packet (LOOP), (2) matches a rule that drops the packet (BLACKHOLE), (3) is directly connected to the destination prefix (ACCEPTED), or (4) is not connected to any other routers (EXIT). We term the above states (i.e., LOOP, BLACKHOLE, ARRIVE and EXIT) as the final state of the symbolic packets. After every symbolic packet reaches its final state, Expresso obtains a set of PECs. Each PEC is represented by a 3-tuple $(pkt, path, fs)$, where $path$ is the forwarding path, pkt is the predicate over packet header and advertiser condition, and fs is the final state of packets in the PEC.

6 Property Analysis

This section introduces how Expresso analyzes routing and forwarding properties defined in §2.2 (§6.1 and §6.2, respectively). Then, we show how Expresso can be used to check other properties that are related to routes or packets §6.3.

Firstly, we define some notations used below. We use V_I and V_E ($V = V_I \cup V_E$) to denote the set of internal and external nodes of the network, respectively. For $u, v \in V$, we use $PECs(u)$ ($PECs(u, v)$) to denote the set of PECs whose forwarding path starts from u (and ends at v). The set of internal and external prefixes are represented as D_I and D_E, respectively. For each symbolic route R, we use $R.o$ to denote its `originator`.

6.1 Routing properties

Expresso analyzes the routing properties by checking the existence or absence of specific routes in specific routers' symbolic RIBs.

RouteLeakFree specifies that routes received from a peer/provider should never be leaked to another peer/provider. Checking this property can detect cases where the network becomes a transit

between peers/providers. Expresso checks this property by: for every external node $u \in V_E$, and every symbolic route $R \in \text{RIB}(u)$, check if R originated from an internal node or itself, i.e.,

$$R.o \in V_I \cup \{u\}.$$

RouteHijackFree specifies that routes originating from a neighbor should never be selected as the best routes for an internal prefix. Checking this property can prevent internal traffic of the network from being leaked to the Internet. Expresso checks this property by: for every internal node $u \in V_I$, every internal prefix $d \in D_I$, and every symbolic route $R \in \text{RIB}(u)$, if R is an internal route and contains prefix d, check whether its advertiser condition for prefix d is `True` (i.e., no external route to d will be more preferable than the internal route under any environment), i.e.,

$$(R.o \in V_I) \wedge (R.\mathcal{D} \wedge d \neq 0) \rightarrow \text{Cond}(R.\mathcal{D} \wedge d).$$

Cond() is extracting the advertiser condition from a BDD predicate. E.g., $\text{Cond}(\neg p_1 \neg p_2) = \mathsf{T}$, $\text{Cond}(p_1 n_1^2) = n_1^2$.

6.2 Forwarding properties

Expresso checks forwarding properties based on the PECs.

TrafficHijackFree specifies that traffic among internal nodes of a network should only traverse the network's own routers. Checking this property can prevent internal traffic from being hijacked by external routers. Expresso checks this property by: for every internal node $u \in V_I$, and every PEC $pec \in PECs(u)$, if its final state is `EXIT`, its packet doesn't overlap with internal prefixes:

$$pec.fs = \texttt{EXIT} \rightarrow \neg(pec.pkt \wedge D_I).$$

6.3 More properties

Besides the above properties, Expresso can be used to check any properties that are related to the existence or absence of specific routes and/or packets. In the following, we discuss two examples.

One example property is `BlockToExternal`, defined by Bagpipe [27]. As will be shown in §7.3, the property specifies that routes with the `BTE` community should never be exported to any neighbors.

Another example property is `EgressPreference`. When there are multiple paths to the same Internet prefix, an internal router should choose the path according to a preference order. Checking this property can ensure the traffic exits the network at tje desired egress point. Specifically, for an internal node $u \in V_I$, an Internet prefix $d \in D_E$, and a preference order of neighbors $e_1 > e_2 > \cdots > e_n$ (prefer e_1 the most), the `EgressPreference` property specifies that for each $i \in [1, n-1]$ and $j \in [i+1, n]$:

$$cond_i \rightarrow \neg cond_j,$$

where $cond_i = \text{Cond}(\bigvee_{pec \in PECs(u,e_i)}(pec.pkt \wedge d))$ represents the advertiser condition for packets whose destination is d and whose egress point is e_i.

7 Evaluation

Implementation. We implemented Expresso with 28K lines of Java codes. Expresso uses the JDD library [26] for BDD operations, and Batfish [2] to parse configuration files.

Datasets. We evaluate Expresso on three sets of configurations (Table 1):

- Two WAN configuration snapshots of a large cloud service provider (CSP). They were collected at different times, with the new one collected two years after the old one. Both datasets consist of configurations for only a subset of WAN devices like peering routers.
- The Internet2 configuration snapshot publicly available at [7], which has been used in Bagpipe [27]. There are 10 routers and ~100K lines of Juniper configurations in total.

Table 1: Statistics of the datasets.

dataset		nodes	links	peers	prefixes	config lines
CSP WAN (old)	region1	$O(10)$	$O(10)$	$O(10)$	$O(200)$	$O(8k)$
	region2	$O(5)$	$O(10)$	$O(20)$	$O(400)$	$O(8k)$
	region3	$O(10)$	$O(30)$	$O(20)$	$O(600)$	$O(15k)$
	region4	$O(10)$	$O(30)$	$O(40)$	$O(2k)$	$O(22k)$
	full	$O(30)$	$O(100)$	$O(90)$	$O(3k)$	$O(54k)$
CSP WAN (new)		$O(130)$	$O(330)$	$O(220)$	$O(10k)$	$O(220k)$
Internet2		$O(10)$	$O(100)$	$O(300)$	$O(32k)$	$O(100k)$

Comparison. We compare Expresso with Minesweeper and Bagpipe. For Bagpipe, we use the results reported in its paper [27]. For Minesweeper, we use its source code available at [8]. Since Minesweeper does not support checking of routing properties like `RouteLeakFree`, and only partially models the longest prefix match (LPM) by selecting best routes firstly based on prefix length, we extend it (with 3K LOC) and call it Minesweeper* (details in Appendix C). We do not compare with other verifiers (e.g., Batfish, ShapeShifter, and SRE), because these tools consider concrete environments (every external neighbor advertises a specific set of routes under each environment), and have to enumerate all possible environments, which cannot finish in reasonable time (1 day). We enumerated 1000 environments (an extremely small portion of all environments) using Batfish, and it already took 2 hours.

Setup. The following evaluations are run on a Linux server with two 12-core Intel Xeon CPUs @ 2.3GHz and 256G RAM.

7.1 Property Violations Found by Expresso

We run Expresso on the two configuration snapshots from the CSP, and check three properties, i.e., `RouteLeakFree`, `RouteHijackFree`, and `TrafficHijackFree` (defined in §6).

Summary of found violations. Expresso found 63 (124) property violations in the old (new) snapshot, as shown in Table 2. After confirming with the operators, some of these violations are due to misconfiguration of route policies (the "fixed" and "will fix" columns), while some of them are due to other reasons (the "others" column), including (1) incompleteness of the snapshot, and (2) uninteresting cases. We haven't report the unconfirmed violations to the operators due to time limit, and we will keep working with the operators to confirm them. In the following, we show three examples of the violations found by Expresso.

Violation 1: Route Leak. Figure 5(a) shows one of the route leaks Expresso finds: when ISP_a announces a route to a /18 prefix, the provider will export the route to ISP_b, providing free transit from ISP_b to ISP_a. After inspecting the configurations, we found that the leaked prefix, if announced by ISP_a, will be permitted by the import policy on PR_1, propagated to the RR, reflected to PR_2, and

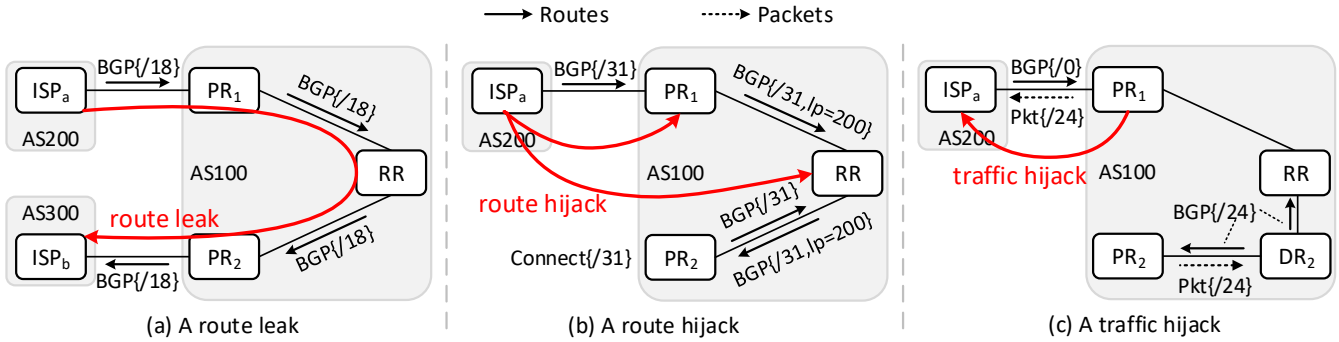


Figure 5: Examples of property violations found by Expresso in the WAN of a large cloud service provider.

Table 2: Summary of violations found by Expresso on the old snapshot and new snapshot.

		total	confirmed			unconfirmed
			fixed	will fix	others	
old	RouteLeak	3	1	0	1	1
	RouteHijack	53	1	0	6	46
	TrafficHijack	7	0	0	5	2
new	RouteLeak	36	1	2	1	32
	RouteHijack	70	0	1	9	60
	TrafficHijack	18	0	0	14	4

permitted by the export policy on PR_2 to ISP_b. After confirming with the operators, we find this route leak is actually caused by the the incompleteness of snapshot. The leaked prefix is actually an internal prefix announced by a router in one of their datacenters, and the route announced by the internal router has a higher local preference than external routes. However, it is still valuable to the operators, because it reveals a misconfiguration of the import policy on PR_1—it should deny all external routes for internal prefixes, but the routes are not filtered due to a missing entry. This makes route leaks possible on failures: when the datacenter router fails to announce the prefix due to interface or link failures, the route will be leaked.

Violation 2: Route Hijack. Figure 5(b) shows one of the route hijacks Expresso finds: if ISP_a announces a route to an internal /31 prefix of the provider, the data centers would choose to use ISP_a (instead of the CSP's WAN) to reach the /31 prefix. We inspect the configuration files and find the hijacked prefix belongs to an interface of a peering router PR_2, and the route for the prefix is redistributed into BGP with default local preference (i.e., 100). If a route for that prefix is advertised by some peering ISPs connecting to PR_1, it will not be filtered by PR_1, and PR_1 will set its local preference to 200. The route reflector RR will select the route received from the ISP as the best route, and reflect it to other routers. The operators acknowledged this was a misconfiguration and corrected it by adding the /31 prefix entry into PR_1's incoming deny list against ISP_a.

Violation 3: Traffic Hijack. Figure 5(c) shows one of the traffic hijacks Expresso found: PR_1 has no route to an internal /24 prefix hosted by DR_2. Since PR_1 has default routes with ISP_a as the next hop, when packets destined for the internal prefix reach PR_1, the packets will match the default route and be sent to ISP_a. After inspecting the configurations, we found the cause: after the internal route leaves DR_2 and reaches RR, the route is denied by the export policy of RR to PR_1. The operators indicated that such a policy was intentional—they want traffic towards DR_2 to enter the cloud at PR_2, but not PR_1. In addition, since PR_1 itself will not generate traffic for the prefix, this traffic hijack will never happen in reality. Such an uninteresting violation can be filtered by specifying where traffic can originate, which is left as one of our future work. Finally, this violation revealed the configuration was not following best practice, i.e., PR_1 should accept the internal route, but not export the route to ISP_a.

7.2 Performance on the CSP's WAN

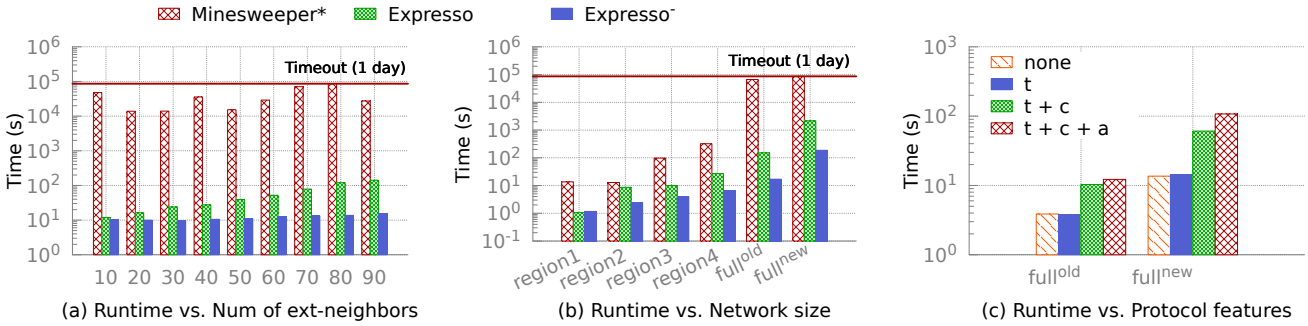
In the following, we evaluate the running time of Expresso, and compare the results with Minesweeper*, with the two CSP snapshots. For a fair comparison with Minesweeper*, which makes AS path length instead of AS path symbolic, we also include the results of Expresso$^-$, which uses concrete, instead of symbolic, AS paths for symbolic routes.

Running time vs. number of neighbors. Figure 6(a) shows the running time of Expresso and Minesweeper* to check `RouteLeakFree` on the old snapshot, with a different number of external neighbors. Generally, Expresso is 2-4 orders of magnitude faster than Minesweeper*. Here, the running time for Minesweeper* is higher than those reported in its paper, since our extension leads to more complex SMT encoding, which leads to a 10× slow down. [9]

Running time vs. network size. To evaluate how Expresso scales with network sizes, we use it to check `RouteLeakFree` property on the four individual regions and full snapshots, and compare with Minesweeper*. As shown in Figure 6(b), Expresso is more scalable with network sizes, and is always at least 1 order of magnitude faster than Minesweeper*.

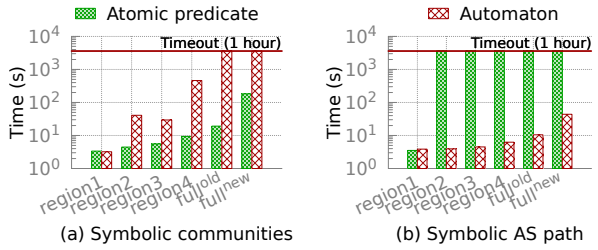
Running time vs. protocol features. Figure 6(c) shows the running time when modeling a different set of protocol features (AS path, community). We randomly choose 10 external neighbors

[9]We also run the original version of Minesweeper, which returned inconsistent answers, due to partial modeling of the longest-prefix-match.



(a) Runtime vs. Num of ext-neighbors (b) Runtime vs. Network size (c) Runtime vs. Protocol features

Figure 6: Running time of Minesweeper*, Expresso, and Expresso⁻ when there are (a) different numbers of neighbors and (b) different network sizes, and (c) the running time of Expresso for different protocol features, including traffic policy ('t'), symbolic community ('c'), and symbolic AS path ('a').



(a) Symbolic communities (b) Symbolic AS path

Figure 7: Runtime of Expresso using automaton or atomic predicate represented (a) symbolic communities, and (b) symbolic AS path.

Table 3: Runtime (in seconds) of SRC, SPF and property analysis.

	SRC	Routing Prop Analysis	SPF	Forwarding Prop Analysis
region1	1.028	0.025	0.552	0.006
region2	1.307	0.042	0.728	0.008
region3	1.690	0.034	0.428	0.008
region4	1.561	0.037	0.304	0.007
full(old)	2.770	0.067	0.734	0.007
full(new)	10.030	0.182	4.054	0.011

and check both routing property and forwarding property (i.e., `RouteLeakFree` and `TrafficHijackFree`). The result shows that modeling community incurs a high overhead, due to the complexity of computing atomic predicates for communities.

Automaton vs. atomic predicates. Figure 7 shows the running time of using automaton and atomic predicate to encode symbolic communities and symbolic AS path. It shows that for symbolic communities, using atomic predicate is more efficient, while for symbolic AS path, is the opposite.

Running time for each stage. Table 3 shows the running time of Expresso for the symbolic route computation (SRC), symbolic packet forwarding (SPF), and property analysis, on both the old and the new snapshot, with 10 randomly chosen external neighbors.

Table 4: Performance of Bagpipe, Minesweeper* and Expresso on the Internet2 to check BlockToExternal.

	Bagpipe	Minesweeper*	Expresso	Expresso ⁻
runtime (s)	28,594 (8h)	2,282	655	338
memory (GB)	-	45	12	12
violations	5	0	4	4

Memory usage. Figure 8 shows the corresponding memory usage of Minesweeper and Expresso for the experiments shown in Figure 6.

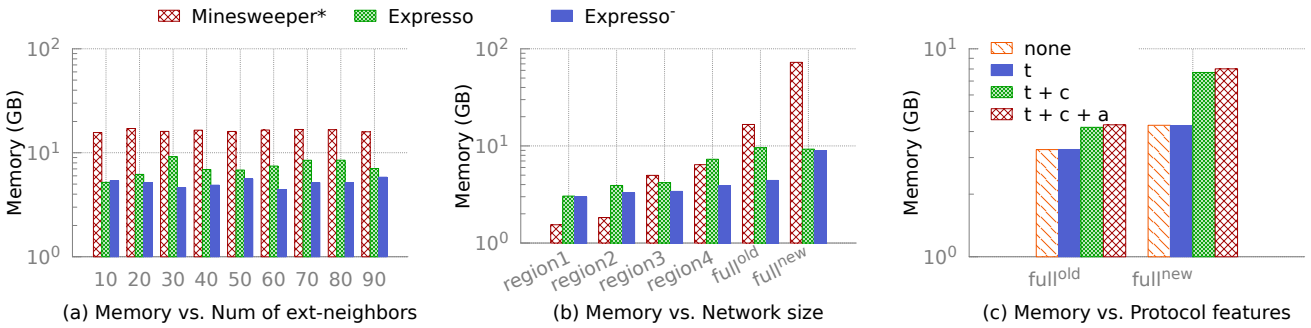
7.3 Performance on the Internet2

We use the public Internet2 dataset, and compare the performance of Expresso, Bagpipe [27], and Minesweeper [12]. Specifically, we use the three verifiers to check the property `BlockToExternal`, which was introduced by Bagpipe. This property ensures that routers of the Internet2 should not export any routes with a community named BTE to any of their external neighbors. It is checked for every external node $u \in V_E$, and every symbolic route $R \in \text{RIB}(u)$, if R doesn't have the BTE community: $\text{BTE} \notin R.C$.

As shown in Table 4, Bagpipe found 5 violations in total, while Expresso found 4 of them. A possible explanation is that Bagapipe reported there were 274 neighbors, while Expresso only recognized 266 neighbors. Minesweeper* did not find any violations. For running time, Bagpipe took 8 hours to finish, while Expresso only took less than 6 minutes (without considering symbolic communities or AS paths), In addition, Expresso also used less memory than Minesweeper*.

8 Limitations

Limited support for symbolic route attributes. Expresso uses concrete (default) values for some transitive BGP attributes like MED (Multi-Exit Discriminator). In addition, when comparing two symbolic routes, since a symbolic AS path contains multiple concrete AS paths, which may have different lengths, we use the shortest AS path length to represent the length of a symbolic AS path.



(a) Memory vs. Num of ext-neighbors (b) Memory vs. Network size (c) Memory vs. Protocol features

Figure 8: Memory usage of Minesweeper*, Expresso, and Expresso⁻ when there are (a) different numbers of neighbors and (b) different network sizes, and (c) the memory usage of Expresso for different property features, including traffic policy ('t'), symbolic community ('c'), and symbolic AS path ('a').

Thus, Expresso may miss property violations caused by external routes with non-default MED values or non-shortest AS path.

Limited support for route dependencies. Expresso assumes BGP depends on IGP, and lets IGP converge first. Therefore, it doesn't support arbitrary dependency, e.g., IGP depends on BGP by redistributing BGP routes. In addition, Expresso doesn't support route aggregation during control plane execution. To support it, we should use more variables for each external neighbor, as discussed in §3.1. Finally, Expresso doesn't support BGP conditional advertisement, which allows the router to control the advertisement of some routes bases on the existence or absence of some other prefixes in the routing table [5].

No support for arbitrary schedule. EPVP computes symbolic routes, which is essentially a batch of concrete routes. Therefore, it may result in a situation when EPVP cannot converge but real-world route computations converge (different routes computed with different schedules).

9 Conclusion

Expresso is a general and scalable network verifier that can comprehensively reason about external routes. Expresso first symbolically executes the network control plane to discover route equivalence classes (RECs), to scale to the colossal external route space. Expresso then symbolically executes the network data plane to discover packet equivalence classes (PECs), to scale to the large header space. On top of RECs and PECs, Expresso can check multiple kinds of routing and forwarding properties. Our future work includes overcoming the limitations of Expresso.

Acknowledgement. We thank our Shepherd Aurojit Panda, and all the anonymous SIGCOMM reviewers for their valuable comments and suggestions. This work is partially supported by the National Natural Science Foundation of China (No. 62272382). Peng Zhang is the corresponding author of this paper.

This work does not raise any ethical issues.

References

[1] [n. d.]. ARTEMIS. https://labs.ripe.net/author/vasileios_kotronis/artemis-neutralising-bgp-hijacking-within-a-minute/.
[2] [n. d.]. Batfish. https://github.com/batfish/batfish.
[3] [n. d.]. BGPalerter. https://github.com/nttgin/BGPalerter.
[4] [n. d.]. Cloudfare Radar. https://radar.cloudflare.com/.
[5] [n. d.]. Configure and Verify the BGP Conditional Advertisement Feature. https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/16137-cond-adv.html.
[6] [n. d.]. Google leaked prefixes and knocked Japan off the Internet. https://www.internetsociety.org/blog/2017/08/google-leaked-prefixes-knocked-japan-off-internet/.
[7] [n. d.]. Internet2 - Visible Backbone. https://vn.net.internet2.edu/Internet2/.
[8] [n. d.]. Minesweeper. https://github.com/batfish/batfish/releases/tag/2021-03-16-minesweeper.
[9] [n. d.]. RIPE NCC. https://www.ripe.net/.
[10] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast and General Network Verification. In *USENIX NSDI*.
[11] Henrik Reif Andersen. 1997. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen* (1997).
[12] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A general approach to network configuration verification. In *ACM SIGCOMM*.
[13] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2020. Abstract interpretation of distributed network control planes. In *ACM POPL*.
[14] Matthew L Daggitt, Alexander JT Gurney, and Timothy G Griffin. 2018. Asynchronous convergence of policy-rich distributed Bellman-Ford routing protocols. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 103–116.
[15] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient network reachability analysis using a succinct control plane representation. In *USENIX OSDI*.
[16] 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*. 43–56.
[17] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A general approach to network configuration analysis. In *USENIX NSDI*.
[18] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM*.
[19] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. 2020. NV: an intermediate language for verification of network control planes. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*.
[20] Timothy G Griffin, F Bruce Shepherd, and Gordon Wilfong. 2002. The stable paths problem and interdomain routing. *IEEE/ACM Transactions On Networking* 10, 2 (2002), 232–243.
[21] John F Lucas. 1990. *Introduction to Abstract Mathematics*. Rowman & Littlefield, 187.
[22] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, P Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *USENIX NSDI*.
[23] Divya Raghunathan, Ryan Beckett, Aarti Gupta, and David Walker. 2022. ACORN: Network Control Plane Abstraction using Route Nondeterminism. In *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2022*. 261.
[24] Joao L Sobrinho. 2005. An algebraic theory of dynamic network routing. *IEEE/ACM Transactions on Networking* 13, 5 (2005), 1160–1173.
[25] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2020. Probabilistic Verification of Network Configurations. In *ACM SIGCOMM*.

[26] Arash Vahidi. [n. d.]. JDD, a pure Java BDD and Z-BDD library. https://bitbucket.org/vahidi/jdd/.
[27] 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. In *ACM OOPSLA*.
[28] Hongkun Yang and Simon S Lam. 2013. Real-time verification of network properties using Atomic Predicates. In *IEEE ICNP*.
[29] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, et al. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *ACM SIGCOMM*.
[30] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. 2022. Symbolic Router Execution. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 336–349. https://doi.org/10.1145/3544216.3544264

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

A Simple Path Vector Protocol Algorithm

Algorithm 1 shows the algorithm for SPVP.

Algorithm 1: SPVP($G(V, E)$)

Input: $G(V, E)$: the topology of the network, where V is the vertex set, and $E : V \times V$ is the edge set.
Input: d: the considered prefix.
Output: $Best_u$: the converged routing table consisting of all best routes.

```
// initialize routes for nodes
foreach u ∈ V do
    Best_u ← {};
    if u announces d then
        Best_u ← Best_u ∪ {0};
// fixed-point computation
converged ← false;
while ¬converged do
    converged ← true;
    foreach u ∈ V do
        Recv_u ← {};
        foreach (u, v) ∈ E do
            Recv_u ← Recv_u ∪ {f_uv(σ) | σ ∈ Best_v)};
        Best_u ← ⊕(Best_u, Recv_u);
        if Best_u has changed then
            converged ← false;
```

B Algorithm for Transfer Function

Algorithm 2 shows the algorithm to compute the complete and non-overlapping unambiguous transfer function set for a transfer function.

The number of unambiguous transfer functions is decided by the number of unique processing flows in the transfer function. The matching predicates (i.e., $\alpha_i, i \in [0, n]$) are complete (Equation (6)) and non-overlapping (Equation (7)), that is, a concrete route r can match one and only one matching predicate:

$$\exists i \in [0, n] : \alpha_i \wedge r \tag{6}$$

$$\forall i, j \in [0, n], i \neq j : (\alpha_i \wedge r) \rightarrow \neg(\alpha_j \wedge r) \tag{7}$$

C Modification to Minesweeper

We made the following two modifications to Minesweeper, so that it can check the properties that we list in §2.2.

Algorithm 2: `transfer`(f)

Input: f: a transfer function.
Output: $\hat{f} = \{(\alpha_0, f_0), \cdots, (\alpha_n, f_n)\}$: a list of match-action pairs.

```
f̂ ← {};
// Predicate for unmatched routes. Initially True.
α ← True;
i ← 0;
foreach rule in Rules(f) do
    // Routes that can match this rule but none of the
       previous rules.
    α_i ← Match(rule, α);
    f_i ← Actions(rule);
    if node.mode ← DENY then
        // Add the deny action.
        f_i ← f_i(R) = ∅;
    if α^i ≠ False then
        // Update the predicate for unmatched routes.
        α ← α ∧ ¬α_i;
        f̂ ← f̂ ∪ {(α_i, f_i)};
    i ← i + 1;
// Deny unmatched routes by default.
f̂ ← f̂ ∪ {(α, f_i(R) = ∅)};
```

(1) Correcting longest prefix match. The open source version of Minesweeper does not encode the longest prefix match as a data plane packet forwarding principal. Instead, it encodes it as a control plane best route selection principal. Specifically, when Minesweeper selects per-protocol/overall best routes, it compares the prefix lengths of different routes. Thus, it may drop shorter prefixes because of low priority, which wrongly prevents the further propagation of shorter prefixes[10]. However, in real networks, the shorter prefixes will continue to propagate and may cause routing property violations. Therefore, we separate the encoding of a network in Minesweeper into multiple control planes (one for a single prefix length) and one data plane, let different control planes select their best routes independently and the data plane selects the route for forwarding from all control planes' routes according to longest-prefix-match.

(2) Checking routing properties. We extend Minesweeper to check routing questions (e.g., does the network satisfy no free transit). While Minesweeper checks forwarding properties by defining a global symbolic packet and constraining all symbolic routes according to its destination IP address, we define a global symbolic prefix and constrain all symbolic routes according to it, thus find a routing property violation (e.g. a route leak) of a prefix or verify there's no routing property violations.

D Correctness Proof for EPVP

D.1 Asynchronous schedule and network states

While a routing algebra defines the basic operations routers perform to modify messages, it does not say how such messages are processed on a given topology. Each device inspects the routing

[10] Essentially, this violates the principle that computations of different routes are independent.

messages from its neighbors after applying neighbor-specific transfer functions $f \in F$, and then merges the messages using merge function $\oplus$, until the network reaches a fixed point. This computation is complicated by the fact that messages are exchanged asynchronously and can be reordered.

Therefore, inspired by [13] and [14], we define an asynchronous execution schedule, which uses a linear notion of time captured by the set of time steps $\mathcal{T}$.

In the following, we use P ($|P| = 2^{33} - 1$) to represent the set of all prefixes, and $G(V, E)$ to represent a network topology, where V is the set of vertexes ($V = V_I \cup V_E$, where V_I and V_E are the set of internal and external vertexes respectively), and $E : V \times V$ is the set of edges.

Asynchronous Schedule. An asynchronous schedule is a tuple $T = (\tau, \omega)$ with respect to a network topology $G(V, E)$, where:

- $\tau : \mathcal{T} \to 2^V$ is an execution schedule that maps a time step to the set of vertices that process routes at that time step.
- $\omega : \mathcal{T} \to V \to V \to \mathcal{T}$ is a trace back function where $\omega(t, u, v)$ gives the time step where the information used at node u at time t was sent by node v. We require that $\omega(t, u, v) < t$, i.e., disseminating routing advertisements from one node to another takes a non-zero amount of time.

Network State. A network state is a $|P| \times |V|$ matrix S, where $S_{p,u}$ represents the route to prefix p on node u. An initial network state I is a special network state, with $I_{p,u}$ denotes the initial route to prefix p on node u. $I_{p,u}$ is typically 0 if node u initially announces prefix p, otherwise ∞.

Given an asynchronous schedule $T = (\tau, \omega)$, a network's state at any time t can be defined as:

$$S^0_{p,u} = I_{p,u}$$

$$S^t_{p,u} = \begin{cases} S^{t-1}_{p,u}, & \text{if } u \notin \tau(t) \\ I_{p,u} \oplus (\bigoplus_{(u,v)\in E} f_{uv}(S^{\omega(t,u,v)}_{p,v})) & \text{if } u \in \tau(t) \end{cases}$$

After zero time steps, the state at node u is given by the initial state I. To find the state after time step t for node u, it checks if u is currently scheduled to process route updates ($u \in \tau(t)$). If not, then the state at node u is the same as at time step $t-1$. Otherwise, node u will update its state by merging the last routes it has learned from each neighbor under schedule ω.

Network Environment. For a network with external neighbors, a network environment is a concrete state which indicates for each prefix and neighbor, whether this neighbor announces this prefix or not. Since V contains external nodes of the network, the initial state I uniquely determines a network environment. In the following, we denote the universe set of network environments as $\mathcal{E}$, with $|\mathcal{E}| = 2^{|P|\times|V_E|}$. The initial state that determines environment $e \in \mathcal{E}$ is denoted as I_e.

Combined Network State. Given an asynchronous schedule, a network's state at time t differs for different initial states (i.e., different environments), we define a combined network state $\mathbb{S}$ by defining for each prefix p, node u and environment e. That is,

$$\mathbb{I}_{p,u,e} = I_{p,u,e}$$

$$\mathbb{S}^t_{p,u,e} = S^t_{p,u,e} = \begin{cases} S^{t-1}_{p,u,e} & \text{if } u \notin \tau(t) \\ (I_{p,u,e} \oplus \bigoplus_{(u,v)\in E} f_{uv}(S^{\omega(t,u,v)}_{p,v,e})) & \text{if } u \in \tau(t) \end{cases}$$

Mapping relation. In the following, we define the mapping relation between concrete routes and symbolic routes.

Definition 1. *Given a symbolic route $R = (\mathcal{D}, attrs)$, its unfolding, denoted as $\overline{R}$, is the concrete route set it represents. That is, $\overline{R} = \{(d, e, attrs) | (d, e) \in \mathcal{D}\}$. Given a symbolic route set $\mathcal{R}$, its unfolding, denoted as $\overline{\mathcal{R}}$, is the union of the unfoldings of the symbolic routes in it. That is, $\overline{\mathcal{R}} = \bigcup_{R\in\mathcal{R}} \overline{R}$.*

Definition 2. *Given a set of concrete routes C, a set of symbolic routes $\mathcal{R}$ is said to be a partition of C, denoted as $\otimes(C) = \mathcal{R}$, if the following conditions hold [21]:*

- *Every symbolic route in $\mathcal{R}$ is a subset of C:*
$$\forall R \in \mathcal{R}, \bar{R} \subseteq C. \tag{8}$$
- *The union of symbolic routes in $\mathcal{R}$ is equal to C:*
$$\bigcup_{R\in\mathcal{R}} \bar{R} = C \tag{9}$$
- *If two concrete routes are included in the same symbolic route, they have the same attributes:*

$$\forall r_1, r_2 \in C, R \in \mathcal{R}, r_1 \in R \wedge r_2 \in R \Rightarrow r_1.attrs = r_2.attrs. \tag{10}$$

As an example, if $C = \{r_1, r_2\}$, then we have:

$$\otimes(\{r_1, r_2\}) = \begin{cases} \{R_1, R_2\}, & \text{if } r_1.attrs \neq r_2.attrs \\ \{R_3\}, & \text{if } r_1.attrs = r_2.attrs \end{cases} \tag{11}$$

, where

$$R_1 = (\{(r_1.d, r_1.e)\}, r_1.attrs)$$
$$R_2 = (\{(r_2.d, r_2.e)\}, r_2.attrs)$$
$$R_3 = (\{(r_1.d, r_1.e), (r_2.d, r_2.e)\}, r_1.attrs)$$

Since the number of unique attributes differs for different time t and node u, we use $\mathcal{A}^t_u$ to denote the set of unique attributes on node u at time t.

Symbolic Network State. Since EPVP is processing symbolic routes, at any time t, the symbolic network state computed by EPVP is:

$$\hat{\mathbb{I}}_u = \bigotimes(\bigcup_{p\in P, e\in\mathcal{E}} \mathbb{I}_{p,u,e}) \tag{12}$$

$$\hat{\mathbb{S}}^t_u = \hat{\mathbb{I}}_u \hat{\oplus} \hat{\bigoplus}(\bigcup_{(u,v)\in E} (\bigcup_{a'\in\mathcal{A}^{\omega(t,u,v)}_v} \hat{f_{uv}}(R^{\omega(t,u,v)}_{a',v})))$$
$$= \bigcup_{a\in\mathcal{A}^t_u} R^t_{u,a} \tag{13}$$

, where Equation (12) corresponds to line 1 in Algorithm 1, and Equation (13) corresponds to line 9 to line 12 in Algorithm 1.

Note that $\hat{\mathbb{S}}^t$ is defined by defining $\hat{\mathbb{S}}^t_u$ for each node u instead of for each prefix p and node u as for $\mathbb{S}^t$. Since prefixes are included in the destination-environment pair set of symbolic routes. To distinguish functions applied on concrete and symbolic routes, we

use $\hat{\oplus}$ and $\hat{f}$ to denote merge and transfer functions for symbolic routes.

D.2 Theorems and Proofs

To prove that the symbolic routes computed by EPVP contains **no more** or **no less** concrete routes, we have to prove that at every time t, the symbolic network state $\hat{\mathbb{S}}^t$ computed by Expresso is equivalent to the combined network state $\mathbb{S}^t$.

Before proving for the equivalence of $\mathbb{S}^t$ and $\hat{\mathbb{S}}^t$, we first lift the merge and transfer functions to a set of concrete or symbolic routes (i.e., Definition 3 - 6), and prove two lemmas that state the equivalence of concrete and symbolic merge (transfer) functions (i.e., Lemma 1, 2).

Definition 3. *Applying the merge function $\oplus$ on a set of concrete routes C is defined as merging the concrete routes in C pairwise. That is, $\oplus(C) = \bigcup_{r_i, r_j \in C, r_i \neq r_j} r_i \oplus r_j$. It represents the most preferred routes in C that are selected as best routes. More generally, merging a set of concrete route sets $\mathbb{C}$ is defined as $\oplus(\mathbb{C}) = \oplus(\bigcup_{C \in \mathbb{C}} C)$.*

Definition 4. *Applying a transfer function f on a set of concrete routes C is defined as applying it on the concrete routes in C separately. That is, $f(C) = \{f(r_i) | r_i \in C \wedge f(r_i) \neq \infty\}$. More generally, transferring a set of concrete route sets $\mathbb{C}$ is defined as $f(\mathbb{C}) = f(\bigcup_{C \in \mathbb{C}} C)$.*

Definition 5. *Applying the merge function $\hat{\oplus}$ on a set of symbolic routes $\mathcal{R}$ is defined as merging the symbolic routes in $\mathcal{R}$ pairwise. That is, $\hat{\oplus}(\mathcal{R}) = \bigcup_{R_i, R_j \in \mathcal{R}, R_i \neq R_j} R_i \hat{\oplus} R_j$. More generally, merging a set of symbolic route sets $\mathbb{R}$ is defined as $\hat{\oplus}(\mathbb{R}) = \hat{\oplus}(\bigcup_{\mathcal{R} \in \mathbb{R}} \mathcal{R})$.*

Definition 6. *Applying a transfer function $\hat{f}$ on a set of symbolic routes $\mathcal{R}$ is defined as applying it on the symbolic routes in $\mathcal{R}$ separately. That is, $\hat{f}(\mathcal{R}) = \bigcup_{R_i \in \mathcal{R}} \hat{f}(R_i)$. More generally, transferring a set of symbolic route sets $\mathbb{R}$ is defined as $\hat{f}(\mathbb{R}) = \hat{f}(\bigcup_{\mathcal{R} \in \mathbb{R}} \mathcal{R})$.*

Lemma 1. *The merge function for symbolic routes is sound and complete to the union operation, that is $\oplus(\overline{R_1} \cup \overline{R_2}) = \overline{R_1 \hat{\oplus} R_2}$.*

This lemma states that merging symbolic routes (i.e., $R_1 \hat{\oplus} R_2$) equals merging concrete routes ($\oplus(\overline{R_1} \cup \overline{R_2})$), every route that should be selected as the best route is indeed selected.

Proof. We prove Lemma 1 by cases. □

For every $r_1 \in \overline{R_1}$ and $r_2 \in \overline{R_2}$:

case 1 ($\rho(r_1.attrs) = \rho(r_2.attrs)$):
On one hand, r_1 and r_2 have the same preference, thus both can be selected as best routes, i.e.,

$$r_1 \in \oplus(\overline{R_1} \cup \overline{R_2}) \text{ and } r_2 \in \oplus(\overline{R_1} \cup \overline{R_2}).$$

On the other hand, since $R_1.attrs = r_1.attrs$ and $R_2.attrs = r_2.attrs$, we have $R_1 \hat{\oplus} R_2 = \{R_1, R_2\}$, which means

$$r_1 \in \overline{R_1} \wedge \overline{R_1} \subseteq \overline{R_1 \hat{\oplus} R_2} \Rightarrow r_1 \in \overline{R_1 \hat{\oplus} R_2}, \text{ and}$$
$$r_2 \in \overline{R_2} \wedge \overline{R_2} \subseteq \overline{R_1 \hat{\oplus} R_2} \Rightarrow r_2 \in \overline{R_1 \hat{\oplus} R_2}.$$

Therefore, in this case, r_1 and r_2 are included in both $\oplus(\overline{R_1} \cup \overline{R_2})$ and $\overline{R_1 \hat{\oplus} R_2}$.

case 2 ($\rho(r_1.attrs) > \rho(r_2.attrs)$):
On one hand, r_1 is preferred than r_2, thus only r_1 can be selected as best route, i.e.,

$$r_1 \in \oplus(\overline{R_1} \cup \overline{R_2}) \text{ and } r_2 \notin \oplus(\overline{R_1} \cup \overline{R_2}).$$

One the other hand, since $R_1.attrs = r_1.attrs$ and $R_2.attrs = r_2.attrs$, we have $R_1 \hat{\oplus} R_2 = \{R_1, (R_2.\mathcal{D} \wedge \neg R_1.\mathcal{D}, R_2.attrs)\}$, which means

$$r_1 \in \overline{R_1} \wedge \overline{R_1} \subseteq \overline{R_1 \hat{\oplus} R_2} \Rightarrow r_1 \in \overline{R_1 \hat{\oplus} R_2}, \text{ and}$$
$$r_2 \notin \overline{R_1} \wedge r_2 \notin \overline{(R_2.\mathcal{D} \wedge \neg R_1.\mathcal{D}, a)} \Rightarrow r_2 \notin \overline{R_1 \hat{\oplus} R_2}.$$

Therefore, in this case, r_1 is included in both $\oplus(\overline{R_1} \cup \overline{R_2})$ and $\overline{R_1 \hat{\oplus} R_2}$ while r_2 isn't.

case 3 ($\rho(r_1.attrs) < \rho(r_2.attrs)$): Similar to case 2.

conclusion: According to the above cases, we have

$$\forall r_1 \in \overline{R_1}, r_1 \in \oplus(\overline{R_1} \cup \overline{R_2}) \Leftrightarrow r_1 \in \overline{R_1 \hat{\oplus} R_2}, \text{ and}$$
$$\forall r_2 \in \overline{R_2}, r_2 \in \oplus(\overline{R_1} \cup \overline{R_2}) \Leftrightarrow r_2 \in \overline{R_1 \hat{\oplus} R_2},$$

which means $\oplus(\overline{R_1} \cup \overline{R_2}) = \overline{R_1 \hat{\oplus} R_2}$.

Lemma 2. *The transfer functions for symbolic routes are sound and complete to the union operation, that is, $f(\overline{R_1} \cup \overline{R_2}) = \overline{\hat{f}(R_1) \cup \hat{f}(R_2)}$.*

This lemma states that transferring symbolic routes (i.e., $\overline{\hat{f}(R_1) \cup \hat{f}(R_2)}$) equals transferring concrete routes (i.e., $f(\overline{R_1} \cup \overline{R_2})$, every route that should be permitted (denied) is indeed permitted (denied) in Expresso.

Proof. We prove Lemma 2 by cases. □

By definition (Equation (3)), we have:

$$\hat{f} = \{(\alpha_1, f_1), \cdots, (\alpha_n, f_n)\}$$

For every $r \in \overline{R_1} \cup \overline{R_2}$:

case 1 (r is permitted by f):
On one hand, we have:

$$f(r) \neq \infty \Rightarrow f(r) \in f(\overline{R_1} \cup \overline{R_2}).$$

On the other hand, suppose $r \in R_1$, we have:

$$\exists i \in [1, n] : r \in \alpha_i \wedge R_1 \text{ and } f_i(\alpha_i \wedge R_1) \neq \emptyset,$$

which means:

$$f(r) \in \overline{\hat{f}(R_1)} \Rightarrow f(r) \in \overline{\hat{f}(R_1) \cup \hat{f}(R_2)}.$$

Therefore, $f(r)$ is included in both $f(\overline{R_1} \cup \overline{R_2})$ and $\overline{\hat{f}(R_1) \cup \hat{f}(R_2)}$.

case 2 (r is denied by f):
On one hand, we have:

$$f(r) = \infty \Rightarrow f(r) \notin f(\overline{R_1} \cup \overline{R_2}).$$

On the other hand, suppose $r \in R_1$ we have:

$$\exists i \in [1, n] : r \in \alpha_i \wedge R_1 \text{ and } f_i(\alpha_i \wedge R_1) = \emptyset,$$

which means:

$$f(r) \notin \overline{\hat{f}(R_1)}.$$

Similarly, if $r \in R_2$, we have:

$$f(r) \notin \overline{\hat{f}(R_2)}.$$

Thus, we have:

$$f(r) \notin \overline{\hat{f}(R_1)} \wedge f(r) \notin \overline{\hat{f}(R_2)} \Rightarrow f(r) \notin \overline{\hat{f}(R_1) \cup \hat{f}(R_1)}.$$

Therefore, $f(r)$ is not in both $f(\overline{R_1} \cup \overline{R_2})$ and $\overline{\hat{f}(R_1) \cup \hat{f}(R_2)}$.

conclusion: According the above cases, we have

$$\forall r \in \overline{R_1} \cup \overline{R_2}, f(r) \in f(\overline{R_1} \cup \overline{R_2}) \Leftrightarrow f(r) \in \overline{\hat{f}(R_1) \cup \hat{f}(R_2)},$$

which means $f(\overline{R_1} \cup \overline{R_2}) = \overline{\hat{f}(R_1) \cup \hat{f}(R_2)}$.

Theorem 3. *Given an asynchronous schedule $T = (\tau, \omega)$, at any time t and node u, the Expresso network state $\hat{\mathbb{S}}_u^t$ computed by merging and transferring symbolic routes is equivalent to the combined network state $\mathbb{S}_u^t$ computed by merging and transferring concrete routes. Since $\mathbb{S}_u^t$ is a set of concrete routes and $\hat{\mathbb{S}}_u^t$ is a set of symbolic routes, the equivalence is formally represented as $\overline{\hat{\mathbb{S}}_u^t} = \mathbb{S}_u^t$.*

Proof. We prove for Theorem 3 by strong induction on the time t. □

case ($t = 0$): By definition $\hat{\mathbb{I}}_u = \bigotimes(\bigcup_{p\in P, e\in\mathcal{E}} \mathbb{I}_{p,u,e}) = \bigotimes(\mathbb{I}_u)$. Since $\otimes$ satisfies the Equation (9), we can conclude that $\overline{\hat{\mathbb{I}}_u} = \mathbb{I}_u$.

case t: To show $\overline{\hat{\mathbb{S}}_u^t} = \mathbb{S}_u^t$, there are two cases to consider.
(1) The first is that $u \notin \tau(t)$. In this case: we have

$$\mathbb{S}_u^t = \mathbb{S}_u^{t-1}, \text{ and } \hat{\mathbb{S}}_u^t = \hat{\mathbb{S}}_u^{t-1}$$

From the inductive hypothesis, we know

$$\mathbb{S}_u^{t-1} = \overline{\hat{\mathbb{S}}_u^{t-1}},$$

therefore, we have

$$\mathbb{S}_u^t = \overline{\hat{\mathbb{S}}_u^t}.$$

(2) The second case is where $u \in \tau(t)$. In this case, we have

$$\mathbb{S}_u^t = \bigcup_{e\in\mathcal{E}, p\in P} (I_{p,u,e} \oplus \bigoplus_{(u,v)\in E} f_{uv}(S_{p,v,e}^{\omega(t,u,v)})), \text{and}$$

$$\hat{\mathbb{S}}_u^t = \hat{\mathbb{I}}_u \hat{\oplus} \hat{\bigoplus}(\bigcup_{(u,v)\in E} (\bigcup_{a'\in\mathcal{A}_v^{\omega(t,u,v)}} \hat{f_{uv}}(R_{v,a'}^{\omega(t,u,v)})))$$

According to Definition 3 and Definition 4, we have:

$$\begin{aligned}\mathbb{S}_u^t &= \bigcup_{e\in\mathcal{E}, p\in P} I_{p,u,e} \oplus \bigoplus_{(u,v)\in E} f_{uv}(\bigcup_{e\in\mathcal{E}, p\in P} S_{p,v,e}^{\omega(t,u,v)}) \\ &= \mathbb{I}_u \oplus \bigoplus_{(u,v)\in E} f_{uv}(\mathbb{S}_v^{\omega(t,u,v)}). \end{aligned} \tag{14}$$

Since $\omega(t,u,v) < t$, by hypothesis, we have:

$$\mathbb{S}_v^{\omega(t,u,v)} = \overline{\hat{\mathbb{S}}_v^{\omega(t,u,v)}}.$$

By definition of Expresso network state and unfolding, we have:

$$\begin{aligned}\overline{\hat{\mathbb{S}}_v^{\omega(t,u,v)}} &= \overline{\bigcup_{a'\in\mathcal{A}_v^{\omega(t,u,v)}} R_{v,a'}^{\omega(t,u,v)}} \\ &= \bigcup_{a'\in\mathcal{A}_v^{\omega(t,u,v)}} \overline{R_{v,a'}^{\omega(t,u,v)}}.\end{aligned}$$

Therefore, Equation (14) is transformed into:

$$\mathbb{S}_u^t = \mathbb{I}_u \oplus \bigoplus_{(u,v)\in E} f_{uv}(\bigcup_{a'\in\mathcal{A}_v^{\omega(t,u,v)}} \overline{R_{v,a'}^{\omega(t,u,v)}}).$$

According to Lemma 2, the above equation is transformed into:

$$\mathbb{S}_u^t = \mathbb{I}_u \oplus \bigoplus_{(u,v)\in E} \overline{\bigcup_{a'\in\mathcal{A}_v^{\omega(t,u,v)}} \hat{f}_{uv}(R_{v,a'}^{\omega(t,u,v)})}.$$

According to Lemma 1, the above equation is transformed into:

$$\mathbb{S}_u^t = \mathbb{I}_u \oplus \overline{\hat{\bigoplus_{(u,v)\in E}} (\bigcup_{a'\in\mathcal{A}_v^{\omega(t,u,v)}} \hat{f}_{uv}(R_{v,a'}^{\omega(t,u,v)}))}.$$

Since for case $t = 0$, we have $\mathbb{I}_u = \overline{\hat{\mathbb{I}}_u}$, the above equation is transformed into:

$$\mathbb{S}_u^t = \overline{\hat{\mathbb{I}}_u} \oplus \overline{\hat{\bigoplus_{(u,v)\in E}} (\bigcup_{a'\in\mathcal{A}_v^{\omega(t,u,v)}} \hat{f}_{uv}(R_{v,a'}^{\omega(t,u,v)}))}.$$

Again, according to Lemma 1, the above equation is transformed into:

$$\mathbb{S}_u^t = \overline{\hat{\mathbb{I}}_u \hat{\oplus} \hat{\bigoplus_{(u,v)\in E}} (\bigcup_{a'\in\mathcal{A}_v^{\omega(t,u,v)}} \hat{f}_{uv}(R_{v,a'}^{\omega(t,u,v)}))}.$$

According to Definition 5, the above equation is transformed into:

$$\mathbb{S}_u^t = \overline{\hat{\mathbb{I}}_u \hat{\oplus} \hat{\bigoplus}(\bigcup_{(u,v)\in E} (\bigcup_{a'\in\mathcal{A}_v^{\omega(t,u,v)}} \hat{f}_{uv}(R_{v,a'}^{\omega(t,u,v)})))}.$$

We can observe that the right hand side of the above equation is $\hat{\mathbb{S}}_u^t$, which means:

$$\mathbb{S}_u^t = \overline{\hat{\mathbb{S}}_u^t}.$$