Efficient Network Reachability Analysis using a Succinct Control Plane Representation

Seyed K. Fayaz Tushar Sharma Ari Fogel*
Ratul Mahajan† Todd Millstein‡ Vyas Sekar George Varghese‡

CMU **Intentionet* *†Microsoft Research* ‡UCLA

Abstract— To guarantee network availability and security, operators must ensure that their reachability policies (e.g., A can or cannot talk to B) are correctly implemented. This is a difficult task due to the complexity of network configuration and the constant churn in a network's environment, e.g., new route announcements arrive and links fail. Current network reachability analysis techniques are limited as they can only reason about the current "incarnation" of the network, cannot analyze all configuration features, or are too slow to enable exploration of many environments. We build ERA, a tool for efficient reasoning about network reachability. Instead of reasoning about individual incarnations of the network, ERA directly reasons about the network "control plane" that generates these incarnations. We address key expressiveness and scalability challenges by building *(i)* a succinct model for the network control plane (i.e., various routing protocols and their interactions), and *(ii)* a repertoire of techniques for scalable (taking a few seconds for a network with $>$ 1000 routers) exploration of this model. We have used ERA to successfully find both known and new violations of a range of common intended polices.

1 Introduction

Network operators need to ensure the correct behavior of their networks. Violations of intended reachability policies (e.g., "Can A talk to B?") can compromise availability, security, and performance of the network. This risk has inspired the field of network verification, which aims to enable operators to systematically reason about their networks [39].

Reasoning about a network is hard, as a real network is in a perpetual churn: route advertisements arrive, links fail, and routers need to be taken offline for maintenance. Nonetheless, an operator needs assurances on the network behaviors because a policy violation may be latent and occur only in a certain future *incarnation* (e.g., a specific route advertisement from a peering network may cause disconnection between A and B [6, 11]). Unfortunately, today operators do not have proper tools for efficient reasoning about the network in different environments.

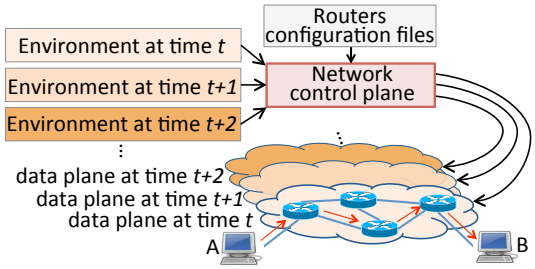


Figure 1: Reachability behavior of a network (e.g., A can talk to B) is determined by its data plane, which, in turn, is the current incarnation of the control plane.

To highlight this challenge, it is useful to consider prior work on network verification. A network is composed of a control plane, which configures the behavior of the data plane, which in turn, is in charge of forwarding actual packets (see Figure 1). The control plane, therefore, can be thought of as a program that takes configuration files and the current network environment (i.e., route advertisements) and generates a data plane. The data plane is conceptually a program that takes a packet and its location (i.e., a router port) as input and outputs a packet at a different location. As a result, if we rest our analysis on the data plane (e.g., Veriflow [29], HSA [28], NOD [36]) and verify its behavior over its inputs (i.e., packets), we are inherently able to reason about only the current incarnation of the control plane (i.e., the current data plane), and cannot say anything about the network behavior under a different environment.

While there is prior work on bug-finding and verification for the control plane, it suffers from critical limitations. Some tools focus on a single routing protocol (e.g., BGP for Bagpipe [41] and rcc [18]) or a limited set of routing protocol features (e.g., ARC [21]). They can thus not capture the behavior of the entire control plane that often uses multiple routing protocols and sophisticated features [22,31,38]. On the other hand, Batfish [19] analyzes the entire control plane in the context of a given environ-

ment, but it does so by simulating the behavior of individual routing protocols to compute the resulting data plane. This simulation is expensive (see §9.2), which makes it prohibitive to iteratively use Batfish to analyze the impact of many environments.

What is critically missing today is the ability to efficiently find network reachability bugs across multiple possible environments. (§3 motivates this need using real-world examples.) Doing so requires reasoning about network reachability directly at the control plane level, without explicitly computing the data plane that manifests in each environment. Such reasoning is challenging due to the complexity of the control plane, which involves various routing protocols (e.g., BGP, OSPF, RIP) each with its own intricacies (e.g., selecting best route to a destination prefix is different for BGP and OSPF) and cross-protocol interactions (e.g., route redistribution [32]).

We address these challenges in a tool called ERA (efficient reachability analysis) by employing several synergistic ideas. First, we design a unified control plane model that succinctly captures the key behaviors of various routing protocols. In this model, a router is viewed as a function that accepts a route announcement as input and produces a set of route announcements for its neighbors. Second, we use binary decision diagrams (BDDs) [30] to compactly represent the route announcements that constitute a user-specified environment. Third, we shrink the BDD representation of route announcements by identifying equivalence classes of announcements that are treated identically by the given network [42]. Each equivalence class is given an integer index, and the reachability analysis is transformed to arithmetic operations directly on sets of these indices. Consequently, we take advantage of vectorized instruction sets on commodity CPUs for fast computation of these set operations (§6).

ERA can be used to identify bugs in reachability policies of the form "A can talk to B" as well as a wide range of common policies that are expressible in terms of reachability relationships, such as valley-free routing and blackhole-freeness (§7). Our implementation of ERA is available as an open source and extensible toolkit to which new kinds of analysis can be added (§8).

We evaluate the utility of ERA in a range of real and synthetic scenarios (§9.1). Across all scenarios, it successfully finds both new and known reachability violations, which were otherwise hard to find using the state of the art techniques. We also evaluate the scalability of ERA and find that it can handle a network with over 1,600 routers in 6 seconds. Our evaluations show that our control plane modeling and exploration techniques yield significant speedup.

2 Related Work

There are several strands of related prior work.

Data plane analysis: Verifying the reachability behavior of the data plane has been widely studied (e.g., Anteater [37], Veriflow [29], HSA [28], NOD [36]). The result from such verification, however, is valid only for the specific data plane being analyzed. There has also been extensive work on testing the data plane (e.g., ATPG [43], Pingmesh [26]). Data plane verification and testing is fundamentally limited, as a network is in a constant churn, which manifests itself as different data planes. For example, a single route advertisement can dramatically change the network behavior (e.g., see [11]).

Control plane analysis: Moving from the data plane to the control plane potentially enables more powerful analysis, as the former is generated by the latter. However, prior work is limited due to supporting only a single routing protocol (e.g., BGP in Bagpipe [41] and rcc [18]) or a limited set of routing protocol features (e.g., ARC [21]). Batfish [19] can reason about the entire control plane but its analysis is expensive because it simulates the individual steps of each routing protocol. In contrast, ERA enables fast exploration using a succinct encoding of control plane behavior.

Clean-slate control plane design: Metarouting [24], glue logic [33], and Propane [16] aim to build a correct-by-design control plane. While worthwhile in the long term, these efforts cannot reason about existing networks.

To summarize, what is critically missing today is the ability to efficiently explore the control plane involving various routing protocols. We illustrate this need below.

3 Motivation: Reachability Bugs

We motivate reasoning about multiple network incarnations using real reachability bugs encountered in a large cloud provider's network. These bugs were latent and manifested only under certain environments.

Maintenance-triggered: Some bugs stem from unexpected interactions of different routing protocols and configuration directives. In this example (Figure 2), the interactions are between static routing and BGP. For redundancy, the operator's goal was to have two paths between the DCN (datacenter network) and the WAN (wide area network), one through R_1 and the other through R_2. One day, the operator decided to temporarily bring down R_2 for maintenance, which she thought was safe because of the assumed redundancy. However, as soon as R_2 was brought down, the entire DCN was disconnected from the WAN (and the rest of the Internet).

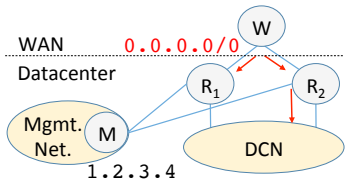


Figure 2: A bug triggered by maintenance.

Manual investigation revealed that R_1 contained a static default route `ip route 0.0.0.0/0 1.2.3.4` (here `1.2.3.4` is the next-hop of the static route, which is the address of the management network). Static routes to a prefix supersede dynamic routes [5, 8]. Thus R_1 preferred the static route over the default BGP route advertised by the WAN (shown in red). Since static routes are typically not propagated to neighbors, R_1 did not advertise the default route to the DCN. Thus, the DCN was entirely dependent on R_2 for external connectivity.

The bug in R_1's configuration was that the operator had forgotten to type keywords to indicate that the static route belonged to the management network, not data network. (These keywords were present in R_2's configuration.) The bug was latent as long as R_2 was up, but was triggered when R_2 was brought down.

Announcement-triggered: In Figure 3, DC_A had several services hosted inside the subprefixes of 10.10.0.0/16. Instead of announcing the individual subprefixes, R_1 was announcing this aggregate prefix. DC_B could reach the services inside DC_A through the WAN. As soon as a new service with prefix 10.10.1.160/28 was launched inside DC_A, all other services inside the /16 prefix became unreachable from DC_B.

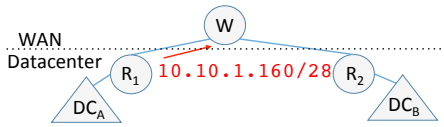


Figure 3: A bug triggered by a BGP announcement.

Investigation revealed two latent configuration bugs that combined to create this outage: *(1)* R_1 was not configured to filter 10.10.1.160/28 in its announcements to the WAN; and *(2)* R_2 was configured with an aggregate route to 10.10.0.0/16 with DC_B as the next hop. The result of the first bug was that the /28 announcement reached R_2 through the WAN. Then, as a result of the second bug, the /16 aggregate route was activated at R_2. This aggregate route, as a local route to router R_2, took precedence over the /16 being announced through the WAN. When the aggregate route was activated, R_2 started dropping all traffic to the /16 except for traffic to the /28. These drops are due to the *sinkhole* semantics of route aggregation—

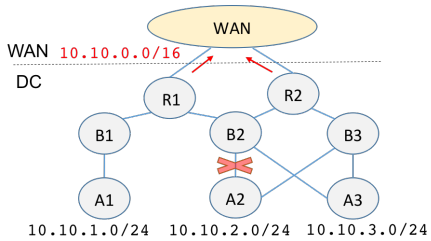


Figure 4: A bug triggered by link failure.

the aggregating router drops packets for subprefixes for which it does not have an active route to prevent routing loops [34].[1] Proper connectivity existed prior to the /28 announcement because the /16 announcement from the WAN did not activate the aggregate route at R_2.

Failure-triggered: In Figure 4, R_1 and R_2 were configured to announce prefix 10.10.0.0/16 that aggregated the subprefixes announced by leaf routers (A_1, A_2, A_3). After link A_2—B_2 failed, WAN traffic destined to A_2's prefix (10.10.2.0/24) started getting blackholed (i.e., dropped) at R_1 even though A_2 had connectivity via B_3 and R_2.

This blackhole was created because R_1 continued to make the aggregate announcement after the failure of link A_2–B_2, as it was still hearing announcements for the other two subprefixes corresponding to A_1 and A_3 (aggregate routes are announced as long as there is at least one subprefix present). As a result, the WAN sent (some) traffic for 10.10.2.0/24 toward R_1. But R_1 dropped those packets per the *sinkhole* semantics (see above).

4 ERA Overview

In this section, we present our approach and discuss the challenges in realizing it. Our target is a (datacenter, enterprise, or ISP) network of a realistic size (e.g., a few to hundreds of routers). As shown in Figure 5, our user is a network operator responsible for configuring routers. The operator has a set of intended reachability policies of the form "Router port A can talk to router port B" (as we will discuss in §7, several other practical policies are derivatives of "A can talk to B"). ERA allows operators to input their assumptions on what the network's environment will send (e.g., based on relationship with peers/providers). It then analyzes the network's behavior under these assumptions and checks whether the behavior satisfies the intended reachability policies. This process can then be iterated with other environmental assumptions, in order to cover a range of possible environments.

[1] For instance, if W announced the default route to R2, R2 would forward traffic for 10.10.2.2 to W, which may then forward them to R2 (because R2 announces the aggregate /16 to W), and so on.

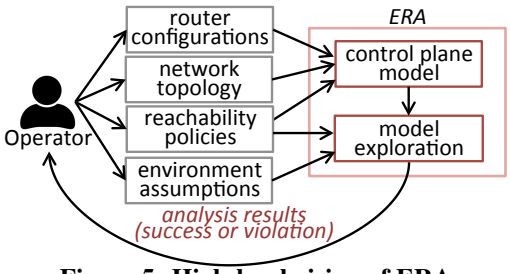


Figure 5: High-level vision of ERA.

4.1 Our Approach

Here we give the intuition behind our approach to control plane analysis.

Relationship between data and control planes: The data plane takes as input a packet on a router port and moves the (possibly modified) packet to another port (on the same or a neighboring router). Thus, we can think of the data plane as a function of the form $DP : (pkt, port) \rightarrow (pkt, port)$. The data plane itself is generated by the control plane function given routers' configuration files, the network topology (i.e., which router ports are inter-connected), and the current environment (which captures the route advertisements sent to the network by the "outside world") of the network: $CP : (env, Topo, Configs) \rightarrow DP(.)$.

Reachability policies via control plane analysis: Since packets are forwarded by the data plane, it is natural to think of an intended reachability policy $\phi_{A \rightarrow B}$ as a predicate that indicates whether a given packet should be able to reach from router port A to router port B. We say data plane DP is policy-compliant if $\phi_{A \rightarrow B}(pkt, DP)$ evaluates to $true$ for all A-to-B packets.

A seemingly natural approach for finding latent bugs is to produce the data plane associated with a given environment and then check reachability on that data plane [19]. However, this approach makes it prohibitively expensive to iteratively check multiple environments (§9.2). This is because for each possible environment (of which there are many), to compute the resulting data plane, we need to account for all low-level message passings and nuances of routing protocols. Instead, we want to be able to reason about the network directly at the level of the control plane and without explicitly computing the data plane.

To this end, our insight is as follows. Rather than producing the data plane that results from a given environment, we can analyze the control plane under that environment to determine i) the routes that each router in the network learns via its neighbors (e.g., a BGP advertisement) or its configuration file (e.g., static routes); and ii) the best route when multiple routes to the same prefix are learned. We can then use this information to directly check reachability.

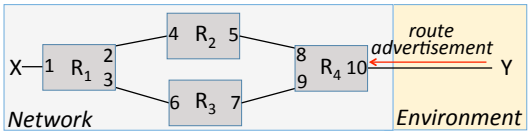


Figure 6: X-to-Y reachability depends on routers configurations and the environment.

An illustrative example: To visualize what it means to reason about reachability using control plane analysis, consider the example shown in Figure 6. Here we want to see what traffic reaches from port X to port Y so that we can check whether it is policy-compliant. From the figure we can see that to find the above traffic, we can try to find the routes that traverse the opposite direction on each of the two paths. Let $T^{i \rightarrow j}_{Router}(route)$ show the output of the configured router $Router$ on its port j given the input $route$ on its port i. (Intuitively, $route$ can be thought of as an abstraction for a route advertisement. The following section will elaborate on this abstraction.) If we knew $T(.)$, the answer would be: $T^{2 \rightarrow 1}_{R_1}(T^{5 \rightarrow 4}_{R_2}(T^{10 \rightarrow 8}_{R_4}(env))) \cup T^{3 \rightarrow 1}_{R_1}(T^{7 \rightarrow 6}_{R_3}(T^{10 \rightarrow 9}_{R_4}(env)))$. The argument env here represents the assumptions that the user makes about the environment.

4.2 Challenges

Control plane-based reachability analysis requires us to address two key challenges:

- **An expressive and tractable control plane model:** To be expressive, this model needs to capture key behaviors of diverse protocols (e.g., BGP, OSPF route advertisements). A naive model (e.g., capturing protocol-specific behaviors verbatim), while expressive, is impractical because it will be too complex to explore. On the other extreme, a very high-level model (e.g., ignoring protocol-specific behaviors altogether) may be tractable to explore, but not expressive (e.g., BGP and OSPF have different ways of preferring routes).
- **Scalable control plane exploration:** Once we have a control plane model, we need the ability to efficiently explore the model with respect to the environment, in order to identify violations of intended reachability policies.

We tackle these challenges in §5 and §6, respectively.

4.3 Scope and Limitations

ERA's analysis requires the user to provide assumptions on the environment (or defaults to assuming that the environment makes all possible route announcements). If these assumptions are incorrect or overly permissive, then ERA can produce false positives, identifying purported errors that in fact will never arise in practice; e.g., a rep-

utable ISP is not likely to hijack its peer's traffic. ERA is designed to have no other source of false positives (i.e., its control plane model is accurate). Though we have not formally proven this yet, empirically speaking, all the bugs that ERA has identified were real bugs.

ERA also has several sources of false negatives. First, ERA will only find bugs under environments specified as inputs and cannot guarantee the absence of bugs under all environments (unless exhaustively iterated on all possible environments). Second, certain classes of errors cannot be found by ERA by design. Specifically, ERA assumes that routing will converge and only analyzes this convergent state, which is key to efficient exploration of the control plane. Therefore convergence errors as well as reachability errors in transient states of the network will not be found (e.g., [23, 25]).

Finally, while ERA supports most of the common configuration directives, our current implementation does not support certain directives such as regular expressions in routing filters. Keeping up with configuration directives is a software engineering challenge due to their large and growing universe. Such limitations, however, are not fundamental to the design of ERA (unlike ARC [21], where the design itself cannot handle certain routing features).

As we will see in §9, ERA can find a large class of real-world bugs despite these limitations.

5 Modeling the Control Plane

We now describe our model for the network control plane. It i) captures all routing protocols using a common abstraction; ii) is expressive with respect to routing behaviors of individual protocols; and iii) lends itself to scalable exploration. At a high level, we identify key behaviors of the control plane (e.g., route selection, route aggregation) and compactly encode them using binary decision diagrams (BDDs) [30].

Since the network control plane is a composition of the control planes of individual routers, we break down the problem of modeling the network control plane into modeling *(i)* the I/O unit of a router's control plane (§5.1), and *(ii)* the processing logic of a router's control plane (§5.2).

5.1 Route as the Model of Control Plane I/O

A naive way of modeling the I/O unit of the control plane of a router is to use the actual specification of route advertisements of different routing protocols, including their low-level details (e.g., keep-alive messages, sequence numbers [3, 9]). While expressive, such an I/O unit makes the control plane model too cumbersome. Conversely, if we completely ignore differences across protocols to simplify our I/O unit model, such a model may not be expressive; e.g., it cannot capture the fact that if a router learns

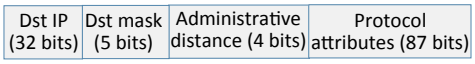


Figure 7: *route* as the model of control plane I/O.

two routes to the same destination prefix from two different routing protocols, the one offered by the protocol that has a smaller administrative distance (AD) will be selected [5, 8]. (We will see an example bug scenario due to this effect in §9.1.2, Figure 15b.)

To strike a balance between expressiveness and tractability, we introduce the notion of an abstract *route* as a succinct yet expressive I/O unit for the control plane model. Conceptually, a route mimics a route advertisement. It is a succinct bit-vector conveying key information in route advertisements that affects routing decisions of a router (see Figure 7). While not fundamental to our design, we have chosen a 128-bit vector to encode a route to enable fast CPU operations as we will discuss in §6.2. To accommodate diverse routing protocols, a route unifies key attributes of various protocols that affect a router's behaviors (i.e., administrative distance and protocol-specific route attributes).[2] To improve scalability, a route abstracts away the low-level nuances of actual protocols (e.g., seq. numbers, acknowledgements).

The fields of our route abstraction are:

- *Destination IP and mask*: Together, they represent the destination prefix that the route advertises. To make a route compact, we store the mask in 5 bits (instead of its naive storage in 32 bits). For completeness, Appendix A shows the details of how we do this.
- *Administrative distance (AD)*: This is a numerical representation of the routing protocol (e.g., BGP, OSPF) of the route such that $AD_A < AD_B$ denotes routing protocol A is preferred to protocol B.
- *Protocol attributes*: This captures protocol-specific attributes of the routing protocol represented by AD. For example, if the value of AD corresponds to BGP, the protocol attributes field encodes the BGP attributes (i.e., weight, local preference). To enable fast implementation of route selection in our router model (that we will discuss in §5.2), we carefully encode the attributes so that preferring a route between two routes $route_1$ and $route_2$ simply becomes a matter of choosing the smaller of two bit-vectors $AD_1.attrs_1$ and $AD_2.attrs_2$ when interpreted as unsigned integers (the symbol . denotes concatenation of the AD and protocol attributes fields of a route). For example, since route selection in BGP involves checking a prioritized list of BGP attributes (e.g., first checking the weight,

[2]Since our route model resembles routing messages in distance-vector protocols, we accommodate link state protocols (e.g., OSPF) by letting the attributes refer to the routes output by the Dijkstra algorithm.

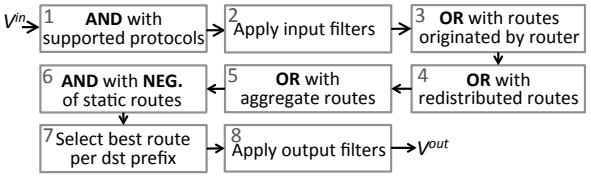


Figure 8: High-level router model processing boolean representation of input routes.

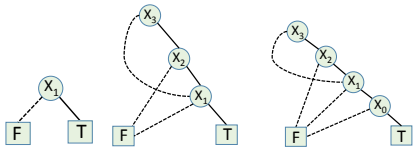
then local preference, etc.) [4], for a BGP route, the highest order bits of the protocol attributes field of the route encode the complement of the BGP weight attribute, followed by the complement of the local preference, and so forth. Note that the designated 87 bits for succinctly capturing protocol attributes have been sufficient in a range of realistic scenarios we have considered (§9), but there might be scenarios where more bits are needed to encode many distinct attributes.

5.2 Control Plane as a Visibility Function

Given the I/O unit of the control plane, next we need to model the processing logic that a router applies to input routes. Intuitively the router model is a function that given a route as its input, computes the corresponding output route(s). We identify 5 key operations of the router control plane: *(i) Input filtering*, which modifies/drops incoming route advertisements to the router; *(ii) Route redistribution*, which is necessary to capture cross-protocol interactions [31,33]; *(iii) Route aggregation*, which is a common mechanism to shrink forwarding tables, yet its improper use can lead to reachability violations [34]; *(iv) Route selection*, which is in charge of selecting the best route to a given destination prefix; and *(v) Output filtering*, which modifies/drops outgoing route advertisements.

Unfortunately, reasoning about the control plane one routing announcement at a time is not scalable. Instead, we *lift* our router model to work simultaneously on a *set* of route announcements. We refer to our router model as the *visibility function* because it captures how the router control plane processes the routing information made visible (i.e., given as input) to it. The input to the router visibility function, V^{in}, is the set of input routes sent by its neighbors and configured static routes; and its output, V^{out}, is the set of corresponding output routes that are sent downstream by the router. The notation $V^{out}_{Router} = T_{Router}(V^{in}_{Router})$ denotes the control plane visibility function of $Router$.

For fast exploration, we use BDDs to symbolically encode the set of I/O routes in a router model. A BDD is a compressed representation of a boolean function that enables fast implementation of operations such as conjunction, disjunction, and negation [30]. Our BDD encoding enables fast router operations by transforming operations



(a) RIP. (b) Static route. (c) Output filter.

Figure 9: Example router model as a BDD. Dashed and solid lines represent the values 0 and 1 of the corresponding binary variable, respectively.

on sets to quick operations on BDDs. For example, taking the complement of a set simply requires flipping the true/false leaves of the corresponding BDD.

Figure 8 shows the high-level procedure for processing a boolean representation of sets of routes. (For completeness, the pseudocode for this is presented in Appendix B.) The steps to turn V^{in} into V^{out} are as follows:

1. *Supported protocols:* First, the routing protocols present in the configuration file are accounted for.
2. *Input filtering*: Then, the input filters are applied.
3. *Originated routes*: In addition to the input route, there are routes that directly stem from the configuration files, which are conceptually ORed with the input.
4. *Route redistribution*: A route redistribution command propagates routing information from a routing protocol (e.g., BGP) into another protocol (e.g., OSPF).
5. *Route aggregation*: If the router receives any input route that is more specific than any configured aggregate route, the aggregate route gets activated.
6. *Static routes*: A static route is a route locally known to the router (i.e., not shared with its neighbors). Further, by default, static routes take precedence over dynamic routes (e.g., OSPF, BGP, RIP, IS-IS) due to having a lower AD value. This behavior is captured by ANDing the negation of static routes with all other routes.
7. *Route selection*: Selecting the best of multiple routes to a destination prefix works as follows: *(i)* if the routes belong to different routing protocols, the one with the lowest AD value is selected, *(ii)* if the routes belong to the same routing protocol, the protocol-specific attributes determine the winner.
8. *Output filtering*: The router applies its output filters.

An illustrative example: We illustrate the procedure of Figure 8 using a small example. For ease of presentation, a route here has only 4 bits $x_3x_2x_1x_0$, with two bits x_3x_2 representing IP prefix, the bit x_1 representing AD, and the bit x_0 representing protocol attributes. A bar over a binary variable denotes its negation. In this example, the network operator assumes the router accepts *all* routes as input, which is captured by setting $V^{in} = 1$ (i.e., $true$).

Suppose a router is configured with a static route and RIP, with AD values of 0 and 1, respectively. Figure 9 shows the BDD representation of the router that has the following four (simplified) configuration commands:

- `RIP`, denoting the presence of RIP on the router, is captured by $1 \wedge x_1 = x_1$, as shown in Figure 9a.
- `static 10/2`: Since this static route overrides the RIP routes with the same prefix, the resulting predicate is $(\overline{x_3}\,\overline{x_2})x_1 = \overline{x_3}x_1 \vee x_2x_1$. This is shown in Figure 9b.
- `output filter: if RIP attribute is 0, make it 1`: The effect of the filter is to replace all occurrences of x_1 by x_1x_0. The resulting predicate is $\overline{x_3}x_1x_0 \vee x_2x_1x_0$. This is captured in Figure 9c.

Intuitively, the output $V^{out} = \overline{x_3}x_1x_0 \vee x_2x_1x_0$, simplified to $V^{out} = (\overline{x_3} \vee x_2) \wedge x_1x_0$, represents the fact that given *every* environment as the input, the router outputs RIP (noted by x_1) with attribute 1 (noted by x_0) and the dest. prefix can be 00, 01, or 11 (noted by $\overline{x_3} \vee x_2$).

In the following section, we will discuss how to reason about the reachability behaviors of the network by exploring the router model we developed in this section.

6 Exploring the Model

Our reachability analysis is based on an exploration of the control plane model above. We first describe this exploration, and then describe how we leverage our BDD-based encoding to devise a set of scalable exploration mechanisms that use *(i)* the Karnaugh map, *(ii)* equivalence classes, and *(iii)* vectorized CPU instructions.

6.1 Exploration Method

We present our approach to finding traffic reachable from port A to port B using a representative example. Consider the scenario shown in Figure 10. The red path is an A-to-B path involving routers $R_A, \ldots, R_i, R_{i+1}, \ldots, R_B$. For ease of presentation, in this example, there is only one path from A to B; the general pseudocode presented in Appendix C accounts for all A-to-B paths.

To see the effect of the environment, consider router R_i, which has three paths to router ports that face the outside world (namely, outside facing ports of routers R_1, R_3, and R_5). Unless the operator makes a more specific assumption on an environment input (i.e., what route advertisements the outside world will send to the network), ERA starts analysis using the boolean value $true$ (represented by a BDD with only one leaf with the value $true$), which represents the fact that *every* possible route are provided by the environment. On the other hand, if the operator is able to make a more scoped assumption about the environment (e.g., based on expected routes from a neighbor), the starting environment will reflect the assumption.

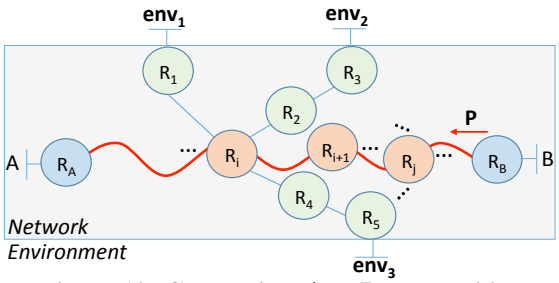


Figure 10: Computing A to B reachability.

Such assumptions can be encoded as a BDD that explicitly includes the relevant variables on the assumed prefix, administrative distance, or attributes values of incoming routes from the environment.

Computing traffic reachable from A to B involves the following steps:

1. *Applying the effect of the environment:* Every router on a A-to-B path that has a topology path to the environment, is affected by it. For router R_i in our examples, it means R_i receives the environment input E_i^{in}, where
$$E_i^{in} = T_1(env_1) \vee T_2(T_3(env_2)) \vee T_4(T_5(env_3))$$
2. *Computing routes reachable from B to A:* As we saw in §4.1, the key to computing traffic prefixes that reach from A to B using control plane analysis is to compute what route prefixes are made visible from B back to A. Let $assumed_B$ show the input the operator assumes about what port B receives from the environment. For the red path, this is captured by
$$reach_{A \rightsquigarrow B} = T_A(E_A^{in} \vee \ldots (T_{i+1}(E_{i+1}^{in} \vee \ldots T_N(E_B^{in} \vee assumed_B) \ldots)))$$
3. *Extracting prefixes reachable from A to B:* Since we are interested in route prefixes reachable from B to A, we eliminate binary variables in the route fields that do not correspond to prefix (i.e., AD and protocol attributes) in all boolean terms of $reach_{A \rightsquigarrow B}$.
4. *Accounting for on-path static routes:* In addition to the routes that reach from B to A, which cause traffic to reach from A to B, there is potentially other traffic that can reach from A to B due to static routes configured on on-path routers. This is because while a router does not advertise its static routes, activated static routes end up in its forwarding table. We account for such prefixes and OR them with the answer from step 3.
5. *Applying ACL rules affecting A-to-B traffic:* While a router configuration file primarily includes directives to configure the router control plane, it may include access control lists (ACLs) that restrict the actual traffic that can pass through the data plane of the router. We, therefore, account for ACLs by taking the result of step 4 and applying the ACLs of the on-path routers.

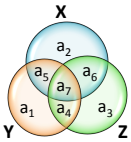


Figure 11: Visualization of predicates X, Y, and Z in terms of members of equivalence classes $a_1, \ldots, a_7$.

Once traffic prefixes reachable from A to B are computed, the network is policy-compliant if the prefixes are equal to $\phi_{A \to B}$ from §4.1. If ϕ is violated, ERA applies the Karnaugh map [27] to the DNF representation of the violating routes to provide the human operator with fewer distinct items to investigate (§4.1); e.g., instead of reporting distinct prefixes $10.20.0.0/17$ and $10.20.128.0/17$ as violations, ERA summarizes and outputs them as $10.20.0.0/16$.

The process above finds policy violations in the context of a single set of environmental assumptions. The user can iterate multiple times with different assumptions in order to expose more errors. Conceptually, each iteration of ERA over a BDD input analyzes a *set* of concrete environments for which the network has an identical behavior. The analysis implicitly identifies this set during exploration, by accumulating constraints from the visibility function of each router in the network. Thus, the number of iterations needed for exhaustive exploration using ERA is far less than those needed with data plane based analysis tools such as Batfish.

6.2 Scalability Optimizations

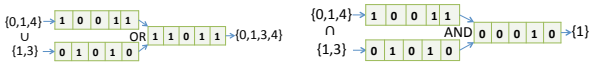
To build an interactive tool for network operators, we want ERA to be able to compute $A - to - B$ reachability in no more than a few seconds. Even with the tractable control plane model that we developed in §5, a naive implementation of the exploration mechanism outlined in §6.1 fails to satisfy our goal. This is because of the very large range of possible environments. Here we present three techniques to scale control plane exploration.

Minimizing collection of routes with the K-map: As a first step, to minimize the binary representation of the router I/O, we apply the Karnaugh map (K-map), which is a common technique in circuit design [27].

Finding equivalence classes: Performing computations (e.g., conjunction and disjunction) on boolean representation of a real control plane is cumbersome. For example, the same or similar destination prefixes may appear on multiple routers. As such, if we encode prefixes naively, this may slow down control plane exploration.

Given this observation, before performing reachability analysis, ERA gets rid of redundant data by finding equivalence classes of routes which are treated identically by the network, using which the data can be rebuilt [42]. The advantage of doing so is that now performing disjunction and conjunction on boolean terms boils down to doing union and intersection on sets of integers (known as atomic predicates [42]). These integers are the indices of the equivalence classes. We illustrate this technique using an example. Suppose we need to compute the conjunction of the boolean terms X, Y, and Z (e.g., representing three routes). Instead of naively computing the conjunction on the raw boolean form of these terms, we do the following:

1. Express each term in terms of equivalence classes as depicted in Figure 11; e.g., $X = a_2 \vee a_5 \vee a_6 \vee a_7$.
2. Represent each term using the indices of members of equivalence classes, e.g., X is the union of members 2, 5, 6, and 7. (This way, irrespective of how bulky the raw form of term a_i might be, it is represented by integer value i.)
3. To compute $X \wedge Y \wedge Z$, intersect the sets of their corresponding indices: $\{1,5,6,7\} \cap \{1,4,5,7\} \cap \{3,4,6,7\} = \{7\}$, which indicates the answer to $X \wedge Y \wedge Z$ is a_7.



(a) Set union using OR. (b) Set intersection using AND.

Figure 12: Fast ∪ and ∩ of two sets of integers.

Implementing fast set operations: As we saw above, using equivalence classes, reachability analysis involves computing union and intersection of sets of integers. We leverage vectorized instructions on recent processors to perform fast set union and intersection of two sets of integers (i.e., the indices of the equivalence classes). The intuition is simple: if a set of integers is represented as a bit vector where each bit represents the presence/absence of the corresponding value, then the union (intersection) of two sets of integers is the bit-wise OR (AND) of the two bit vectors.

Figure 12 shows this approach using an example. In our implementation, we use instructions on 256-bit vectors in our Intel AVX2 implementation [13].

7 Going beyond Reachability

Building on basic A-to-B reachability, ERA can be used to check a wider range of policies. In §9, we will discuss scenarios involving these policies.

Valley-free routing: Operators often want to implement "valley-free" routing [20], which means that traffic from a neighboring peer or provider must not reach another such neighbor. This condition is a form of reachability policy that ERA can easily check.

Equivalence of two routers: Operators often use multiple routers to provide identical connectivity for fault tolerance. Checking if they are identically configured (e.g., using configuration syntax) is hard because the routers may be from different vendors and many aspects of the configuration (e.g., interface IP addresses) can legitimately differ across routers of even the same vendor. To check semantic equivalence of two routers' policies, we use the following property of BDDs: if two boolean functions defined over n boolean variables are equivalent (i.e., they generate the same output for the same input), their Reduced Ordered BDDs (ROBDDs) are identical [17]. In our implementation, we check the equality of the adjacency matrix representations of the BDDs of the two functions, which takes $O(n^2)$. In contrast, a brute force method will take $O(2^n)$.

Blackhole-freeness: A blackhole is a situation where a router unintentionally drops traffic. The blackholed traffic from A to B is the complement of the reachable traffic: $blackhole_{A \rightsquigarrow B} = \overline{reachability_{A \rightsquigarrow B}}$. Note that computing blackholes by ERA having computed reachability takes $O(1)$, as the negation of a BDD is the same BDD with its two leaves (corresponding to true and false) flipped.

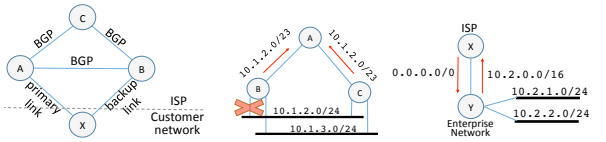
Waypointing: Operators may want traffic from A to B to go through an intended sequence of routers (e.g., to enforce advanced service chaining policies [15,35]). ERA checks waypointing by explicitly checking whether traffic reachable from A to B goes through the intended routers.

Loop-freeness: ERA can find permanent forwarding loops (e.g., created by static or aggregate routes—see Figure13c in §9.1) by checking whether the same router port appears twice in the reachability result.

8 Implementation

Our implementation of ERA [1] supports several configuration languages (e.g., Cisco IOS, JunOS, Arista). It uses Batfish's configuration parser, which normalizes a vendor-specific configurations to vendor-agnostic format. ERA, then, uses this vendor-agnostic format as input. We implement the control plane model, the K-map, and atomic predicates in `Java`. To operate on BDDs, we use the `JDD` library [7]. We implement our fast set intersection and union algorithms in `C` using Intel AVX2, which expands traditional integer instructions to 256 bits [13].

A natural question might be how much effort it takes to add support for various routing protocols to ERA. In our experience, this effort is minimal. It took two of the authors a few hours to model the common routing protocols because of two reasons. First, there are fewer than 10 common routing protocols (e.g., BGP, OSPF, RIP, IS-IS). Second, for each protocol, the key insight for creating the model is to know how the protocol prefers a route over another in the steady state, which is concisely defined in protocol specifications.



(a) Violation of waypointing [32]. (b) Black-hole [34]. (c) Permanent loop [34].

Figure 13: Finding known bugs in synthetic scenarios.

9 Evaluation

In this section, we evaluate ERA and find that:

- It can help find both known and new reachability violations (§9.1).
- It can scale to large networks (e.g., it can analyze a network with over 1,600 routes in 6 seconds), and our design choices are key to its scalability (§9.2);

9.1 Finding Reachability Bugs with ERA

We show the utility of ERA in finding reachability violations in scenarios involving known bugs as well as new bugs across both real and synthetic scenarios. These scenarios illustrate violations that are latent and get triggered only in certain environments (i.e., a certain router advertisement sent to the network by the routers located int the outside world). Even for scenarios involving only a small number of routers, existing network verification techniques lack the ability to find latent bugs (§2), and trying to extend these tools to enumerate different environments poses a serious scalability challenge (e.g., we will quantify this for Batfish, a recent network verification tool, in §9.2). Further, as we will discuss in §9.2, ERA scales to large networks (e.g., over 1,600 routers).

All experiments below were done under the assumption that the environment sends *all* possible route announcements, i.e. the BDD of each environmental input is simply the predicate *true*. Though this environmental assumption is not guaranteed to cover all possible environments, in practice it is effective at rooting out latent bugs due to its "maximal" nature, as we show below. This points out an important advantage of ERA over Batfish [19]. While both tools require an environment as input, Batfish's low-level simulation of routing protocols makes it prohibitively expensive to run with such a maximal environment, so in practice Batfish users must craft specific environments that are suspected to cause problems.

9.1.1 Finding Known Bugs in Synthetic Scenarios

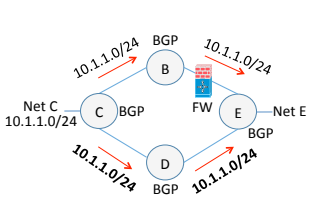
- **Violation of waypointing due to route redistribution:** In this scenario borrowed from [32] and shown

in Figure 13a, the customer wants to waypoint its traffic through $X - A - C$ and use $X - B - C$ as the backup path. However, static routes configured on routers A and B are redistributed into BGP, and the ISP advertises them into the rest of the Internet. As a result, $B - X$ acts as a primary link. (One way to prevent this would be for the customer to adjust the default AD values of BGP and static routes on B.)

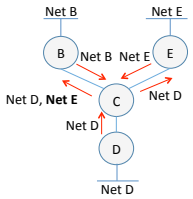
- **Blackhole due to route aggregation:** In this scenario borrowed from [34] and shown in Figure 13b, both routers B and C are configured to announce aggregate route 10.1.2.0/23 to router A. After the marked interface of B fails, B continues to announce the aggregate route, which causes A to send packets destined to 10.1.2.0/24 to B. B will drop this traffic, as the its link to the 10.1.2.0/24 subnetwork is down.
- **Permanent loop due to route aggregation:** In this scenario borrowed from [34] and shown in Figure 13c, the ISP router X advertises the default route 0.0.0.0/0 to router Y. Even though Y has connectivity to only 10.2.1.0/24 and 10.2.2.0/24, it has been configured to advertise to the ISP the aggregate route for the entire 10.2.0.0/16 prefix. Now since 10.3.0.0/24 is as subprefix of 10.2.0.0/16, the ISP may send traffic to destination prefix 10.3.0.0/24 to Y. Consequently, since Y does not know how to reach 10.3.0.0/24, this traffic will match its default route entry and be bounced back to the ISP. This traffic, therefore, will trap in a permanent loop between X and Y.

To further evaluate the effectiveness of ERA, we did a red team-blue team exercise. In each scenario, the red team introduced misconfigurations that cause a reachability violation unbeknownst to the blue team. Then the blue team uses ERA to check whether the intended policy is violated. Across all scenarios, the blue team successfully found the violation. Here is a summary of the scenarios:

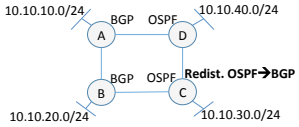
- **Violation of waypointing:** In Figure 14a, the intended policy is to ensure traffic originating from network E destined to network C goes through path $E - B - C$ (so that it is scrubbed by the firewall). However, this policy is violated because router E receives the prefix of network C from both routers B and D, which means $NetE \rightarrow NetC$ traffic may go through path $E - D - C$ skipping the firewall. The root cause of the problem was the fact that none of routers C, D, or E filtered the route advertisement for the 10.1.1.0/24 prefix on the $E - D - C$ path.
- **Violation of valley-free routing:** In Figure 14b, B and E are providers for C, which in turn, is a provider for D. A missing export filter on C caused C to advertise the prefix for $NetE$ to B. This is a violation



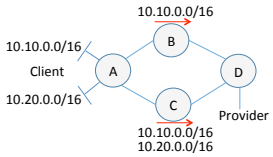
(a) Violation of waypointing via B.



(b) Not valley-free.



(c) Violation of isolation between $\{A, B\}$ and $\{C, D\}$.



(d) Misconfigured backup path $D - B - A$.

Figure 14: Finding known bugs in synthetic scenarios using the red-blue teams exercise.

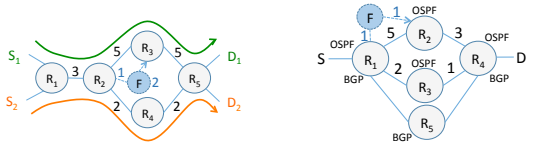
of the valley-free routing property, specifically, due to customer C providing connectivity between two of its providers, namely, B and E.

- **Violation of intended isolation:** In Figure 14c, we want the traffic from segments $\{A, B\}$ (running BGP) and $\{C, D\}$ (running OSPF) to remain isolated from each other. However, this policy is violated due to a misconfiguration on C whereby OSPF is redistributed into BGP, that will allow traffic from $\{A, B\}$ to reach $\{C, D\}$.
- **Misconfigured backup path:** In Figure 14d, the client has two /16 networks connected to A and intends to maintain two paths to the provider to ensure reachability in case of failure on one of them. This policy is violated because of an incorrect filter configured on B that drops the advertisement for the 10.20.0.0/16 network. As a result, if path $D - C - A$ fails, the 10.20.0.0/16 network will be unreachable from the provider.

9.1.2 Finding New Bugs in Synthetic Scenarios

Finding reachability bugs in hybrid networks: Operators may prefer to opt for a hybrid network, which involves deploying SDN alongside traditional network routing infrastructure for scalability and fault tolerance [40]. Next we show how ERA can find policy violations arising in such hybrid deployments.

Fibbing [40] is a recent method to allow an operator to use an SDN controller to flexibly enforce way-pointing policies in a network running vanilla OSPF. The key primitive is "fibbing" whereby the SDN controller pretends to be a neighboring router and makes fake route advertisements with carefully crafted costs. For example, consider the network of Figure 15a, where links are annotated with their OSPF weights. If we run OSPF, both



(a) Route aggregation on R_2. (b) Cross-protocol effects.

Figure 15: New bugs in a synthetic scenario involving hybrid (i.e., SDN-traditional) networks.

source to destination flows will take the cheaper path $R_1 - R_2 - R_4 - R_5$. Now, for load balancing purposes, the operator wants to make $S_1 \rightarrow D_1$ traffic take the path $R_1 - R_2 - R_3 - R_5$ without fiddling with OSPF wights. Fibbing will let her accomplish this by using a fake router F that claims to be able to reach D_1 at a cost of 2. As a result, now R_2 will start sending traffic destined to D_1 through F, as the new cost 1+2=3 is better than the cost 2+2=4 of going through R_4.

A hybrid network is particularly error-prone due to intricate interactions between SDN and traditional protocols. To show the utility of ERA in reasoning about such networks, we describe two scenarios:

- **Interaction between fibbing and aggregate routes:** In Figure 15a, the goal is to use fibbing to enforce the waypointing denoted by green and orange paths. We used ERA to find a violation of this policy. The root cause was an aggregate route configured on R_2 to destination prefix $D_1 \cup D_2$ pointing to R_4 as its next hop. As a result, both $S_1 \rightarrow D_1$ and $S_2 \rightarrow D_2$ traversed the orange path, which violated the policy.
- **Cross-protocol effects:** In Figure 15b, the goal is to use fibbing to waypoint traffic to D through $R_1 - R_2 - R_4$. We used ERA to find a violation of this in a red team-blue team exercise. Each router in the figure is annotated with the routing protocol(s) it runs. Router R_4 had a static route to D that is redistributed into BGP and OSPF. As a result, router R_1 received route advertisements for D from both OSPF (from R_2 and R_3) and BGP (from R_5). Now since BGP, by default, has a lower AD value than OSPF, R_1 chose the advertisement offered by R_5! Therefore, fibbing here fails to enforce the waypointing policy.

Fibbing is proven to be correct [40], but only if the network merely runs OSPF. The takeaway from the above scenarios is that for hybrid networks to be practical, we need to account for realistic router configurations (e.g., route aggregation by R_2 in Figure 15a) and cross-protocol interactions (e.g., BGP/OSPF in Figure 15b).

Note that finding arbitrary SDN bugs is beyond the scope of ERA. ERA handles SDN only if its behavior can be abstracted in our control plane model, in a manner similar to what we do for conventional routing protocols.

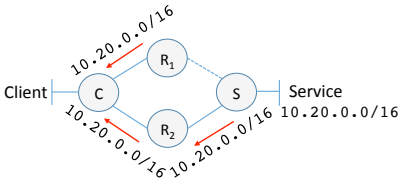


Figure 16: R_1 leaks the service prefix.

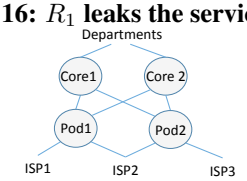


Figure 17: A schematic of the analyzed *CampusNet*.

9.1.3 Finding Known Bugs in Real Scenarios

Bugs reported in a cloud provider: The motivating scenarios we saw in §3 are based on bugs in a production network that we successfully reproduced using ERA.

Finding BGP route leaks: Roughly speaking, a route leak scenario involves: *(i)* a router incorrectly advertising the destination prefix of a service, and *(ii)* another router incorrectly accepting it. The combination of these results in absorbing traffic destined to the service on the wrong path, which can cause high-impact disruptions. Route leak is not a new problem (e.g., see AS 7007 incident [2]), but continues to plague the Internet to date (e.g., Google [6] and Amazon AWS [11] outages in 2015). To demonstrate the utility of ERA in proactive finding of route leak-prone configurations, we use a representative scenario shown in Figure 16. The intended path from the client to the service is through R_2; however, the client's traffic ends up taking the wrong path $C \rightarrow R_1$ because *(i)* R_1 incorrectly advertises the service prefix, and *(ii)* C prefers the route advertisement made by R_1 over the one made by R_2. ERA can proactively find route leaks, as a route leak is essentially a violation of waypointing. In this example, the traffic from client to server needs to be exclusively waypointed through R_2. We have synthesized a few route leak scenarios and used ERA to successfully find violations.

9.1.4 Finding New Bugs in Real Scenarios

Next we show the utility of ERA in finding new bugs in a campus (*CampusNet*) and a large cloud (*CloudNet*).

Finding new bugs in *CampusNet*: Figure 17 shows a simplified topology of the core of a large campus network, with a global footprint and over 10K users. The two core routers are in charge of interconnecting the three ISPs and the departments. There are two intended policies involving these four routers, both of which are violated:

- **Equivalence of core routers:** $Core2$ is meant to be $Core1$'s backup. ERA revealed that $Core1$ has OSPF

configured on one of its interfaces, which is missing on $Core2$. As a result, if $Core1$ fails, the departments that rely on OSPF will be disconnected from the Internet.

- **Equivalence of pod routers:** $Pod1$ and $Pod2$, connecting the campus to the Internet, are both connected to $ISP2$ with the intention that link $Pod1 - ISP2$ is active and $Pod2 - ISP2$ is its backup. ERA revealed that the ACLs on $Pod1$ and $Pod2$ affecting their respective links with $ISP2$ are different. Specifically, $Pod2$ has more restrictive ACLs than $Pod1$. This means if link $Pod1 - ISP2$ fails, a subset of campus-to-$ISP2$ traffic will be mistakenly dropped by $Pod2$.

Finding new bugs in *CloudNet*: We used ERA to check equivalence of same-tier routers (analogous to routers R_1 and R_2 in Figure 2) on configurations of seven production datacenters of a large cloud provider. ERA revealed that seven routers in two datacenters had a total of 19 static routes responsible for violations of equivalence policies. The operators later removed all of these violating routes.

9.2 Scalability of ERA

Testbed: We run our scalability evaluation experiments on a desktop machine (4-core 3.50GHz, 16GB RAM).

Why not existing tools? The closest tool to ERA is Batfish [19], which *(1)* takes a concrete network environment; *(2)* runs a high-fidelity model of the control plane (e.g., low-level models of various routing protocols) to generate the data plane (i.e., routers forwarding tables); and *(3)* performs data plane reachability analysis. To put this in perspective, in an example scenario involving a chain topology with two routers, Batfish took about 4 seconds. In contrast, ERA took 0.17 seconds to analyze the same network (a 23X speedup over Batfish). Further, as mentioned earlier, Batfish's performance will degrade as the size of the environment increases, while ERA's BDD-based approach allows it to naturally handle even the "maximal" environment, represented by the BDD *true*.

Effect of optimizations: Table 1 shows the effect of our optimizations from §6.2, namely, the K-map, equivalence classes (EC), and fast set operations compared to a baseline involving use of BDDs without these optimizations. The tables shows the average values from 100 runs, each involving A-to-B reachability analysis between two randomly selected ports. Stanford [12] and Purdue [10] are campus networks, OTEGlobe [14] is an ISP, and FatTree is a synthetic datacenter topology. The takeaway here is that our optimizations yield a speedup of 2.5× to 17×, making ERA sufficiently fast to be interactively usable.

To see the effect of the type of policy on the analysis latency, we measured the analysis latency for all properties from §7 on the Purdue and OTEGlobe topology, none of which took more than 6.1 seconds. This is expected, as these policies are derivatives of reachability analysis.

Topo.	#routers/ave path len.	Reachability analysis latency (sec)			
		baseline	kmap	kmap+EC	ERA
Stanford	16/2	5	1.8	0.30	0.29
OTEglb	92/3.3	7.8	3.5	1.97	1.84
FatTree	1,024/5.89	13.8	7.01	6.1	5.4
Purdue	1,646/6.8	15	8	6.5	6

Table 1: Effect of our optimizations.

10 Conclusions

Since networks are constantly changing (e.g., new route advertisements, link failures), operators want the ability to reason about reachability policies across many possible changes. In contrast to prior work, which either focuses on a subset of the network's control plane or focuses on one incarnation of the network as represented by a single data plane, ERA models the entire control plane and checks network reachability directly in that model. Our design addresses key expressiveness and scalability challenges via a unified protocol-invariant routing abstraction, a compact binary decision diagram based encoding of the routers' control plane, and a scalable application of boolean operations (e.g., vector arithmetic).

We showed that ERA provides near-real-time analysis capabilities that can scale to datacenter and enterprise networks with hundreds of nodes and uncover a range of latent reachability bugs. While ERA does not automatically reason about all possible of environments, it helps find latent reachability bugs by allowing the users to specify a rich *set* environments using BDDs and quickly analyzing each such set. For instance, a particularly challenging environment, of all possible routing announcements from a neighbor, can be captured simply using BDD *true*.

In future work, we will identify conditions under which a single run of ERA is guaranteed to cover all possible environments and extend ERA to automatically explore all possible environments. Another natural direction for future work is to prioritize bug fixing based on the likelihood of occurrence and severity of aftermath, and to bring the human operator into the debugging and repair loop.

Acknowledgments

We thank our shepherd George Porter and the OSDI reviewers for their constructive feedback. This work was supported in part by NSF Awards CNS-1552481 and CNS-1161595 and by a VMware Graduate Fellowship.

Appendix

The goal of the following appendices is to provide details of our control plane modeling and exploration approach that we presented in §5 and §6.

A Computing Destination Prefix

To make our route abstraction compact, we store the destination mask field in 5 bits (instead naively storing it in 32 bits) as we saw in §5.1. Here we concretely describe how we do this. Let $dstIP$ and $dstMask$ denote a 32-bit destination IP address and our 5-bit encoding of the destination mask. To compute the destination prefix that the destination IP and mask represent, we first transform the mask to its customary 32-bit representation (e.g., 255.255.0.0), and then AND it with the IP address:

$dstPrefix \leftarrow dstIP \& ((2^{32} - 1) << (32 - dstMask))$

where <<denotes the shift left operator.

B Route Visibility Function

For completeness, the pseudocode of Figure 18 shows the details of the router control plane model from §5.2. The pseudocode describes how a configured router turns the boolean representation of its input routes to output routes. Note that we account for per-port (i.e., router interface) behaviors because, in general, a router can have distinct routing behaviors configured on its different ports.)

1. The input to the router is the disjunctive normal form (DNF) boolean representation of input routes. This represents the input route(s) the environment of the router (i.e., its neighbors) sends to it (line 1). The output of the pseudocode is the DNF boolean representation of the output routes (line 2).
2. First, the routing protocols present in the configuration file are accounted for (lines 6-7).
3. Then, the input filters are applied to the input route (Lines 8-12).
4. In addition to the input route, there are route advertisements that directly stem from the configuration files (e.g., the `network` bgp configuration command). These are unioned with the input route in lines 13-14.
5. A route redistribution command propagates routing information from a routing protocol, denoted by $fromProto$ (e.g., BGP) into another, denoted by $toProto$, (e.g., OSPF). This is captured in lines 15-22.
6. A route aggregation (a.k.a. route summarization) command works as follows: if the router receives any input route that is more specific than an aggregate route, the aggregate route is activated (lines 23-28).[3]

```
1  ▷ Inputs: (1) Configuration information pertaining to router output port
     Router_port including: static routes sr[.], route redistribution rr[.], route
     aggregation ra[.], supported routing protocols proto[.], input filters if[.],
     output filters of[.]
     (2) Input to the router is a boolean function in DNF form:
     V^in = X_1^in ∨ ··· ∨ X_N^in
2  ▷ Output: Boolean representation of Router_port in DNF

3  ▷ Route bit vector from Figure 7, denoted by X, is concatenation of 3 fields:
     X = X_prefix.X_proto.X_attr
4  ▷ We show the length of an array array by size(array[.])
5  V^out = V^in ▷ Initializing the output

6  ▷ Applying supported routing protocols
7  V^out = V^out ∧ {⋁_i X_proto[i]}

8  ▷ Applying input filters
9  for i = 1 to size(if[.])
10   for each disjunctive term of V^out, denoted by V_j^out
11     if V_j^out matches if[i].condition
12       apply action if[i].action

13 ▷ Accounting for routes that Router originates, denoted by V^local
14 V^out = V^out ∨ V^local

15 ▷ Applying route redistribution
16 for i = 1 to size(rr[.])
17   for each disjunctive term of V^out, denoted by V_j^out
18     if V_j^out.X_proto == rr[i].fromProto
19       newTerm = V_j^out
20       newTerm.X_proto = rr[i].toProto
21       newTerm.X_attr = defaultAttr[proto]
22       V^out = V^out ∨ newTerm

23 ▷ Applying route aggregation
24 for i = 1 to size(ra[.])
25   newTerm.X_prefix = ra[i].prefix
26   newTerm.X_proto = ra[i].proto
27   newTerm.X_attr = defaultAttr[proto]
28   V^out = V^out ∨ newTerm

29 ▷ Applying static routes
30 for i = 1 to size(sr[.])
31   for each disjunctive term of V^out, denoted by V_j^out
32     if AD(V_j^out.X_proto) > AD(static)
33       V_j^out = V_j^out ∧ (¬sr[i].prefix)

34 ▷ Applying route selection
35 for each prefix prfx present in V^out
36   precedence = +∞
37   for each disjunctive term of V^out, denoted by V_j^out
38     if (V_j^out.prefix == prfx)&&(V_j^out.AD.attr < precedence)
39       precedence = V_j^out.AD.attr ▷ Finding best route
40   for each disjunctive term of V^out, denoted by V_j^out
41     if (V_j^out.prefix == prfx)&&(V_j^out.AD.attr > precedence)
42       Eliminate V_j^out from V^out ▷ Eliminating others
43   V_j^out = V_j^out ∨ prfx.precedence

44 ▷ Applying output filters
45 for i = 1 to size(of[.])
46   for each disjunctive term of V^out, denoted by V_j^out
47     if V_j^out matches of[i].condition
48       apply action of[i].action

49 return V^out
```

Figure 18: Route control plane visibility function.

7. A static route, if present in the configuration file, is a route locally known to the router (i.e., not shared with its neighbors). Further, by default, static routes have a

[3] In line 21 (line 27) of Figure 18, if there are explicit attributes configured for route redistribution (route aggregation), we use those values instead of default attributes.

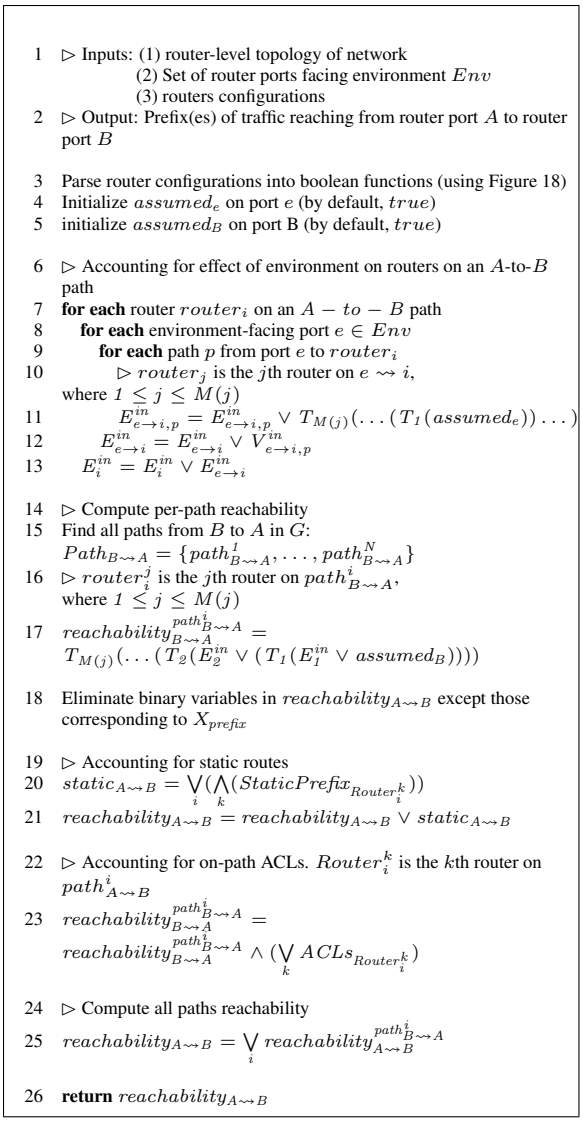


Figure 19: Computing A-to-B reachability.

lower AD value than dynamic routing protocols (e.g., OSPF, BGP, RIP, IS-IS), which makes them take precedence over these protocols. These behaviors are captures in lines 29-33.

8. Route selection is in charge of selecting the best route out of multiple routes to the same destination prefix: (i) if the routes belong to different routing protocols, the routing protocol with the lowest AD value is selected, (ii) if the routes belong to the same routing protocol, the protocol-specific attributes determine which one is selected. We have encoded protocol-specific attributes in such a way that a smaller value denotes a more preferred route. Route selection is shown in lines 34-43.
9. As lines 44-48 denote, the last operation of the router is applying the output filters.

C Computing Traffic Reachable from A to B

We saw the high-level procedure to compute the traffic reachable from port A to port B in the network in §6.1. For concreteness, here we present the pseudocode for doing so (Figure 19).

1. First, we account for the effect of the environment on the routers that are located on a path from A to B (lines 6-13).
2. The pseudocode then computes the routes that can reach from B to A over all paths between the two ports. For each path, we sequentially use the visibility functions of the on-path routers (lines 16-17). At this point, we have computed all route advertisement prefixes that reach from B to A, which is the traffic prefixes that reach from A to B.
3. Then, since we are interested in route prefixes reachable from B to A, we ignore route fields that do not correspond to prefix (i.e., AD and attributes) in line 18.
4. In addition to these prefixes, there is potentially other traffic that can reach from A to B to static routes configured on on-path routers. This is because while a router does not advertise its static routes, proper static routes end up in its forwarding table. By a proper static route we mean a static route that points to the next on-path router as its next hop. We account for static routes in lines 19-20.
5. Since routers ACL rules restrict what traffic prefixes will actually be forwarded, we then account for them in lines 22-23.

Finally, the computed per-path reachability results are unioned (lines 24-25).

References

[1] ERA. https://github.com/Network-verification/ERA.

[2] 7007 Explanation and Apology. http://bit.ly/1e4djtW.

[3] BGP Message Generation and Transport, and General Message Format. http://bit.ly/1VMOI0R.

[4] Border Gateway Protocol Path Selection. http://bit.ly/1T1w7IH.

[5] Cisco—What Is Administrative Distance? http://bit.ly/1OkgevM.

[6] Finding and Diagnosing BGP Route Leaks. https://blog.thousandeyes.com/finding-and-diagnosing-bgp-route-leaks/.

[7] JDD, a pure Java BDD and Z-BDD library. https://bitbucket.org/vahidi/jdd/wiki/Home.

[8] Juniper—Route Preferences. http://juni.pr/1fQC4LY.

[9] OSPF Message Formats. http://bit.ly/1TvOwwL.

[10] Purdue campus network configuration files. https://engineering.purdue.edu/~isl/network-config/.

[11] Route Leak Causes Amazon and AWS Outage. https://blog.thousandeyes.com/route-leak-causes-amazon-and-aws-outage/.

[12] Stanford campus network configuration files. http://bit.ly/1rvoK5h.

[13] The Intel Intrinsics Guide. http://intel.ly/24sk3uz.

[14] The Internet Topology Zoo. http://www.topology-zoo.org/dataset.html.

[15] High Performance Service Chaining for Advanced Software-Defined Networking (SDN) . http://intel.ly/1ilX5PG, 2014.

[16] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proc. SIGCOMM*, 2016.

[17] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[18] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proc. NSDI*, 2005.

[19] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *Proc. NSDI*, 2015.

[20] L. Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Trans. Netw.*, 9(6), Dec. 2001.

[21] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *Proc. SIGCOMM*, 2016.

[22] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan. Management plane analytics. In *Proc. IMC*, 2015.

[23] T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.*, 10(2):232–243, Apr. 2002.

[24] T. G. Griffin and J. L. Sobrinho. Metarouting. In *Proc. SIGCOMM*, 2005.

[25] T. G. Griffin and G. Wilfong. An analysis of BGP convergence properties. In *Proc. SIGCOMM*, 1999.

[26] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. SIGCOMM*, 2016.

[27] F. J. Hill and G. R. Peterson. *Introduction to Switching Theory and Logical Design*. 1981.

[28] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proc. NSDI*, 2012.

[29] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proc. NSDI*, 2013.

[30] D. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. 2011.

[31] F. Le, G. G. Xie, D. Pei, J. Wang, and H. Zhang. Shedding light on the glue logic of the internet routing architecture. In *Proc. SIGCOMM*, 2008.

[32] F. Le, G. G. Xie, and H. Zhang. Instability free routing: beyond one protocol instance. In *Proc. CoNEXT*, 2008.

[33] F. Le, G. G. Xie, and H. Zhang. Theory and new primitives for safely connecting routing protocol instances. In *Proc. SIGCOMM*, 2010.

[34] F. Le, G. G. Xie, and H. Zhang. On route aggregation. In *Proc. CoNEXT*, 2011.

[35] W. Liu, H. Li, O. Huang, M. Boucadair, N. Leymann, Z. Cao, and J. Hu. Service Function Chaining (SFC) Use Cases. `http://bit.ly/1JTVneh`, 2014.

[36] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *Proc. NSDI*, 2015.

[37] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *Proc. SIGCOMM*, 2011.

[38] D. A. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjálmtýsson, and A. Greenberg. Routing design in operational networks: A look from the inside. In *Proc. SIGCOMM*, 2004.

[39] G. Varghese. Technical perspective: Treating networks like programs. *Commun. ACM*, 58(11):112–112, Oct. 2015.

[40] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford. Central control over distributed routing. In *Proc. SIGCOMM*, 2015.

[41] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock. Bagpipe: Verified BGP configuration checking. In *Proc. OOPSLA*, 2016.

[42] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *IEEE Transactions on Networking*, 2015.

[43] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNEXT*, 2012.