Fast Control Plane Analysis Using an Abstract Representation

Aaron Gember-Jacobson[*◦], Raajay Viswanathan[*◦], Aditya Akella[◦], Ratul Mahajan[†]
[◦]University of Wisconsin-Madison, [†]Microsoft Research
[◦]{agember,raajay,akella}@cs.wisc.edu, [†]ratul@microsoft.com

ABSTRACT

Networks employ complex, and hence error-prone, routing control plane configurations. In many cases, the impact of errors manifests only under failures and leads to devastating effects. Thus, it is important to proactively verify control plane behavior under arbitrary link failures. State-of-the-art verifiers are either too slow or impractical to use for such verification tasks. In this paper we propose a new high level abstraction for control planes, ARC, that supports fast control plane analyses under arbitrary failures. ARC can check key invariants without generating the data plane—which is the main reason for current tools' ineffectiveness. This is possible because of the nature of verification tasks and the constrained nature of control plane designs in networks today. We develop algorithms to derive a network's ARC from its configuration files. Our evaluation over 314 networks shows that ARC computation is quick, and that ARC can verify key invariants in under 1s in most cases, which is orders-of-magnitude faster than the state-of-the-art.

CCS Concepts

•**Networks** → **Control path algorithms;** *Network dynamics; Network reliability;*

Keywords

Network verification; control plane; abstract representation

1. INTRODUCTION

A network's routing control plane is responsible for generating the data plane (i.e., forwarding tables) using one or more distributed routing protocols (e.g., OSPF, RIP, BGP). Prior work has shown that the configuration of these routing protocols [5, 6] and their interactions [19, 21] can be quite complex in modern networks. Consequently, control planes are prone to configuration errors that compromise network security, availability, and performance [8, 25].

Many control plane errors manifest *only* during failures and can have a devastating impact. For example, in 2012, failure of a router in a Microsoft Azure data center triggered previously unknown configuration errors on other devices, degrading service in the West Europe region for over 2 hours [23]. Another large class of errors arises when refactoring a network's control plane: e.g., consolidating routing domains to improve manageability [5], changing routing protocols to improve scalability [18], or replacing old devices, potentially with hardware from a different vendor. These common errors highlight the importance of *proactively* analyzing a control plane.

Unfortunately, many network verification tools [15, 16, 17, 20] analyze a network's current data plane. This limits the scope of their analyses to the current live network and prevents them from being used for proactive analysis. To overcome this limitation, more recent tools, such as Batfish [9], simulate the control plane and generate the network's expected data plane under specific failure scenarios, e.g, a single link failing. However, these tools operate at a low level of abstraction, modeling individual protocol message exchanges to generate the data plane. As such, they tend to be slow. Further, these tools must generate the complete data plane for every possible failure scenario of interest. These attributes render them impractical for several key tasks, such as proactively verifying certain security and availability invariants under arbitrary failures, where the tools must generate an exponential number of data planes.

Fortunately, we observe that detailed data plane generation is not always necessary due to two factors. First, proactive analysis tasks often require computing *properties* of paths, not the paths themselves. For example, invariants I1–I4 in Table 1 focus on the existence (or absence) of paths; I5 relies on the set of paths taken, but we show that actually computing the paths is unnecessary. Second, many enterprise and data center networks use only a handful of routing protocols which interact in very specific ways (§7.1).

We leverage the above factors to develop a new abstraction that operates at a higher level than today's control plane

[*]These authors contributed equally to this work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGCOMM '16, August 22-26, 2016, Florianopolis , Brazil
© 2016 ACM. ISBN 978-1-4503-4193-6/16/08...$15.00
DOI: http://dx.doi.org/10.1145/2934872.2934876

verifiers, enabling more direct proactive analyses. We call this *abstract representation for control planes*, or ARC. ARC enables the aforementioned proactive analyses to run orders of magnitude faster than state-of-the-art tools.

ARC abstracts the mechanics of individual routing protocols and simply captures the collective impact they *could* have on the network's data plane. ARC is composed of a series of weighted digraphs that are routing protocol-independent. We develop algorithms for generating ARCs that accurately model the common protocols (OSPF, RIP, and eBGP) and mechanisms (static routes, ECMP, access control lists, and route redistribution) used in enterprise and data center networks; for example, our university network and hundreds of data center networks operated by a large online service provider (OSP) use these constructs (§7.1). To maintain ARC's efficiency, we do not model protocols that are less common in enterprise and data center networks (e.g., iBGP).

Crucially, the ARC's edges and vertices are chosen such that the true forwarding path between network locations under any failure scenario is provably included in the ARC's digraphs. Consequently, verifying key security and availability invariants (e.g., I1–I4 in Table 1) boils down to computing simple *graph characteristics* of the ARC, such as connected components and max-flow, which run in polynomial time.

Furthermore, for control planes using a restricted set of features—e.g., AS path length is the only path selection criterion used by eBGP instances—we show how to configure edge weights in the ARC such that the shortest path computed on the ARC under a given failure scenario is the same as the path computed by the real network. Our university network and a large fraction (97%) of the data center networks we study use such a restricted set of features. In these cases: (1) We can use the ARC to generate a counter-example for violations of the security and availability invariants (I1–I4) in Table 1, where the counter-example includes a failure scenario and an invariant-violating path.[1] Operators can use these examples to make proactive fixes to buggy configurations before rolling them out into a live network. (2) We can enable equivalence testing (I5) for networks satisfying the restrictions by simply comparing the ARCs for the old and new configurations.

We implement our ARC generation algorithms in Java, using Batfish [9] to parse configurations written in vendor-specific languages; our code is publicly available [1]. To evaluate ARC, we check the control planes of 314 data center networks operated by a large OSP against the invariants in Table 1. We find that each network's ARC can be generated in a few seconds. Verifying many of the invariants under arbitrary failure scenarios takes less than 1s for 99% of networks. Checking $I3$ across collections of traffic flows is the most time-consuming: it can take up to a few tens of minutes. In contrast, state-of-the-art tools [9] are 3-5 orders of magnitude slower in checking limited failure scenarios.

2. MOTIVATION

[1]We can still verify (but not present a counter-example for) I1–I4 even if networks do not satisfy the restrictions.

Invariant	Example
I1: Always blocked	External hosts can never communicate with hosts in subnet S
I2: Always reachable with $< k$ failures	Up to 5 links can fail without breaking connectivity between subnets S_1 and S_2
I3: Always isolated	Traffic between subnets S_1 & S_2 and S_3 & S_4 never traverses the same link simultaneously
I4: Always traverse waypoint	Traffic between external hosts and internal hosts must always traverse a firewall
I5: Equivalent(C_1, C_2)	Traffic between hosts must always traverse the same paths if control plane C_2 were to replace C_1

Table 1: Invariants of common interest

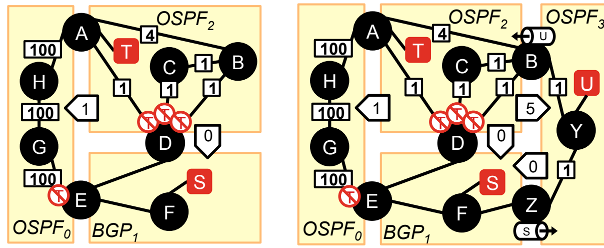
The network control plane is the heart of a network. It may be composed of multiple routing domains, or *routing instances*. Each routing instance is a collection of processes running on different routers that exchange information (e.g., link-state updates) using a specific protocol (e.g., OSPF, RIP, BGP) [13, 21]. Routing processes on the same device may exchange routes with each other using route redistribution. Static routes may also be used. The routing instances collectively generate the network's data plane, i.e., forwarding tables, based on protocol-specific algorithms (e.g., Bellman-Ford), various parameters (e.g., link weights), access control lists (ACLs), and the current state of network links. In addition, operators may define knobs (e.g., redistribution costs and administrative distances) that determine how a router chooses among the many routes it learns via the processes configured on it. Routing processes and their parameters, access controls, and the operator defined knobs are specified in a router's configuration file.

Recent work has shown that most networks' control planes use complex designs to realize sophisticated goals [5, 6, 19]. Unfortunately, the complexity makes these control planes error prone [12]. In particular, critical errors may arise *only* during failures, e.g., when one or more links fail simultaneously, or while refactoring the control plane's design.

In this paper, we focus on two important ways of identifying such errors in control planes: (1) *verifying security and availability invariants hold across arbitrary failures* and (2) *equivalence testing*. Below, we illustrate the importance of these tasks using a toy example. We argue that no existing tool is capable of performing these tasks, at least not feasibly on sufficiently large networks. We then highlight opportunities we can leverage to efficiently support the two tasks.

2.1 Satisfying Invariants

Figure 1a shows an example network's control plane. It uses three routing instances: one BGP instance and two OSPF instances. Routers A, E, G, H run processes for the $OSPF_0$ instance, routers D, E, and F run processes for the BGP_1 routing instance, and routers A, B, C, and D run processes for the $OSPF_2$ instance. The picture also shows links costs (e.g., 4 on the A-B link), route redistribution and the cost of such redistributed routes (e.g., from $OSPF_2$ to BGP_1, whose cost is 0), and data plane ACLs (e.g., at router D, on the



(a) Initial control plane **(b)** Expanded control plane

Figure 1: An example enterprise network: Circles represent routers and rectangles represent routing instances. Links between routers are labeled with OSPF costs. No-entry symbols represent ACLs blocking traffic destined for T. Arrows between instances indicate route redistribution, and specify the cost assigned to such routes. Tubes represent static routes that are redistributed in the direction of the arrow.

A-D link). One of the objectives this control plane ensures, among many, is that S cannot communicate with T.

Security & availability. To avoid undesirable outcomes from manifesting under failures, operators may require that certain key security and availability invariants always hold in their network (i.e., even under arbitrary failures). In the above example, it is easy to see the invariant "subnet S can *never* send traffic to subnet T" always holds, because the only possible path from S to T is via D or E, and every interface on D (E) that participates in the $OSPF_2$ ($OSPF_0$) routing instance has an ACL that blocks traffic destined for T. Now assume the enterprise acquires a startup and connects the startup's network—represented by $OSPF_3$ in Figure 1b—to the existing network. To allow subnets S and T to communicate with subnet U, the operator configures route redistribution and static routes on routers B and Z.

Unfortunately, the change introduces the subtle side-effect that S can now send traffic to T under *some* specific failure scenarios, violating the operator's requirement that the invariant always hold. In particular, without any failures, there is a cheaper path from S to T: $F{\rightarrow}Z{\rightarrow}Y{\rightarrow}B{\rightarrow}D{\rightarrow}A$; since this passes through D, traffic is still blocked. However, if the B–D and C–D links both fail, the new cheapest path goes directly from B to A, bypassing the ACLs on D.

Equivalence. In some situations, operators refactor the network's control plane to simplify device configurations and improve manageability. For example, an operator may want to combine multiple OSPF instances (e.g., $OSPF_2$ and $OSPF_3$ in Figure 1b) into a single OSPF instance [5]. Operators inform us that, in making such a change, they wish to ensure the optimized control plane is equivalent to the original control plane: i.e., under *arbitrary failures* the new and old control planes should generate the *same data plane*. Testing for equivalence is also important when operators alter their control plane design to use a different set of protocols (e.g., replace OSPF with BGP for scalability reasons [18]) or different hardware (e.g., replace old devices, potentially with hardware from a different vendor). It is difficult to evaluate the equivalence of such changes, because the raw device configurations look very different.

2.2 Limitations of Existing Verifiers

Current verification tools are designed to check the network in its present state or under a limited set of failure scenarios (e.g., all single link failures). As a result, current tools are incapable of, or very inefficient at, both determining that the above reachability invariant can be violated and testing if the optimized control plane is equivalent to the original.

Data plane modeling. Many tools [15, 16, 17, 20] build a model of the data plane based on snapshots of device forwarding tables, or SDN control messages. Because they verify the current data plane, these tools *cannot proactively check* if an invariant, such as our reachability example above, would be satisfied if links failed. Likewise, data plane verifiers cannot be used for equivalence testing.

Control plane modeling. Older control plane tools model specific devices (e.g., firewalls [24]) or routing protocols (e.g., BGP [8]). As such, they are not well suited for verifying today's enterprise and data center networks, which make use of multiple device types and routing protocols [12].

A more recent tool, Batfish [9], models several routing protocols and their interactions using Datalog. This allows Batfish to *generate data plane models* for a set of failure scenarios and verify an invariant holds across the generated data planes. Unfortunately, tools such as Batfish are slow because they do not abstract the network at all, but instead try to mimic low level protocol interactions and generate a full data plane. This can take as long as a few minutes (§7.3). Furthermore, Batfish must generate the data plane for every possible k link failure scenario. In our reachability example above, Batfish can only detect the invariant violation after generating and examining $\mathcal{O}(|\ell|^2)$ data planes for single and two-link failure scenarios, where ℓ is the set of links in the network. In the worst case, Batfish must generate an exponential (in $|\ell|$) number of data planes, making it impractical.

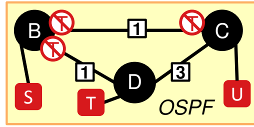
For equivalence testing, control plane verifiers must generate data planes under all possible failures for both control planes and compare them, which is again impractical.

2.3 Opportunities for Improvement

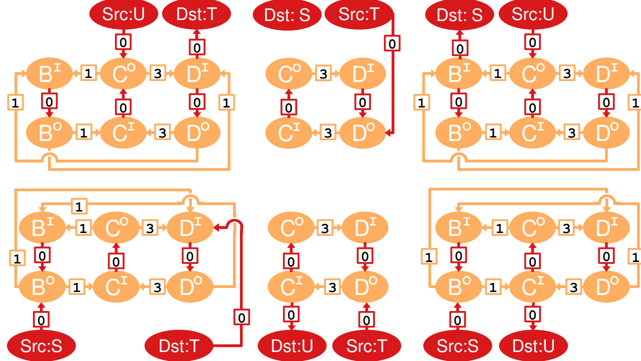
Our insight is that, in practice, checking many key invariants does not require computing the actual forwarding paths. In the reachability example above, which illustrates I1 in Table 1, we ideally only need to check if S and T are in different connected components of a logical graph induced by the network's control plane configuration and physical topology. As we show later, the remaining invariants (I2–I5) can also be analyzed without generating the data plane.

Furthermore, many networks, especially enterprise and data center networks employ a limited set of routing constructs in their control plane design. We list the key attributes we observe in our university network and the data center networks of a large online service provider in Table 3. Notably, only a handful of routing protocols are used, and they operate and interact in limited ways.

Thus, our intuition is that by focusing on what checking key invariants actually entails and by considering constrained control plane designs, we can develop a new ab-



(a) Control plane for a network with three subnets (squares) and 3 routers (circles) participating in a single OSPF instance (rectangle); no-entry symbols indicate inbound ACLs on traffic from T



(b) Abstract representation for the control plane (ARC): it contains one digraph for every pair of source and destination subnets; vertices correspond to routing processes; edges represent the possible flow of traffic enabled by the exchange of routing information between the connected processes

Figure 2: Example network with three endpoint groups and three routers participating in a single OSPF instance

straction that operates at a higher level than today's control plane verifiers and enables more direct analysis.

3. A NEW ABSTRACTION: ARC

To avoid modeling a network's data plane, which depends on the current state of network links, and instead analyze the network at a higher-level, we present an *abstract representation for control planes* (ARC).

A network's ARC is a data structure that contains a collection of *weighted digraphs*, one for each "traffic class", i.e., a source-destination subnet pair. As an example, Figure 2b shows the ARC for the simple control plane in Figure 2a; the ARC has six graphs: one for each possible combination of source and destination subnets. Each digraph models the behavior of the routing instances/protocols in the control plane, and the interactions among them, with respect to the corresponding traffic class. Vertices correspond to routing processes—each has an *in* (I) and *out* (O) vertex for reasons described in §4.2. Directed edges represent the possible flow of data traffic enabled by the exchange of routing information between the connected processes.

For an ARC to be useful for verification and equivalence checking, its constituent digraphs must satisfy two key attributes: *pathset-equivalence and path-equivalence*. We describe these next.

Pathset-equivalent graphs. Each digraph in an ARC is constructed such that it contains every path between the source and destination endpoints that is used in the real network, and does not contain any path that is infeasible in the real network, under arbitrary failures. We say such a digraph is *pathset-equivalent*, because it encodes all possible and no impossible forwarding behaviors, respectively. In §4.2, we describe how to construct provably pathset-equivalent graphs for networks that use OSPF, RIP, eBGP, static routes, ACLs, route filters, and route redistribution.

One of the main benefits of a pathset-equivalent ARC is that verifying invariants $I1$–$I4$ in Table 1 for arbitrary link failures boils down to *checking simple graph attributes*. For example, suppose we want to verify that "subnet T can never send traffic to subnets S or U under any link failures" in the network shown in Figure 2a. Assuming the graphs in Figure 2b are pathset-equivalent, this can be done by checking if T and S (or T and U) are in separate connected components of the graphs for the corresponding traffic classes (center graphs in Figure 2b). Because T and U are in the same connected component in the lower-center graph, there is *some* link failure scenario where the invariant is violated and T can send traffic to U (e.g., when the B–D link fails).

Path-equivalent graphs. To aid operators in debugging violations, and allow for fast equivalence testing, the edge weights in each digraph are assigned such that, after removing edges corresponding to failed links, the *min-cost path* in the digraph between the source and destination vertices is the exact path taken in the real network. We say such a graph is *path-equivalent*, because it encodes the network's actual forwarding behavior under arbitrary link failures.

For example, when there are no link failures in the network in Figure 2a, traffic from S to U takes the path $S \rightarrow B \rightarrow C \rightarrow U$, which is the min-cost path in the lower-right graph in Figure 2b. When the $B - C$ link fails, the actual and min-cost path is $S \rightarrow B \rightarrow D \rightarrow C \rightarrow U$. While in this example, edge weights are the same as OSPF cost metrics, in a real ARC the weights are a function of the relative rank of specific routing protocols, AS paths, and network links.

In §4.3, we describe how to construct provably path-equivalent graphs for networks under some restrictions, i.e., the route redistribution policy is acyclic and the costs assigned to redistributed routes are congruent with each process's administrative distance (AD).

When the digraph is path-equivalent, we can produce all min-cost paths from T to U as counter-examples to the aforementioned invariant. The operator can use this to add the missing ACL to C and prevent T and U from ever communicating. Additionally, we can check the equivalence of two control planes by directly comparing the graphs contained in their ARC. If each graph in each control plane's ARC has the same vertices and edges, and the edge weights are proportional, then the control planes are equivalent.[2]

The main challenge in constructing ARCs is determining the appropriate vertices, edges, and weights to use for the graphs to be pathset- and path-equivalent.

[2]An equivalent ordering of edges by weight does not guarantee the ordering of paths is the same: e.g., changing the weight of the B–C edge in Figure 2a to 2.5 results in the same ordering of edges but causes the path $D \rightarrow C$ to be preferred over the path $D \rightarrow B \rightarrow C$.

4. GENERATING A NETWORK'S ARC

We start by discussing the practical challenges in designing the ARC. Then, we motivate and present our approach for constructing a pathset-equivalent ARC. Finally, we describe our algorithms for deriving edge weights to get a path-equivalent ARC.

For simplicity, we focus on networks that use OSPF, RIP, eBGP, static routes, AD-based route selection, route redistribution, data plane ACLs, and/or route filters. These are the constructs we find in our campus network and hundreds of data center networks operated by a large OSP (§7.1). However, other protocols (e.g., EIGRP) can be accommodated through extensions to our algorithms.

4.1 Opportunities and Challenges

Modeling the collective behavior of multiple routing instances in a series of weighted digraphs in the ARC is enabled by the fact that *most routing protocols in use today employ a cost-based path selection algorithm*. For example, OSPF uses Dijkstra's algorithm to compute min-cost paths from a source to all destinations; RIP computes shortest paths using the Bellman-Ford algorithm. If multiple min-cost paths are available and ECMP is enabled, then traffic is evenly divided among the paths using multi-path routing. BGP associates cost labels with paths based on numeric metrics: e.g., operator-defined local preference, path length, and multi-exit discriminator (MED) [7, 14]. These have similar properties to link costs used in IGPs, except BGP costs are per-path rather than per-link.

While these similarities allow us to use weighted digraphs to model routing behavior, differences between protocols introduce at least two challenges:

1. In the actual control plane, interior and exterior gateway protocols (IGPs and EGPs, respectively) compute routes at different *granularities*. An IGP treats each router as a node, while an EGP views each AS as a node. Fortunately, enterprise and data center networks tend to use EGPs in restricted ways (§4.3.3) that mirror IGPs' view.
2. Each routing protocol uses a different *currency* for expressing link and path costs/preferences: e.g., a link with an OSPF cost of 1 may be less desirable than an AS path whose local preference is 1, or vice versa. Thus, we cannot directly add or compare costs between protocols. However, in real network control planes, redistributed routes are assigned fixed costs (§4.3.4); this masks the costs used in other routing instances and provides an avenue for reconciling differences in currency.

There are other subtle aspects of network routing that also impact our modeling:

- *Traffic-class-specific policies.* Only certain classes of traffic are blocked by data plan ACLs and route filters.
- *Redistribution of routes between routing instances.* A routing process may advertise routes computed by another routing instance, allowing traffic to traverse a path composed of segments selected by different protocols.
- *Selection of routes based on AD.* When multiple routing processes on the same device identify a route to a destination, only the route from the process with the lowest administrative distance (AD) is installed in the device's global routing information base (RIB) [19].

We next describe how we select ARC vertices, edges, and weights to accommodate the above issues.

4.2 ARC Vertices and Edges

A network's physical topology may seem like a natural starting point for the ARC's graphs. By having a vertex for each router and an edge for each physical link, we can assign edge weights based on the per-interface cost metrics defined for IGPs (e.g., OSPF and RIP) and the AS preferences defined for BGP. However, this is too coarse to express route selection and redistribution policies between routing processes running on the same device.

4.2.1 *Extended Topology Graph*

To accommodate these features, we introduce an abstraction we call an *extended topology graph* (ETG). Figure 3 shows the ETG for the example control plane depicted in Figure 1b. Vertices in the ETG correspond to individual routing processes.[3] Directed edges represent inter- and intra-device communication paths between routing processes, including: *hardware paths*—a single physical link or multiple physical links that form a layer-2 network—and *software paths*—inter-process communication channels used to exchange information between processes on the same device.

Some aspects of a network's control plane only apply to specific traffic classes: e.g., data plane ACLs, route filters, and static routes. To accommodate these features, an ARC includes a *customized ETG for each traffic class*. As mentioned earlier, a traffic class represents the set of traffic flowing from one *endpoint group*—a set of related hosts, subnets, etc.—to another. We use the network prefixes in device configurations, including prefixes assigned to interfaces, advertised by routing processes, and referenced in ACLs, as the basis for determining a network's endpoint groups. Because some prefixes may overlap, we use standard firewall rule optimization algorithms [10] to compute a set of non-overlapping prefixes. We generate a list of traffic classes by enumerating all possible pairings of prefixes.

Modeling forwarding behavior at the level of routing processes results in an ARC that is not protocol-independent. This model is nevertheless useful to answer control plane verification questions. In §5.2, we show, under restricted assumptions, how to transform an ETG from a process-based to an interface-based model, resulting in a protocol-independent ARC that can be useful for equivalence testing.

4.2.2 *Constructing ETGs*

We now describe how to construct ETGs from device configurations. The complexity of constructing an ETG is $\mathcal{O}(\max_r |I_r|^2 + \max_r |I_r| * \max_i |R_i|)$, where $\max_r |I_r|$ is the maximum number of routing processes running on a single device, and $\max_i |R_i|$ is the maximum number of devices participating in a single routing instance.

[3]Static routes are also viewed as a routing process.

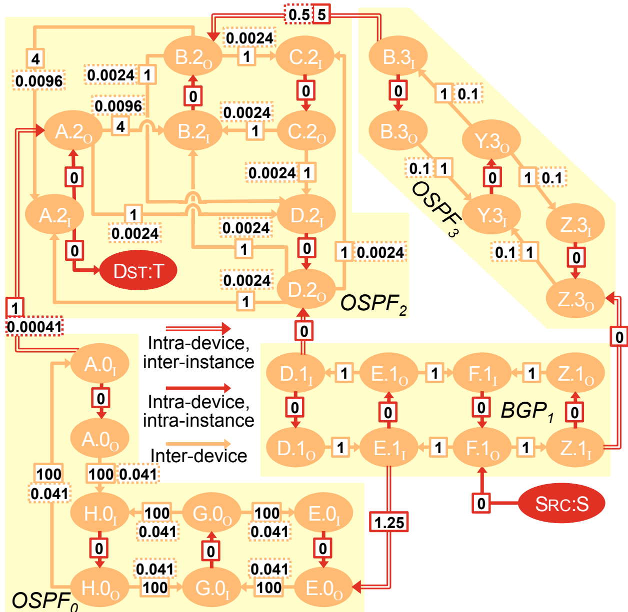


Figure 3: ETG for the control plane in Figure 1b (sans ACLs) for the $S \rightarrow T$ traffic class: light shaded regions indicate routing instances; the structure and weights are the same for all traffic classes, with the exception of endpoint edges, edges removed due to ACLs, and static route vertices; weights in dashed boxes are assigned by our scaling algorithm to model route redistribution and selection

Vertices. The ETG contains two vertices (*in* and *out*) for each routing process, including static routes. For example, the processes on routers B, Y, and Z for routing instance *OSPF*$_3$ in Figure 1b are represented by vertices $B.3_\mathrm{I}$, $B.3_\mathrm{O}$, $Y.3_\mathrm{I}$, $Y.3_\mathrm{O}$, $Z.3_\mathrm{I}$, and $Z.3_\mathrm{O}$ in Figure 3. We use two vertices per process in order to accommodate route selection and redistribution (described in detail below). We identify a network's routing processes from the `router` stanzas in device configurations [21].

We also add special source and destination vertices (SRC and DST, respectively) to the ETG to represent the source and destination endpoints associated with the traffic class.

Inter-device edges. The *out* vertex for a routing process on one device is connected to the *in* vertex for a process on another device if: (1) the two devices are connected by a (sequence of) physical link(s),[4] and (2) the routing processes participate in the same routing instance. Such an *inter-device edge* thus represents two things. First, it represents the direct exchange of routing information (e.g., link-state updates or AS-level path advertisements) within a routing instance. Second, it represents a possible physical path over which data traffic may be forwarded due to the RIB entries resulting from the aforementioned exchange of routing information. Inter-device edges always go from an *out* vertex to an *in* vertex and point in the direction data traffic flows, which is the inverse of the direction routing information flows.

For example, in the network shown in Figure 1b, the *BGP*$_1$ routing process on router E may compute a route to the subnet S via router F as a result of routing information sent by the *BGP*$_1$ process on router F. The flow of routing information from F to E and the resulting flow of data traffic from E to F is represented by the edge from $E.1_\mathrm{O}$ to $F.1_\mathrm{I}$ in Figure 3. There is a similar edge from $F.1_\mathrm{O}$ to $E.1_\mathrm{I}$, because routing information also flows from E to F and may result in the flow of data traffic from F to E.

Unlike routes computed by IGP and BGP processes, static routes are not based on advertisements from a specific neighboring process. Thus, we connect a static route's *out* vertex to the *in* vertices for *all processes on the next hop device*—i.e., the device with an interface whose IP address matches the next hop IP specified in the static route.

Intra-device edges. The ETG also contains edges between vertices associated with routing processes running on the same device to model the flow of data traffic resulting from route redistribution and route selection.

When a routing process (*redistributor*) is configured to redistribute routes into another process (*redistributee*) on the same device, we connect the redistributee's *in* vertex to the redistributor's *out* vertex; similar to above, this *intra-device* edge points in the opposite direction that routing information flows. For example, the redistribution of routes from routing instance *OSPF*$_2$ to *OSPF*$_3$ is represented by the edge from $B.3_\mathrm{I}$ to $B.2_\mathrm{O}$ in Figure 3. Similarly, we connected a process's *in* vertex to its own *out* vertex to model a routing process's role in propagating routes. For example, the route propagation performed by the process for *OSPF*$_3$ on B is represented by the edge from $B.3_\mathrm{I}$ to $B.3_\mathrm{O}$ in Figure 3.

When multiple routing processes on a device have a route to the destination, the route from the process with the lowest administrative distance (AD) is used to reach the destination. The aforementioned intra-device edges already capture cases where the lowest-AD process with a route to the destination: (1) redistributes routes into higher-AD processes, or (2) advertises a route to the destination to processes on neighboring devices. However, we must also model the case where a higher-AD process (H) advertises a route to the destination to processes on neighboring devices, but the lower AD process's (L) route will be used at the device in question to reach the destination. This situation only occurs when both H and L have independently learned routes to the destination (i.e., there is no route redistribution). We model this case by connecting H's *in* vertex (H_I) to L's *out* vertex (L_O), assuming there exists at least one path from H_I to DST and L_O to DST prior to the addition of this edge.

Endpoint edges. Edges are added from the SRC vertex to a routing process's *out* vertex if the device on which the process runs can be directly reached by the source endpoint(s) using layer-2 forwarding: e.g., SRC $\rightarrow A.2_\mathrm{O}$ in Figure 3. Similarly, edges are added from a routing process's *in* vertex to the DST vertex if the device on which the process runs can directly reach the destination endpoint(s): e.g., $F.1_\mathrm{I} \rightarrow$ DST. For traffic classes whose source endpoint is external, we add an edge from SRC to the *out* vertices of all processes that send external route advertisements; we add similar edges for

[4] We assume two devices are connected if they each have an interface that participates in the same subnet [21].

external destinations.

Factoring in ACLs and route filters. Data plane ACLs prevent particular classes of traffic from entering or leaving a router. Similarly, route filters prevent a routing process from advertising particular prefixes to a process on another device, or a process on the same device through route redistribution.

To account for these filtering mechanisms, we prune some edges from the ETG. In particular, we prune an inter-device edge if: (i) there is an outgoing or incoming data plane ACL configured on the interfaces associated with the physical link(s) the edge represents, and (ii) the ACL blocks the traffic class associated with the ETG. We also prune an inter-device edge if a route filter that blocks the traffic class's destination prefix has been applied to the process whose *out* vertex is incident with the edge. Similarly, we prune an intra-device edge if a route filter that blocks the traffic class's destination prefix is applied to routes redistributed by the process whose *out* vertex is incident with the edge.

4.2.3 Pathset-equivalence

We now prove the above methodology results in pathset-equivalent ETGs. We first show that a path-equivalent ETG is also pathset-equivalent. (Our technical report [11] contains proofs of ETG path-equivalence.)

THEOREM 1. *A path-equivalent ETG is pathset-equivalent.*

PROOF. Let P be the min-cost path in the ETG from SRC to DST under some failure. Now assume the actual network has a more preferred path P' between the source and destination, but P' does not exist in the ETG. Because P' does not exist in the ETG, the min-cost path in the ETG is incorrect. This contradicts the assumption that the ETG is path-equivalent. Thus, a path-equivalent ETG must contain every path taken by the actual network under all possible failures.

Now assume the ETG contains a path P'' from SRC to DST which is infeasible in the actual network. Also assume all edges not on the path have been removed due to failures. The only, and hence min-cost, path through the ETG will be P''. Because P'' is infeasible in the actual network, the min-cost path in the ETG is incorrect. This contradicts the assumption that the ETG is path-equivalent. Thus, a path-equivalent ETG must not contain any paths that are infeasible in the actual network. □

For some route redistribution policies we cannot generate a path-equivalent ETG (details in §4.3). However, we can still generate a pathset-equivalent ETG.

THEOREM 2. *An ETG is pathset-equivalent when routes are redistributed between OSPF, RIP, and/or eBGP instances.*

We refer readers to our technical report [11] for the proof.

4.3 ARC Edge Weights

While pathset-equivalent ETGs are sufficient for verifying many important invariants, such as I1–I4 in Table 1, path-equivalent ETGs are required for generating counterexamples or testing equivalence (I5). The key challenge in constructing path-equivalent ETGs is determining the appropriate edge weights such that the min-cost path through the ETG matches the actual path taken in the network under arbitrary failures. Next, we describe how to assign such weights to different types of edges. In our technical report [11], we prove the resulting ETGs are path-equivalent.

4.3.1 Endpoint edges

When assigning weights to endpoint edges, we must consider the route selection policies of the devices to which the source and destination endpoints are connected.

Source edges. When the source is connected to a device with *one routing process*, then the best route (if any) computed by that process is always used. Thus, the edge from SRC to the process's *out* vertex is assigned a weight of 0.

If the device has *multiple routing processes*, then a route computed by a process with a lower AD is preferred over a route computed by a process with a higher AD. We model this by assigning edge weights proportional to a process's AD. The weight of an edge from SRC to $r.i_{\mathrm{O}}$ is set to

$$AD_i * \max_{i' \in I_r} \left(\sum_{e \in E_{i'}} w_e \right) \tag{1}$$

where AD_i is the AD for routing instance i, I_r is the set of routing instances in which router r participates, $E_{i'}$ is the set of edges originating from the *in* and *out* vertices for the routing processes in instance i', and w_e is the weight assigned to edge e. This ensures the cost of a path originating at a process with a higher AD is always more expensive than the longest possible path through a routing instance whose process has a lower AD.

Destination edges. When the destination is directly connected to a device, the device always sends traffic directly to the destination; no other route is ever preferred. Thus, edges to DST are assigned a weight of 0.

4.3.2 Inter-device edges for IGPs

For inter-device edges connecting RIP or single-area OSPF processes, we directly assign the cost metric specified in the device configurations. If no cost is explicitly defined, we assign the DEFAULT-RIP-COST or DEFAULT-OSPF-COST defined by the device vendor. For example, edges $A.2_{\mathrm{O}} \rightarrow B.2_{\mathrm{I}}$ and $B.2_{\mathrm{O}} \rightarrow A.2_{\mathrm{I}}$ in Figure 3 are assigned the OSPF cost configured on the $A - B$ link in Figure 1b. This methodology, along with the methodology described in §4.3.4 and §4.3.5, also allows us to identify the multiple min-cost paths that are used when ECMP is enabled.

4.3.3 Inter-device edges for eBGP

eBGP processes inside the network. We model the primary path selection criterion used by eBGP for computing paths: AS path length. We do not model other path selection criteria, e.g., local preference, because: (1) it is not possible to statically assign edge weights such that local decisions (e.g., local preferences) override global cost computations (e.g., route selection based on AS path length) without compromising our modeling of the global cost computations; and

(2) local preference is not widely used to select between private ASes within enterprise and data center networks (§7.1).

In the absence of iBGP (which we show in §7.1 is not used in the networks we study), each autonomous system (AS) can only have a single eBGP speaker (i.e., process) that is directly connected to the eBGP speakers of neighboring ASes.[5] Thus, the length of an AS path is simply the number of eBGP processes traversed. We capture this by assigning a weight of 1 to inter-device edges connecting eBGP processes. For example, edges $F.1_{\mathrm{O}} \rightarrow E.1_{\mathrm{I}}$ and $E.1_{\mathrm{O}} \rightarrow D.1_{\mathrm{I}}$ in Figure 3 are assigned weight 1.

eBGP processes outside the network. We cannot precisely model paths that depend on advertisements from external eBGP processes, because we do not know the length of paths advertised. As discussed in §4.2.2, if the source or destination is external, we simply add an edge to/from the *out*/*in* vertex, respectively, of every eBGP process inside the network that peers with an eBGP process outside the network. Edge weights are assigned as described in §4.3.1.

4.3.4 Intra-device edges for route redistribution

As discussed in §4.2.2, route redistribution is modeled via intra-device edges connecting the *in* vertex of one routing process (the *redistributee*) to the *out* vertex of another process (the *redistributor*). When assigning weights to these edges, we must consider the *fixed costs* assigned to redistributed routes. Routing processes within the redistributee's routing instance use these fixed costs, along with the costs assigned to links within the instance, to determine the forwarding path through the instance.

Modeling route redistribution is difficult, because paths through the ETG are computed globally and include edge weights associated with multiple routing instances; whereas in an actual network the path to the destination through the redistributor's routing instance (denoted by i) does not affect forwarding within the redistributee's routing instance (denoted by j). To ensure an ETG is path-equivalent, we: (1) rescale the weights of edges in the ETG such that even the longest path through routing instance i is less than the minimum difference of path lengths within instance j, and (2) assign fixed costs to the intra-device edges representing redistribution in accordance with the redistributee instance's scaling factor.

Constraints. To produce a path-equivalent ETG for a network leveraging route redistribution, the control plane must satisfy two constraints. First, route redistribution must be acyclic—i.e., a routing instance's routes are not redistributed back to itself. Prior work has shown that cyclic route redistribution is fragile [19], so data center and enterprise networks tend to use only acyclic route redistribution. We can check if a network's route redistribution policy is acyclic by constructing a graph with one vertex for each routing instance and directional edges between instances in the direction routes are redistributed (the inverse of intra-device edges in the ETG).

Second, the fixed costs assigned to redistributed routes must be congruent with the ADs assigned to the redistributor(s). More formally, if $AD_i < AD_{i'} < ..$, then it must be the case that $c_{i,j} < c_{i',j} < ...$, where $c_{i,j}$ is the fixed cost assigned to routes redistributed from routing instance i to instance j. This constraint ensures we accurately model the fact that a route is redistributed only when: (1) the redistributor is the only routing process on the device that has a route to the destination, or (2) the redistributor has the lowest AD among the processes that have a route to the destination.

Scaling edge weights. We scale the weights of all edges between vertices corresponding to the redistributor's routing instance (i)[6] by a scaling factor f_i. The scaling factor is computed using the equation on line 15 of Algorithm 1. C is the set of routing instances into which i redistributes routes; the scaling factor f_i must be less than the scaling factor for all $j \in C$. The term g_j denotes the minimum non-zero difference in cost between any two acyclic paths between any two vertices corresponding to routing processes associated with routing instance j. We can conservatively compute g_j by finding the greatest common divisor (GCD) of the unscaled edge weights corresponding to edges associated with routing instance j. For example, the GCD for edges associated with routing instance BGP_1 in Figure 3 is 1. The denominator in the equation for f_i represents the maximum possible path length through routing instance i. The scaling factor for a routing instance can only be obtained after we have determined the scaling factor for all instances into which it redistributes routes (C); if $|C| = 0$, the scaling factor is 1.

Consider the control plane shown in Figure 1b and the corresponding ETG in Figure 3. Assume the G–E link has failed. A packet from $S \rightarrow T$ can either be forwarded through router D or through router Z via the $OSPF_3$ routing instance. In the absence of link failures, the path through $OSPF_3$ will be chosen by BGP_1, because the hop count to the destination is 1, as opposed to a hop count of 2 through router D. However, the shortest path through the ETG with unscaled weights is: $\textsc{Src} \rightarrow F.1_O \rightarrow E.1_I \ldots D.1_I \rightarrow D.2_O \ldots A.2_O \rightarrow \textsc{Dst}$; this path is chosen because of the higher cost of the path through $OSPF_3$. However, by scaling weights using the computed scaling factors, we ensure that no path to the destination through router D is shorter than a path through router Z, because weights on edges in BGP_1 dominate overall cost.

4.3.5 Intra-device edges for route selection

While the above methodology applies to routers whose processes engage in route redistribution, we need a slightly different approach for routers where multiple routing processes have a route to the destination but there is no route redistribution—i.e., there is route selection without route redistribution. Thus, after the above methodology has been applied, we apply the methodology described below.

[5] An AS without iBGP can have multiple eBGP speakers, but each eBGP speaker can only compute paths through ASes with which it has a direct connection; different eBGP processes in the same AS cannot directly exchange routes.

[6] Including intra-device edges corresponding to route redistribution from another routing instance k to instance i.

Algorithm 1 Procedure to determine scaling factors

Input:
I: the set of all routing instances in the network
$\leq_I$: the partial order over I determined by route redist.
Output:
$f_i, h_i, \forall i \in I$: the scaling factors for all routing instances

```
1: f_i = 1, h_i = ∞,   ∀i ∈ I                        ▷ Init. scaling factors
2: ≤ = ≤_I                                           ▷ Init. partial order
3: RESCALE(I, ≤_I)          ▷ Route redistribution rescaling §4.3.4
4: for all routers, r, with no route redist. do        ▷ Route selection
5:     for all i ∈ I_r do                              ▷ rescaling §4.3.5
6:         h_i = f_i g_i / (1+|I|)
7:         for all i' ∈ I_r do
8:             if AD(i') < AD(i) then
9:                 ≤ = ≤ ∪ (i', i)
10:                D = DOWNSTREAMINST(i', I, ≤)
11:                RESCALE(D, ≤_I)

12: procedure RESCALE(T, ≤)
13:     for all i ∈ reverseTopologicalSort(T, ≤) do
14:         C = getChildren(i, ≤)
15:         f_i = min_{j∈C} min(h_j, f_j g_j) / (1 + Σ_{e∈E_i} w_e)
16:         if h_i < ∞ then
17:             h_i = f_i g_i / (1+|I|)          ▷ Update h_i based on new f_i

18: function DOWNSTREAMINST(i, I, ≤)
19:     D = {i}
20:     for all j ∈ I do
21:         if (j, i) ∈ ≤ then
22:             D = D ∪ j
23:     return D
```

Consider the example control plane in Figure 1b (sans ACLs) and the corresponding ETG in Figure 3. There is no route redistribution on router E. However packets arriving on E destined to T have two possible paths: (1) through router D, and (2) through routers G and H via the routes learned through $OSPF_0$. Under the scenario where there are no failures, the path through $OSPF_0$ will never be taken, because $OSPF_0$'s routes are not redistributed within BGP_1. However, if the F–Z link fails, F will forward packets to E along the two hop path $F \rightarrow E \rightarrow D$ to the destination; at E, the process with the lowest AD ($OSPF_0$) dominates and the packets will be forwarded along G and H, ignoring the path computed by the BGP_1 process on E.

To ensure that any path through router E is more expensive than a path through $OSPF_3$ and less expensive than the path through router D, weights of the intra-device route selection edges are obtained by summing two components:

$$w = f_i \left(\min_{b \in B_i \cup \{\textsc{Dst}\}} dist_i(r, b) \right) + h_i * rank(i') \qquad (2)$$

The first component is the minimum cost path from the current router (r) to a border router with a path to the destination ($b \in B_i$), or to the destination itself; we include the cost of intra-device route redistribution edges at a border router. For example, in the BGP_1 routing instance, there are two border routers with a path to the destination: D and Z; the path from E to D is shorter (1 versus 2). This ensures that the path from SRC to DST through router E is greater than a path through $OSPF_3$. The second component is used to distinguish between different processes on the current router (r), each with a route to DST, based on AD's. The routing instance to which the route selection edge points is denoted by i' and $rank(i')$ is the rank of routing instance i' based on its AD. The example has only one alternate process, $E.0$, whose rank is 1 (the process $E.1$ has rank 2). The scaling factor h_i ensures the path through $OSPF_0$ is lower-cost than the path through router D; h_i is computed using the equation on line 6 of Algorithm 1.

Construct	Pathset	Path-equivalent
OSPF	yes	single area
RIP	yes	yes
eBGP	yes	select only by AS path length
Static routes	yes	yes
ECMP	yes	yes
ACLs	yes	yes
Route filters	yes	yes
Route redistrib.	yes	acyclic + costs & ADs align
Route selection	yes	instance's processes use same AD

Table 2: Control plane constructs modeled in ARC

After h_i's are obtained for all the routing instances, the downstream routing instances[7] need to be re-scaled such that the weights on the intra-device route selection edges dominate the costs of the routing instances to which they point. Thus, we update f_i using the equation on line 15 of Algorithm 1. Upon recomputing f_i, the scaling factors for all downstream instances have to be recomputed.

Algorithm 1 provides an overview of the steps involved in computing the scaling factors for all routing instances. It assumes that for a pair of routing instances, i and i', the relative AD ordering is the same in all routers, and if a route selection edge from a process in i to a process in i' exists, then i does not redistribute routes to i'. This restriction is required to avoid any cyclical dependency between instances in terms of re-scaling.

Table 2 summarizes the protocols and features for which an ETG is pathset- and path-equivalent. A network with any combination of these constructs results in ETGs that are pathset-equivalent and, if the listed constraints are met, path-equivalent. In the next section, we describe how to use a network's ARC for verification and equivalence testing.

5. USING ARC

ARC enables us to check important invariants across arbitrary failure scenarios. It is particularly well suited for verifying invariants that pertain to properties of a path. In such cases, verification/equivalence testing is a matter of (dis)proving that an ETG, or a pair of ETGs for different control planes, has a specific *graph-level* characteristic. This section describes our verification and equivalence testing algorithms that at their essence compute such graph characteristics. Furthermore, we describe how to use precise ETGs

[7]downstream routing instances refer to possible instances encountered on a path to the destination (line 18 in Alg. 1).

to generate counter-examples when violations occur. These help an operator take corrective actions before a buggy control plane is made "live" on the network.

5.1 Verifying Security/Availability

Invariants $I1$–$I4$ in Table 1 can be expressed as graph characteristics that can be computed on a pathset-equivalent ETG using polynomial-time graph algorithms.

I1: Always blocked. For security reasons, an operator may want to ensure that a particular traffic class is always blocked. For this to be true under arbitrary failure scenarios, there must not exist a path from *src-node* to *dst-node* in the traffic class's ETG. We can check for the existence of a path by traversing (e.g., depth-first or breadth-first) the ETG starting from *src-node*. If *dst-node* remains unvisited, then the property holds. Otherwise, assuming the ETG is path-equivalent, we provide the shortest path as a counterexample.

I2: Always reachable with $< k$ failures. To improve availability, an operator may want to ensure that a particular destination d can always be reached from a particular source s as long as there are fewer than k link failures in the network. To verify this, we can leverage properties of graph cuts. In particular, according to Menger's Theorem, the maximum number of edge-disjoint paths from s to d in a digraph equals the minimum number of edges whose removal separates s and d [3]. Thus, as long as the ETG has at least k edge-disjoint paths from s to d, d will always be reachable from s.

Finding the number of edge-disjoint paths in an arbitrary acyclic digraph is NP-Complete [22], but in a unit-weight graph the problem reduces to computing the max-flow/min-cut. Because we are only concerned with the presence of paths, and not which paths are chosen under specific failures, we can safely convert the weight of all inter-device edges in the ETG to 1 and the weight of intra-device edges to ∞. We set the weight of intra-device edges to ∞, because we are only concerned with counting physical-link-disjoint paths, not device-disjoint paths, and a weight of ∞ allows multiple physical-link-disjoint paths to traverse the same device. We compute the max-flow/min-cut on the ETG with modified weights to identify the number of edge-disjoint paths. When the max-flow is $\geq k + 1$, the invariant is satisfied.

When the invariant is violated, we produce a counter-example set of edges that form a cut of size $\leq k$.

I3: Always isolated. For security or performance reasons, an operator may want to ensure that two disjoint traffic classes ($s_1 \rightarrow d_1$ and $s_2 \rightarrow d_2$; $s_1 \neq s_2$, $d_1 \neq d_2$) can never simultaneously traverse the same link. Thus, the preferred path for $s_1 \rightarrow d_1$ must never overlap with the preferred path for $s_2 \rightarrow d_2$ under any scenario. Such overlap is possible in some scenario if the ETG for $s_1 \rightarrow d_1$ has an edge in common with the ETG for $s_2 \rightarrow d_2$. An extreme scenario is where all links have failed except those used in paths that contain the common edge. The traffic isolation invariant is guaranteed to hold only if the ETGs for the two traffic classes do not have *any edges in common*, a property we can easily check.

ETGs constructed using our algorithm in §4.2 are complete, so they do not contain extra paths. However, the ETGs may still contain "dead-ends"—i.e., extra edges and vertices that are never part of any path from SRC to DST. Dead-ends arise because: (i) we do not add intra-device edges between vertices associated with different processes unless one process has a lower AD than the other or one process redistributes routes into the other, and (ii) we remove edges to account for ACLs and route filters (§4.2). When an ETG contains extra edges, we will inadvertently claim that two traffic classes are not always isolated, when in reality the extra edges are not part of any path from SRC to DST and hence have no bearing on traffic isolation. Thus, prior to checking this property, we recursively remove all vertices (excluding SRC and DST) whose in- or out-degree is 0; these vertices are dead-ends. When removing such vertices, we also remove their incident edges.

If the pruned ETGs have any edges in common, we return the set of common edges as a counter-example.

I4: Always traverse a waypoint. When a network includes middleboxes, such as firewalls, an operator may want to ensure that traffic always traverses some instance of the middlebox (i.e., a waypoint) under arbitrary failure scenarios. To verify this, we augment the ETG to include special vertices that represent waypoints. Then we remove all waypoint nodes from the ETG and check if there exists a path from *src-node* to *dst-node*. If such a path exists, then there is some path that may be taken by the traffic that does not traverse a waypoint; we return this path as a counterexample.

Other invariants. Other important security and availability invariants can also be verified by computing graph-level attributes on the ARC. For example, we can verify traffic "always traverses a chain of waypoints" by removing the vertices associated with one type of waypoint at a time, and checking if there exists a path from a vertex associated with one of the preceding waypoints in the chain to a vertex associated with one of the following waypoints in the chain. We can verify forwarding of particular traffic class is "always loop free" by checking that the ETG does not have a cycle containing a static route vertex and one or more vertices associated with processes in the same routing instances. We omit details for brevity.

5.2 Equivalence Testing

Invariant $I5$, equivalence, differs from the other invariants in three respects: (1) equivalence testing involves multiple ARCs; (2) it requires path-equivalent ARCs, because the actual paths taken in the network are the attributes under scrutiny; and (3) it is implemented by comparing ETGs, rather than computing graph characteristics of ETGs. However, prior to comparing the ETGs from different ARCs, we must make two transformations to the ETGs.

Convert process-based ETGs to interface-based ETGs. Modeling forwarding behavior at the level of routing processes (§4.2) prevents us from determining if *any* two control planes are *equivalent*, because the two control planes may use a different set of routing instances, causing their ETGs to contain a different set of vertices and edges. To address this issue, we convert our process-based ETGs into

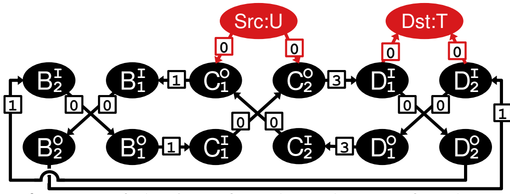


Figure 4: Part of the interface-based ARC for the example control plane in Figure 2a

interface-based ETGs, which depends only on the physical network topology, not the routing processes running atop it. As an example, Figure 4 shows the transformed ETG that corresponds to the upper-left ETG in Figure 2b. Our conversion process assumes each interface is used by at most one routing process to send/receive route advertisements; this is common practice in data center and enterprise networks, including those we study (§7.1).

In particular, we take the following steps:

1. Replace each process's *in* and *out* vertices with an *in* and *out* vertex for each *physical interface* over which the process sends/receives route advertisements. We identify interfaces on which a routing process operates based on overlap between the IP addresses assigned to interfaces and the networks (for OSPF and RIP) and neighbors (for eBGP) specified in the `router` stanzas in device configurations. For example, B_I and B_O in Figure 2b are replaced with vertices B_1^I, B_1^O, B_2^I, and B_2^O in Figure 4.
2. Replace the inter-device edges that used to connect the *out* vertex of a process P on one device to the *in* vertex of a process P' on another device with an edge connecting the *out* vertex of the interface over which P sends advertisements to P' to the *in* vertex of the interface over which P' receives advertisements from P. For example, the edge $B_O \rightarrow D_I$ in Figure 2b is replaced with the edge $B_2^O \rightarrow D_2^I$, because the second interface on router B is connected to the second interface on router D.[8]
3. Replace the intra-device edge E that used to connect a routing process's *in* and *out* vertices by a set of edges that connect the *in* vertex of each interface associated with the process to the *out* vertex of every interface associated with the process. The edge weight is the same as the edge weight that was assigned to E. Note that an edge is not created between the *in* and *out* vertices of the same interface, because a router will never send traffic out the same interface on which it arrived. For example, the edge $B_I \rightarrow B_O$ in Figure 2b is replaced with the edges $B_1^I \rightarrow B_2^O$ and $B_2^I \rightarrow B_1^O$ in Figure 4.
4. For each intra-device edge that connected the *in* vertex of a routing process P to the *out* vertex of another routing process P', create an intra-device edge from the *in* vertex associated with each of P's interfaces to the *out* vertices associated with P''s interfaces; again, the weight of these edges is the same as the weight of the original edge. (No such edges exist in Figure 2b.)

An ARC constructed in this manner represents a network's routing behavior with the same fidelity as the ARC described earlier, because it captures the exact same pathways between routers (no inter-device edges are added), and models at a fine granularity the same software pathways within routers.

Convert edge weights to canonical weights. There are infinitely many ARCs that differ only in the scale of their edge weights. All of these will produce the same data plane under all failure scenarios, and hence are equivalent. To ensure we can detect such equivalence, we must *reduce all edge weights to canonical weights*. In other words, we compute the lowest possible weight for every edge in every ETG in the ARC such that the relative order of all possible loop-free paths between *src-node* and *dst-node* in each ETG is the same as using the original weights. We can perform such a reduction using a linear program; details are included in our technical report [11].

After applying the above transformations, we can test the equivalence of two control planes by checking whether their ARCs have the same vertices, edges, and edge weights. This is facilitated by the fact that vertices are always named based on the device interfaces to which they pertain. Thus, vertices and their incident edges can be easily matched across ARCs.

6. IMPLEMENTATION

We implemented the ARC generation process described in §4 and the verification tasks described in §5.1 in Java. We use Batfish [9] to parse Cisco IOS configurations. From these, we extract traffic classes and generate ETGs. We use JGraphT [2] to apply common graph algorithms (Dijkstra's shortest path, max-flow/min-cut, etc.) to the generated ETGs and obtain the information required to verify a particular property. Our tool outputs the results of the requested verification for all of a network's flows. Our code is open source [1], so operators can apply it to their own networks.

7. EVALUATION

We now evaluate ARC along two different dimensions: (1) How efficiently can we represent real network control planes using ARC? (2) How quickly can we verify key invariants using ARC? How does this compare to state-of-the-art control plane verification tools (e.g., Batfish [9])?

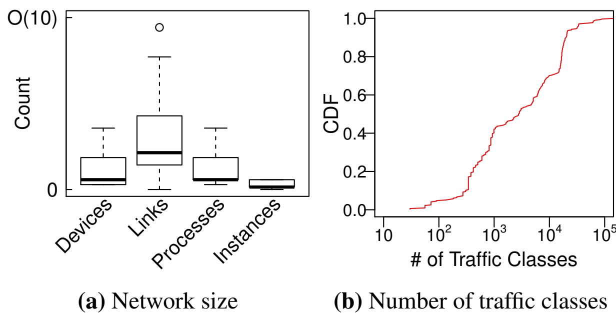
We generate ARCs and verify invariants using a machine with a quad-core Intel Xeon 2.8GHz CPU and 24GB of RAM.

7.1 Network Characteristics

In our evaluation we use configurations from 314 data center networks operated by a large OSP. These networks have between two and a few tens of routers connected using between one and several tens of physical links (Figure 5a).

Two-thirds of the networks have a single routing process on each device, while the remaining third have two processes per device on average (ignoring static routes). Similarly, two-thirds of the networks have a single routing instance, while the rest have a handful of instances (Figure 5a). As shown in Table 3, only two routing protocols are used—OSPF (37% of networks) and eBGP (all networks)—along with static routes (27% of networks). Only one network has OSPF processes that use multiple areas, and only 10 net-

[8]We assume interfaces corresponding to the inter-router links in Figure 2a are numbered clockwise for each router.



(a) Network size **(b)** Number of traffic classes

Figure 5: Scale of the OSP's networks

Protocols	% of Networks	Modifiers	% of Networks
OSPF	37.6%	ACLs	100.0%
single area	37.3%	Route filters	84.1%
eBGP	100.0%	Route redistribution	5.4%
no local prefs	96.8%	acylic	5.4%
Static routes	27.1%	costs & ADs align	5.4%

Table 3: Constructs used in the OSP's data center networks

works have eBGP processes that use local preference (in addition to AS path length) for computing routes. Route redistribution occurs in 5% of the networks; in all cases, the redistribution conforms to the constraints necessary to produce a path-equivalent ARC (§4.3.4).

The number of distinct traffic classes ranges from less than 100 to more than 100K (Figure 5b). There are less than 10K traffic classes in 69% of the networks, and less than 1000 in 41% of networks. As shown in Table 3, the OSP uses route filters (84% of networks) and ACLs (all networks) to selectively block certain traffic classes.

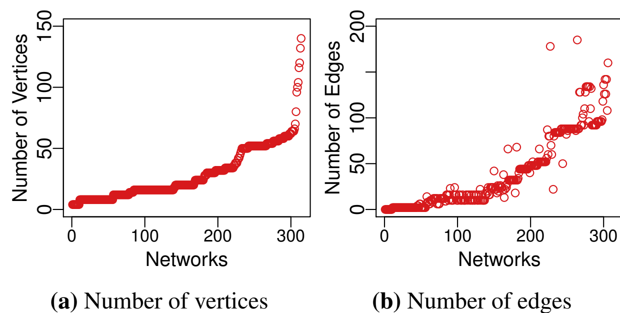
By comparing Tables 2 and 3, it is clear we can generate a pathset-equivalent ARC for *all* of the OSP's networks, and a path-equivalent ARC for 97% of the networks (those with one OSPF area and no BGP local preferences).

For comparison, we examined the routing protocols and features used in our university network. The same protocols used in the OSP's networks (OSPF, eBGP, and static routes) are used in our university, along with RIP; ACLs, route filters, and acyclic route redistribution are also used. BGP local preferences are not used, but there are multiple OSPF areas, so we can generate a pathset-equivalent but not path-equivalent ARC for our university network.

7.2 ARC Efficiency

We now examine how efficiently we can represent real network control planes using ARC. We consider both the time to generate the ARC and the ARC's size, and we show how this relates to the size and complexity of a network.

Time. Figure 6 shows the time required to generate the ARC, included edge weights, for each of the OSP's networks. ARC generation takes less than 5s for 78% of the networks, and at most 11.8s across all the networks we study. The majority of the time (85% on average) is spent parsing network configurations; this time is roughly correlated with the number of devices in the network (Pearson correlation co-efficient of 0.58). The remaining time is dedicated to constructing the ETGs; this time is roughly correlated with the



(a) Number of vertices **(b)** Number of edges

Figure 7: Size of the ETGs for the OSP's networks; networks are sorted by number of vertices in the ETG

number of traffic classes in the network (Pearson correlation co-efficient of 0.62), because the ARC contains an ETG for every traffic class.

Size. Figure 7 characterizes the size of the generated ETG for each network. We observe that ETGs are relatively compact: 45% (45%) of the ETGs have fewer than 20 vertices (edges) and 74% (70%) have fewer than 50. By design, the number of vertices is directly correlated with the number of routing processes (including static routes) in the network for which the ETG is generated.

As mentioned above, the number of ETGs required for each network is a function of the number of traffic classes (Figure 5b). Although this seems substantial, 78% of networks' ETGs take less than 100MB of space when stored as serialized Java objects; more efficient encoding schemes could significantly reduce this. Furthermore, we show in the next section that exhaustively verifying key invariants for all of a network's traffic classes takes less than a second for most networks.

7.3 Verification Efficiency

We next examine how efficiently we can verify the invariants discussed in §5.1. We compare the speed of ARC-based control plane verification against Batfish [9], a state-of-the-art network configuration analysis tool.[9]

Figure 8 shows the time required to verify invariants $I1$, $I2$, $I3$, and $I5$ for all traffic classes (or pairs of traffic classes) for each of the OSP's networks.[10] When checking equivalence, we compare a network's control plane against itself.

We observe that invariant $I1$ can be checked for arbitrary link failures and all traffic classes in less than 500ms for 97% of the OSP's networks, and 62% of the networks can be checked in less than 100ms. The networks that take the longest to verify have the most traffic classes. The time per traffic class ranges from 8μs to 347μs (median 21μs).

The time required to verify invariant $I2$ is slightly higher, because computing max-flow/min-cut is more complex than checking if two nodes reside in separate connected components. However, for 99% of the OSP's networks, this property can be checked in less than 1s, and 54% of networks can

[9] We do not compare against other verifiers [16, 15, 20, 17], because they only consider the current network data plane.

[10] We do not check invariant $I4$, because we do not know where middleboxes reside in the OSP's networks.

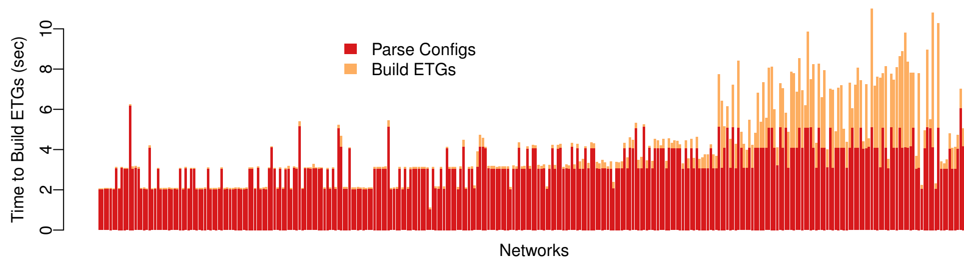
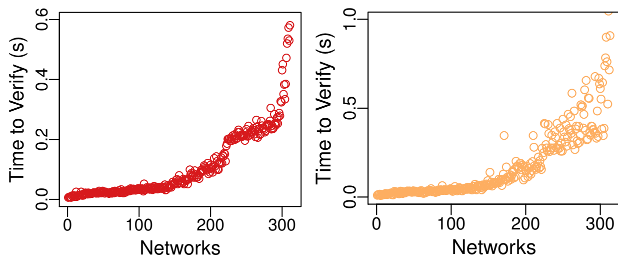
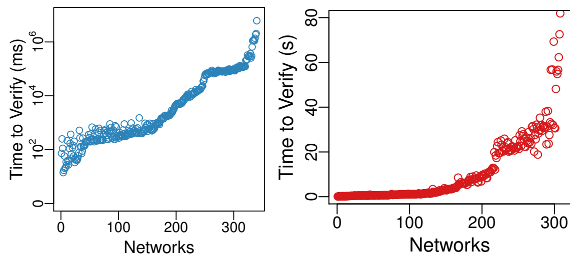


Figure 6: Time required to generate ARC for the OSP's networks; networks are sorted by number of traffic classes



(a) I1: Always blocked, using ARC

(b) I2: Always reachable with $< k$ failures, using ARC



(c) I3: Always isolated, using ARC

(d) I5: Equivalent, using ARC

Figure 8: Time required to check key invariants for all traffic classes (or pairs of traffic classes) using ARC; networks are sorted by number of traffic classes.

be checked in less than 100ms. In the worst case, verification takes 1.13s. Again, the networks that take the longest to verify are those with the most traffic classes. The time per traffic class ranges from 7μs to 467μs (median 32μs). Note that the time required for checking this property is independent of the value of k.

Invariant $I3$ takes substantially longer to verify, because we check all pairs of traffic classes, as opposed to each individual traffic class. It takes about 1.7 hours to check all pairs of traffic classes in the network with the largest number of traffic classes ($> 100K$), but this property can be checked in less than 1 minute for all pairs of traffic classes in 73% of networks. In practice, only a subset of traffic classes in a network require isolation, so the number of traffic class pairs that need to be checked is substantially smaller.

While $I3$ takes the longest to verify overall, equivalence checking ($I5$) takes the most time per traffic class: 0.9ms to 4.8ms (median 1.3ms). Most of this time is spent generating and solving the linear program used to compute canonical weights (§5.2). The former requires generating all possible paths between SRC and DST, which is significantly more computationally expensive than the graph algorithms used

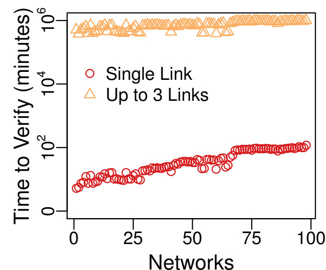


Figure 9: Time required by Batfish to verify (lack of) reachability across a limited set of failure scenarios; networks are sorted by number of links.

to verify $I1$–$I3$. Nonetheless, we can check control plane equivalence in less than one minute for 98% of networks and less than one second for 28% of networks.

Comparison with Batfish. To put our performance results in perspective, we ran Batfish [9] on the device configurations from one-third of the OSP's networks. We chose networks of varying size and complexity. We ran Batfish's "failure consistency" checker, which verifies that each traffic class is consistently blocked or allowed when any one of the network's links fails. This is similar to verifying invariants $I1$ and $I2$ ($k = 2$) using ARC, except verification with ARC covers all link failures, not just single link failures.

Figure 9 shows the time required for Batfish to check the reachability of all traffic classes under a limited set of link failures. We observe that the time taken by Batfish to check all single link failure scenarios is at least *three orders of magnitude larger* than the time required for ARC-based verification to check *all* link failure scenarios. If we were to run Batfish for all scenarios with up to 3 link failures, the time would further increase by up to *five orders of magnitude* making Batfish impractical to use in this case.

The time required by Batfish to verify invariants across a set of link failure scenarios is a function of: (1) the number of scenarios, and (2) the time required to generate the data plane and verify the invariant for each scenario. In our experiments, Batfish takes between 48s and 131s (median 92s) to generate the data plane and verify the invariant for each link failure scenario. With ARC, the time required to verify invariants across arbitrary link failure scenarios is a function of: (1) the number of traffic classes, and (2) the time required to generate the ETG and verify the invariant for each traffic class. As mentioned above, the median verification time per ETG for invariant $I1$ is 21μs and the median ETG build time is 98μs. Thus, a network with a single link would

need to have over 773K traffic classes in order for ARC to be less efficient than Batfish.

Identifying problems. In addition to evaluating verification performance, we used the output from our tool to identify possible configuration errors. For each traffic class, we compared the results of $I1$ and $I3$ against the behavior of the network in the absence of link failures. We assumed the control plane was configured incorrectly if traffic was blocked (or isolated) in the failure-free scenario but not always blocked (or isolated). We did not find any such cases in the networks we studied; this is likely due to the fact that the organization whose networks we studied already employs other verification tools. However, we were able to detect such errors when we intentionally introduced bugs into configurations modeled after real networks; see examples in our repository [1].

8. CONCLUSION

We described ARC, a new high level abstraction for routing control planes that enables fast verification of key properties under arbitrary failure scenarios. ARC achieves this by avoiding data plane generation, which is made possible by the nature of common verification tasks and control plane designs observed in networks today. On real data center configurations, ARC offers orders of magnitude faster verification performance than existing verifiers.

This paper lays the groundwork for accelerating the verification and repair of network control planes, and a variety of important problems remain open. These include: modeling additional protocols and features, such as those commonly used in service provider networks (e.g., iBGP) and for traffic engineering (e.g., MPLS TE [4]); generating ARCs that over- or under-approximate pathset-equivalence to handle non-deterministic protocols (e.g., RSVP); producing ARCs for software-defined network (SDN) control planes, including hybrid control planes that use both SDN and traditional distributed routing protocols; and automatically generating *minimal* configuration repairs (e.g., changing a minimal set of devices or adding a minimal number of lines of configuration) when invariant violations are detected. We plan to address these open problems in our future research.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Brad Karp for their insightful comments. This work is supported by the Wisconsin Institute on Software-defined Datacenters of Madison and National Science Foundation grants CNS-1302041, CNS-1330308, and CNS-1345249.

10. REFERENCES

[1] Abstract representation for control planes. http://bitbucket.org/uw-madison-networking-research/arc.

[2] JGraphT. http://jgrapht.org.

[3] R. Aharoni and E. Berger. Menger's theorem for infinite graphs. *Inventiones mathematicae*, 2008.

[4] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus. Requirements for traffic engineering over MPLS. RFC 2702, RFC Editor, September 1999.

[5] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *NSDI*, 2009.

[6] T. Benson, A. Akella, and A. Shaikh. Demystifying configuration challenges and trade-offs in network-based ISP services. In *SIGCOMM*, 2011.

[7] Cisco Systems. BGP best path selection algorithm. http://cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/13753-25.html.

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

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

[10] E. W. Fulp. Optimization of network firewall policies using directed acyclic graphs. In *IEEE Internet Mgmt Conf*, 2005.

[11] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. Technical report, University of Wisconsin-Madison, 2016.

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

[13] S. Hares and D. Katz. Administrative domains and routing domains: A model for routing in the internet. RFC 1136, RFC Editor, Dec 1989.

[14] Juniper Networks. Understanding BGP path selection. http://juniper.net/documentation/en_US/junos12.1/topics/reference/general/routing-ptotocols-address-representation.html.

[15] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

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

[17] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[18] P. Lapukhov, A. Premji, and J. Mitchell. Use of BGP for routing in large-scale data centers. Internet-Draft draft-ietf-rtgwg-bgp-routing-large-dc-07, IETF Secretariat, Aug 2015.

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

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

[21] 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 *SIGCOMM*, 2004.

[22] A. Slivkins. Parameterized tractability of edge-disjoint paths on directed acyclic graphs. *SIAM J. Discret. Math.*, 24(1):146–157, Feb. 2010.

[23] Y. Sverdlik. Microsoft: misconfigured network device led to azure outage. http://datacenterdynamics.com/servers-storage/microsoft-misconfigured-network-device-led-to-azure-outage/68312.fullarticle, July 2012.

[24] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. FIREMAN: a toolkit for FIREwall Modeling and ANalysis. In *IEEE SP*, 2006.

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