Automatically Repairing Network Control Planes Using an Abstract Representation

Aaron Gember-Jacobson
Colgate University
agemberjacobson@colgate.edu

Aditya Akella
University of Wisconsin-Madison
akella@cs.wisc.edu

Ratul Mahajan
Intentionet
ratul@ratul.org

Hongqiang Harry Liu
Microsoft Research
harliu@microsoft.com

ABSTRACT

The forwarding behavior of computer networks is governed by the configuration of distributed routing protocols and access filters—collectively known as the network control plane. Unfortunately, control plane configurations are often buggy, causing networks to violate important policies: e.g., specific traffic classes (defined in terms of source and destination endpoints) should always be able to reach their destination, or always traverse a waypoint. Manually repairing these configurations is daunting because of their inter-twined nature across routers, traffic classes, and policies.

Inspired by recent work in automatic program repair, we introduce CPR, a system that automatically computes correct, minimal repairs for network control planes. CPR casts configuration repair as a MaxSMT problem whose constraints are based on a digraph-based representation of a control plane's semantics. Crucially, this representation must capture the dependencies between traffic classes arising from the cross-traffic-class nature of control plane constructs. The MaxSMT formulation must account for these dependencies whilst also accounting for all policies and preferring repairs that minimize the size (e.g., number of lines) of the configuration changes. Using configurations from 96 data center networks, we show that CPR produces repairs in less than a minute for 98% of the networks, and these repairs requiring changing the same or fewer lines of configuration than hand-written repairs in 79% of cases.

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.

SOSP '17, Shanghai, China
© 2017 ACM. 978-1-4503-5085-3/17/10. . . $15.00
DOI: 10.1145/3132747.3132753

CCS CONCEPTS

•Networks → Network management;

1 INTRODUCTION

Computer networks often rely on distributed routing protocols to determine how data is forwarded through the network. Each router in the network runs one or more routing protocols to exchange information with its neighbors and compute the best (e.g., least cost) paths to various destinations. These routing protocol instances, along with access filters and inter-protocol communication mechanisms, collectively form a network's *control plane*.

The control plane must be carefully configured by network operators to satisfy various *policies*: e.g., specific classes of traffic (i.e, that between particular source and destination hosts) should always reach its destination, or always be blocked. Policies must hold even when links or routers fail. This task is difficult due to the low-level nature of the abstraction exposed by router configuration languages, the diversity of policies a network must satisfy (e.g., blocking some traffic classes while allowing others), and the desire for policy compliance under arbitrary link or router failures. Consequently, router configurations are highly prone to bugs that lead to outages [50, 34].

Over the last decade the research community has developed many tools to verify a network's policy compliance. Some tools [26, 27, 28, 35, 43] analyze a snapshot of the network's forwarding state to check whether the network conforms to all policies in its current state. Other tools take the network's router configurations as input [7, 16, 17, 18, 20] and analyze the control plane's behavior for policy compliance under various failure scenarios. However, both classes of tools stop at finding policy violations, and do not help operators *repair* the buggy configurations.

Researchers have only recently begun to develop tools that automatically repair broken data [25] and control planes [47],

but they focus on software-defined networks (SDNs). Currently, no tools are capable of repairing distributed control planes, which continue to be used in the majority of networks.

Repairing non-compliant distributed control plane configurations can be extremely challenging. One source of difficulty is the intertwined nature of configurations, across routers, policies, and traffic classes. A repair may involve changes to multiple routers (e.g., modifying the costs of multiple links to ensure that traffic favors or disfavors certain paths); a repair that fixes one policy violation may trigger another violation for the same traffic class (e.g., blocking traffic from traversing a link will also reduce its degree of fault tolerance); and a repair that leads to policy compliance for one traffic class may violate policies for another traffic class (e.g., removing a routing protocol adjacency to prevent a traffic class from using a link will prevent other traffic classes from using the link too). Another source of difficulty is that not all valid repairs are equally desirable. Depending on the context, a more desirable repair is one that minimizes the number of routers or the number of total lines that need to be updated. Today, operators must manually reason about all such dependencies and factors when determining the repair strategy.

The same challenges arise when a network operator wants to change the policies a network satisfies or incrementally grow or shrink the network. For example, to add new routers or end-hosts to the network, an operator must manually determine how to *"repair"* the network's configurations to ensure the new hosts are reachable.

We develop *CPR* (for Control Plane Repair), the first tool to automatically repair distributed control plane configurations. It takes as input a network's existing configurations and desired security and availability policies (i.e., a specification) and outputs configuration patches. After applying the patches, the network is guaranteed to compute policy-compliant paths for all traffic classes under arbitrary failures.

As a starting point, CPR uses ARC, an Abstract Representation for Control planes [20], which compactly captures the result of interactions amongst routers using a collection of digraphs, with one digraph per traffic class. However, because it was developed with a focus on verification—where the central question is to check if the network satisfies a given policy for a given traffic class—ARC and its underlying algorithms do not capture interactions between traffic classes, nor do they provide a way to convert ARC back into (minimal) routing configurations.

CPR uses several techniques to address these limitations and achieve its goal of correct, minimal repairs. First, we extend ARC to Hierarchical ARC (HARC) which captures dependencies between traffic classes. Unlike ARC, HARC has multiple types of digraphs and, instead of being independent, digraphs are related based on configuration constructs that impact multiple traffic classes. Second, to capture dependencies between policies, we encode policy compliance using SMT (satisfiability modulo theories) constraints. These constraints encode the semantics of the graphs that form a network's HARC, and the solution yields which edges must be added or removed from the graphs to satisfy all policies for all traffic classes. Third, to compute minimal repairs, we add soft constraints to our SMT encoding, thus turning it into a MaxSMT problem. These constraints lead the solver to prefer minimal repairs among all valid repairs. Finally, we develop methods to scale our MaxSMT formulation to large networks and convert the resulting (policy-compliant) HARCs to router configurations.

We have implemented CPR in Java and made our code publicly available [1]. Our experiments using configuration snapshots from 96 data center networks, each with a median of 8 routers and 1K policies, show that CPR computes repairs in less than a minute for 98% of the networks, and these repairs require the same or fewer configuration changes than hand-written repairs in 79% of the cases. Consequently, CPR significantly advances the state of the art in network management.

2 BACKGROUND AND CHALLENGES

In this section, we provide an overview of the control planes in typical data center and local area networks. We then discuss the challenges in automatically repairing control planes to conform to a set of policies.

2.1 Network Control Planes

A network's control plane computes the paths taken by traffic to reach its destination. Every router in a network runs distributed control logic that performs these computations and produces a set of forwarding rules that the router uses to forward traffic along the appropriate path. The control logic includes implementations of many different protocols (e.g., RIP [36], OSPF [38], BGP [41]), each of which uses different algorithms to exchange route advertisements with neighboring routers and compute paths.

A configuration file written by a network operator or automatic generator [8, 44] in a vendor-specific configuration language (e.g., Cisco IOS [2]) instructs the router which protocol(s) it should use, what destinations it should advertise to its neighbor(s), how it should select between multiple possible paths (e.g., link costs), etc. Figure 1 shows an example configuration. The configuration is broken into blocks of statements, i.e., *stanzas*, that are related to a particular protocol, physical interface, etc.

Every instance of a routing protocol configured on a router (e.g., lines 11–15 in Figure 1) is called a *routing process*. A routing process only exchanges route advertisements with

```
1  hostname C
2  interface Ethernet0/1
3    description Link-to-A
4    ip address 10.0.2.3/24
5  interface Ethernet0/2
6    description Link-to-B
7    ip address 10.0.3.3/24
8  interface Ethernet0/3
9    description Subnet-T
10   ip address 10.20.0.0/16
11 router ospf 10
12   redistribute connected
13   passive interface Ethernet0/1
14   passive interface Ethernet0/3
15   network 10.0.0.0/16
```

Figure 1: Example router configuration

processes of the same type running on neighboring routers it is authorized to speak with (according to the configuration). A pair of routing processes on neighboring devices form a *routing adjacency*. A router may also internally share computed paths between its processes, even if they are running different protocols, if the configuration enables *route redistribution* between them.

Routers are also equipped with mechanisms to filter (i.e., block) specific traffic. An *access control list* (ACL) is composed of a set of permit and deny statements that explicitly permit or deny certain classes of traffic (i.e., traffic between particular sources and destinations). This filtering is applied when traffic enters or exits a specific physical interface on the router. Filtering can also happen during the path computation phase: a *route filter* can prevent a routing process from advertising a path or destination to another routing process.[1]

2.2 Challenges in Automated Repair

Repairing control plane configurations to satisfy a set of reachability-related policies introduces three challenges; two of these pertain to preventing undesirable side-effects and the third pertains to optimality or, equivalently, minimality, of the repair. We illustrate these challenges using the example control plane shown in Figure 2a.

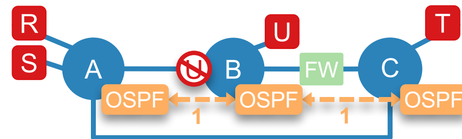
#1: Multiple policies. Assume the control plane must satisfy four different policies:

EP1: Under all possible failures, traffic from S to U is always blocked;

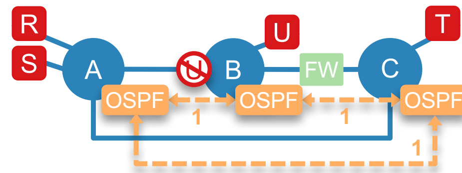
EP2: Under all possible failures, traffic from S to T always traverses a firewall;

EP3: S can reach T as long as there is at most one link failure;

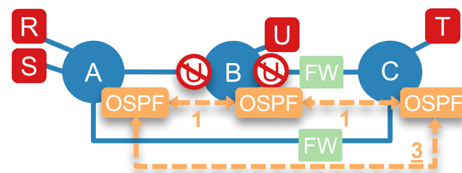
[1] In some contexts, e.g., within an OSPF area, route filters only prevent a route from being used on the router containing the route filter, rather than filtering advertisements to other routers.



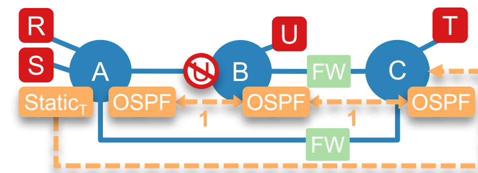
(a) Physical network and original control plane: Blue circles are routers, red squares are subnets, and green rectangles are firewalls. Orange rectangles are OSPF processes and red no-entry symbols are ACLs blocking traffic destined for U. Solid blue lines are physical links and dashed orange lines are routing adjacencies. Router C's configuration is shown in Figure 1.



(b) Add routing adjacency between A and D



(c) Increase cost of A–C and add ACL blocking U



(d) Add static route to A with lower preference than OSPF

Figure 2: An example control plane and repair attempts

EP4: In the absence of failures, traffic from R to T uses the path $A \rightarrow B \rightarrow C$.

Currently, the control plane satisfies three of the four policies: *EP1* because the only path for S to reach U is $A \rightarrow B$, which includes an ACL blocking traffic destined for U; *EP2* because the only path for S to reach T is $A \rightarrow B \rightarrow C$, which includes a firewall; and *EP4* because the only path for R to reach T is $A \rightarrow B \rightarrow C$. The control plane violates *EP3*, because failure of the A–B or B–C links renders T unreachable.

According to Menger's Theorem [4], the maximum number of edge-disjoint paths between two vertices (e.g., S and T) equals the number of edges whose removal separates those vertices. Thus, to satisfy *EP3*, the network must contain at least two edge-disjoint paths between S and T. A simple repair is to create a routing adjacency between the OSPF processes on routers A and C (Figure 2b). In particular, we remove line 13 from router C's configuration (Figure 1). This will allow the OSPF process on router A to use the path

$A \rightarrow C$ or the path $A \rightarrow B \rightarrow C$ to reach T, thereby ensuring S can still reach T even if a link fails.

In the absence of failures, OSPF will prefer the newly available shorter path $A \rightarrow C$. Because this path does not contain a firewall, our repair now causes a violation of *EP2*. To satisfy both *EP2* and *EP3*, we must add a firewall on the A–C link.[2] This illustrates our first challenge: *repairing a control plane to simultaneously satisfy multiple policies*.

#2: Cross-traffic-class effects. By default, OSPF (and BGP) routing adjacencies apply to all traffic classes—i.e., all pairs of source and destination subnets. Consequently, adding the routing adjacency between A and C impacts not only the traffic from S to T, but also traffic from R to T and from S to U. The newly available path to T ($A \rightarrow C$) will be used for traffic originating from both S and R; this violates *EP4*. Furthermore, If the A–B link fails, the OSPF process on router A can now compute an alternative path to reach U: $A \rightarrow C \rightarrow B$. This path does not contain any ACLs blocking traffic destined for U, so *EP1* is violated. Thus, the second challenge we must address is: *accounting for cross-traffic-class effects*.

#3: Complexity of configuration changes. We can avoid violating *EP4* by changing router A's configuration to increase the cost of the link to C, and we can avoid violating *EP1* by adding an ACL on router B's second interface that blocks traffic destined for U (Figure 2c). We have now modified the configurations of all three routers, with one line removed from C and one line added to each of A and B, as well as added a firewall. This raises the question: *is there a simpler repair?*

We initially increased the number of edge-disjoint paths between S and T by creating an OSPF routing adjacency between router's A and C (Figure 2b). However, we can achieve the same effect by configuring a static route on A that directs traffic for T to router C (Figure 2d); this eliminates the configuration change on C in favor of a configuration change on A. Because the static route only applies to traffic destined for T, traffic destined for U can only traverse the originally available path $A \rightarrow B$; this eliminates the need to add an ACL on B. Finally, to ensure the static route does not take precedence over a path computed by OSPF and violate *EP4*, we increase the administrative distance (i.e., cost) of the static route so the path computed by OSPF is preferred. In total, this alternative repair only requires adding two lines of configuration to router A, plus a firewall on the A–C link. This illustrates the third challenge we must address: *ensuring that the repair is minimal*. Minimality may be measured along several different dimensions, e.g., number of routers changed, number of lines changed, etc.

[2] We assume the network uses virtual network functions and tunnels [49] to allow waypoints to be added along arbitrary links.

3 CONTROL PLANE REPAIR (CPR)

The goal of CPR is to *automatically* repair a set of router configurations such that a network satisfies a collection of reachability-related policies under arbitrary link failures. CPR takes as input existing router configurations and a set of operator-defined policies and produces configuration patches to correct errors and account for policy changes. Crucially, CPR avoids cross-policy and cross-traffic class effects and computes minimal repairs.

Instead of directly manipulating configuration syntax, as is frequently done in program synthesis and repair [29, 32, 46], CPR uses an abstract representation of the control plane's semantics. We do this because router configuration languages are low-level [11] and the configuration of an individual router has limited bearing on the end-to-end treatment of the traffic. These factors make the space of possible changes intractable. We could constrain the search space by considering only changes to the configurations of routers and protocols included in counter-examples produced by control plane verifiers [16, 18, 20]; or we could limit repairs to removing and replicating existing lines of configuration [32, 46]. However, the search space is still large, and worse, viable repairs may not exist in these constrained spaces.

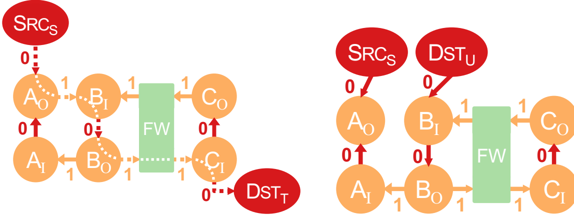
Several encodings of control plane semantics have been developed recently [7, 8, 16, 18, 20]. We extend our previously developed abstract representation for control planes (ARC) [20], which abstracts away the low-level details of individual routing protocols and messages, and concisely captures protocols' eventual impact on the network's forwarding behavior under arbitrary failures.

With our semantic-based approach, repairs in CPR boil down to: (1) converting the input configurations to their semantic encoding; (2) repairing the encoding to satisfy all operator-defined policies; and (3) translating the repaired encoding into router configurations. The next three sections describe how we accomplish each task.

4 MODELING CONTROL PLANE SEMANTICS

Because ARC [20] was designed to *verify* network properties of individual traffic classes, it abstracts away details about how the network handles related traffic classes—e.g., those with a common destination. This makes it difficult to use ARC to compute repairs without cross-traffic-class interference and to ensure the repairs can be realized using the destination-based primitives available in routers.

In this section, we first provide an overview of ARC and discuss in more detail its limitations with respect to control plane repair. Then we describe a new *hierarchical abstract*



(a) ETG for $S \rightsquigarrow T$ traffic class (b) ETG for $S \rightsquigarrow U$ traffic class

Figure 3: ETGs for the example control plane in Figure 2a: Orange circles are process vertices (I=incoming, O=outgoing); red ovals are SRC and DST vertices; green rectangles are firewalls. Orange (light) lines are inter-device edges; red (dark) lines are intra-device edges.

representation for control planes (HARC) that tracks important information about the network's handling of related traffic classes and is amenable to automatic repair.

4.1 Abstract Representation for Control Planes (ARC)

Our previously developed abstract representation for control planes (ARC) [20] models a network's forwarding behavior under arbitrary failures using a collection of directed graphs called *extended topology graphs* (ETG). There is one ETG per traffic class which models the behavior of the network's routing protocols, and the interactions among them, for the traffic class. Vertices correspond to routing processes; there is one incoming (I) and one outgoing (O) vertex per process. Directed edges represent the possible flow of data traffic enabled by the exchange of route advertisements between the connected processes. For example, Figure 3a shows the ETG for the $S \rightsquigarrow T$ traffic class for the control plane in Figure 2a: there are two vertices for the OSPF process on each router (e.g., A_I and A_O) and edges representing the routing adjacencies (e.g., $A_O \rightarrow B_I$) and intra-device communication (e.g., $A_I \rightarrow A_O$).

The algorithm for constructing an ETG from a network's control plane configurations is summarized in Algorithm 1. By construction ETGs are *pathset-equivalent*: i.e., an ETG contains a particular path between the source and destination endpoints iff that path is used in the real network under some combination of failures (including no failures) [20]. For some networks, ETGs can also model the *exact* path used by the real network under specific failures—a property known as *path-equivalence* [20]. To achieve this, edge weights are set such that, after removing all edges corresponding to failed links, the shortest path through the ETG is the exact path taken in the real network. For example, the inter-device edge weights in Figure 3a match the OSPF link costs in Figure 2a. Such modeling is only possible for networks that use routing protocols, route redistribution, and ACLs in restricted ways.

Algorithm 1: Process for constructing an ETG for traffic class *tc* from control plane configurations

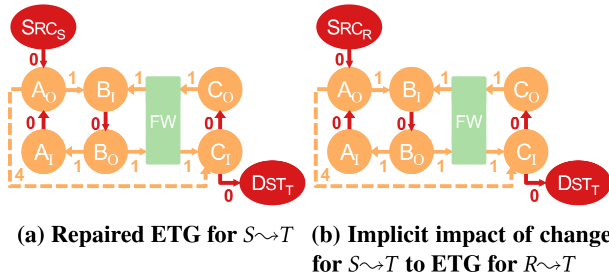
```
1  foreach RoutingProcess proc do
2      Add vertices proc_I and proc_O
3      Add edge proc_I → proc_O                         // Intra-dev
4      if proc blocks routes to tc.dst then
5          continue
6      foreach RoutingProcess proc′ on proc.device do
7          if proc′ redistributes routes from proc
8            ∧ proc′ does not block routes to tc.dst then
9              Add edge proc′_I → proc_O                // Intra-dev
10     foreach Interface intf used by proc do
11         if ∃ phy link intf → intf′
12           ∧ ∃ RoutingProcess proc′ uses intf′
13           ∧ proc’ does not block routes to tc.dst then
14             Add edge proc_O → proc′_I                // Inter-dev
15             if intf.acl blocks tc ∨ intf′.acl blocks tc then
16                 Remove edge proc_O → proc′_I
17 foreach Device dev do
18     if ∃ phy link tc.src → dev then
19         foreach RoutingProcess proc on dev do
20             Add edge SRC → proc_O                    // Source
21     if ∃ phy link dev → tc.dst then
22         foreach RoutingProcess proc on dev do
23             Add edge proc_I → DST                    // Destination
```

Policy class	ETG characteristic
PC1: Traffic from SRC to DST is always blocked	SRC and DST are in separate components
PC2: Traffic from SRC to DST always traverses a waypoint	After removing edges with waypoints, SRC and DST are in separate components
PC3: SRC can always reach DST when there are $< k$ link failures	Max-flow from SRC to DST in a unit-weight ETG is $\geq k$
PC4: Traffic from SRC to DST uses path P in the absence of failures	Shortest path from SRC to DST is P

Table 1: Class of policy and the characteristics an ETG must possess to ensure compliance with it

Using ARC to verify a control plane is policy-compliant (under arbitrary failures) boils down to checking simple characteristics of the constituent ETGs. Table 1 lists a few common classes of reachability-related policies along with the characteristics an ETG must possess to ensure the policy holds for the corresponding traffic class.[3] For example, suppose we want to check whether the control plane depicted in Figure 2a satisfies *EP3* (S can reach T as long as there is at most one link failure). As discussed in §2.2, we can verify this policy by computing the max-flow of a unit-weight

[3]Other characteristics can be used to verify other classes of policies [21].



(a) Repaired ETG for $S \rightsquigarrow T$ **(b) Implicit impact of change for $S \rightsquigarrow T$ to ETG for $R \rightsquigarrow T$**

Figure 4: Repaired ETGs for the example control plane in Figure 2a

version of the ETG for $S \rightsquigarrow T$. We observe the max-flow of the ETG is one (dashed path in Figure 3a), so the policy is violated.

Computing a repair with ARC entails adding, removing, and adjusting the weights of edges in ETGs to obtain the desired (quantity of) paths between the SRC and DST vertices in each ETG.[4] For example, we can add the edge $A_\text{O} \to C_\text{I}$ to the ETG for $S \rightsquigarrow T$ (dashed line in Figure 4a) to satisfy *EP3*.

4.2 Limitations of ARC

Using a separate ETG for each traffic class works well for verification, because individual policies can be verified in isolation. However, for network repair we must consider the fact that distributed routing protocols compute paths on a per-destination (not per-traffic-class) basis. Consequently, a single configuration may impact multiple traffic classes. For example, we showed in §2.2 that adding a static route to T on router A causes all traffic destined for T to be sent to the specified next hop (C), irrespective of the traffic's source (S or R). This configuration change effectively adds the edge $A_\text{O} \to C_\text{I}$ to the ETGs for both $S \rightsquigarrow T$ and $R \rightsquigarrow T$ (dashed lines in Figure 4), even though we only intended to add the edge to the ETG for $S \rightsquigarrow T$ (Figure 4a) to fix the violation of *EP3*.

Although ARC includes (or excludes) the same edge in multiple ETGs when a control plane construct impacts multiple traffic classes (Algorithm 1), ARC does not explicitly encode the fact that multiple ETGs contain the same edge due to a single control plane construct. For example, the ETGs for both $S \rightsquigarrow T$ and $S \rightsquigarrow U$ (Figure 3a and 3b, respectively) for the control plane in Figure 2a include the edges $B_\text{O} \to C_\text{I}$ and $C_\text{O} \to B_\text{I}$, because the OSPF routing adjacency established in C's configuration (lines 11–15 in Figure 1) and B's configuration (not shown) applies to all traffic classes. However, the ETGs are completely disjoint, with no indication that they contain a common edge derived from a single construct (the routing adjacency).

Thus, when repairing ETGs of an ARC, a problematic situation may arise: a common edge derived from a single control plane construct that applies to multiple traffic classes can be removed from one ETG without removing the edge from all other ETGs that contain it. Similarly, an edge can be added to one ETG without adding it to others. This results in an ARC whose modeled behavior cannot be implemented in practice.

4.3 Hierarchical ARC (HARC)

Based on the observations above, we extend ARC to track common edges across ETGs due to specific control plane constructs. We refer to this abstraction as *hierarchical ARC* (HARC). HARC maintains the core building block of ARC—an ETG—but creates multiple types of ETGs to track common edges resulting from control plane constructs impacting traffic classes at different granularities.

Since control plane constructs apply to a specific traffic class (e.g., an ACL), a specific destination (e.g., a static route or route filter), or all traffic classes (e.g., an OSPF or BGP routing adjacency), we construct three different types of ETGs: *traffic class ETGs* (tcETGs), *destination ETGs* (dETGs), and an *all-traffic-classes ETG* (aETG). tcETGs are synonymous with the ETGs included in the original ARC, and model the network's forwarding behavior under arbitrary failures for a specific traffic class. dETGs model the network's forwarding behavior for a specific destination subnet (and all possible source subnets); dETGs take into account static routes and route filters, which apply to specific destinations, but dETGs ignore ACLs, which apply to specific source-destination pairs. Finally, the aETG models the forwarding behavior resulting from OSPF and BGP routing adjacencies.

Due to the control plane constructs modeled in each type of ETG, there exists a "hierarchy" among the ETGs. Edges that exist in tcETGs must exist in dETGs, because traditional control planes employ destination-based routing—i.e., no control plane construct can enable reachability for only a single traffic class.[5] Similarly, edges that exist in dETGs must either exist in the aETG (because routing adjacencies apply to all destinations) or be associated with static routes.

This hierarchy enables us to constrain the space of potential HARC repairs to those that can be realized using available control plane constructs (§5). Additionally, the hierarchy implicitly encodes which type of control plane construct causes each edge to be present (or absent) in an ETG, which is required to translate HARC modifications to configuration changes (§6). For example, if an edge is present in a dETG but not the aETG, we know the edge must be associated with a static route, because the dETG accounts for static routes but the aETG does not. Similarly, if an edge is absent in a tcETG

[4] An ETG may also be repaired by adding and removing vertices, but, for simplicity of exposition, we restrict the problem to modifying edges.

[5] Programmable (e.g., OpenFlow) switches and some traditional routers support source-based routing, but such features are beyond our scope.

but not the corresponding dETG, we know the edge must be excluded due to an ACL, because the tcETG accounts for ACLs but the dETG does not.

5 MINIMALLY REPAIRING HARC

Given a network's HARC, computing a repair entails adding, removing, and adjusting the weights of edges in ETGs[6] to obtain the desired (quantity of) paths between the SRC and DST vertices in each tcETG.

One way to modify ETGs is to using polynomial-time graph algorithms similar to those used in ARC [20]. For example, to repair a tcETG to satisfy *PC1*, we can compute the tcETG's min-cut and remove all edges in the min-cut. Similarly, for *PC2*, we can temporarily remove all waypoint vertices, compute the min-cut, and either add waypoints on or remove all edges in the min-cut. For *PC3*, we can construct a tcETG containing all possible edges, compute the max-flow on a unit-weight version of this tcETG, and, for k paths in the max-flow, add the edges in the paths to the original tcETG (and dETG). Lastly, for *PC4*, we can solve the inverse shortest paths problem [10].

Unfortunately, applying graph algorithms to the general repair problem is non-trivial when we consider the challenges from §2.2. For example, we can show that finding a minimal repair that ensures *PC4* holds for multiple tcETGs is NP-Hard (omitted for brevity). Thus, we look for a more general approach to compute suitable repairs.

Inspired by recent work in program repair [37, 14, 40, 23] we cast ETG repair as a constraint solving problem and use a Satisfiability Modulo Theory (SMT) solver to efficiently search for a solution. In this section, we start with a basic encoding of ETG repairs using SMT to prevent cross-policy and cross-traffic-class effects. We extend it to achieve minimality in §5.2.

5.1 Repair as Constraint Solving

At the heart of our SMT formulation is a set of boolean variables representing the edges that may exist in each ETG. The variable $edge_{tc}^{v_1-v_2}$ represents the edge from v_1 to v_2 in the tcETG for traffic class tc; similarly the variables $edge_{dst}^{v_1-v_2}$ and $edge_{all}^{v_1-v_2}$ represent the edges from v_1 to v_2 in the dETG for destination dst and the aETG, respectively. Constraints on the paths present in each tcETG are derived from the set of provided policies and defined in terms of the edge variables. A satisfying solution is an assignment of values to the edge variables such that all tcETGs possess the requisite characteristics, and the tcETGs, dETGs, and aETG represent a valid HARC.

[6] We can only add edges for which there is a corresponding physical link or intra-router communication channel.

// Constraints for PC1

1 $\neg path_{tc}^{\text{SRC}-\text{DST}}$

2 $\forall edge_{tc}^{v_1-v_2} : edge_{tc}^{v_1-v_2} \Rightarrow path_{tc}^{v_1-v_2}$

3 $\forall edge_{tc}^{v_1-v_2}, path_{tc}^{v_2-v_3} : edge_{tc}^{v_1-v_2} \wedge path_{tc}^{v_2-v_3} \Rightarrow path_{tc}^{v_1-v_3}$

// Constraints for PC2

4 $\neg nwpath_{tc}^{\text{SRC}-\text{DST}}$

5 $\forall edge_{tc}^{v_1-v_2} : edge_{tc}^{v_1-v_2} \wedge \neg wedge^{v_1-v_2} \Rightarrow nwpath_{tc}^{v_1-v_2}$

6 $\forall edge_{tc}^{v_1-v_2}, nwpath_{tc}^{v_2-v_3} :$
$edge_{tc}^{v_1-v_2} \wedge \neg wedge^{v_1-v_2} \wedge nwpath_{tc}^{v_2-v_3} \Rightarrow nwpath_{tc}^{v_1-v_3}$

// Constraints for PC3*, repeat for* $1 \leq k \leq K$

7 $\forall edgek_{tc}^{v_1-v_2} : edgek_{tc}^{v_1-v_2} \Rightarrow edge_{tc}^{v_1-v_2}$

8 $\exists edgek_{tc}^{\text{SRC}-v_1} : edgek_{tc}^{\text{SRC}-v_1}$

9 $\exists edgek_{tc}^{v_1-\text{DST}} : edgek_{tc}^{v_1-\text{DST}}$

10 $\forall edgek_{tc}^{v_2-v_3} : v_2 \neq \text{SRC} \wedge edgek_{tc}^{v_2-v_3} \Rightarrow$
$\exists edgek_{tc}^{v_1-v_2} : edgek_{tc}^{v_1-v_2}$

11 $\forall edgek_{tc}^{v_1-v_2} : v_2 \neq \text{DST} \wedge edgek_{tc}^{v_1-v_2} \Rightarrow$
$\exists edgek_{tc}^{v_2-v_3} : edgek_{tc}^{v_2-v_3} \wedge \nexists edgek_{tc}^{v_2-v_4} : edgek_{tc}^{v_2-v_4}$

12 $\forall$ inter-device $edgek_{tc}^{v_1-v_2}$: $edgek_{tc}^{v_1-v_2} \Rightarrow$
$\neg(edge1_{tc}^{v_1-v_2} \vee \ldots$ skip $edgek_{tc}^{v_1-v_2} \ldots \vee edgeK_{tc}^{v_1-v_2})$

// Constraints for PC4

13 $\forall cost^{v_1-v_2} : cost^{v_1-v_2} > 0$

14 $scost_{tc}^{\text{SRC}} = 0$

15 $pred_{tc}^{\text{SRC}} = \text{SRC}$

16 $\forall edge_{tc}^{v_1-v_2} : edge_{tc}^{v_1-v_2} \wedge (\nexists edge_{tc}^{v_3-v_2} :$
$edge_{tc}^{v_3-v_2} \wedge scost_{tc}^{v_3} + cost^{v_3-v_2} < scost_{tc}^{v_1} + cost^{v_1-v_2}) \Rightarrow$
$scost_{tc}^{v_2} = scost_{tc}^{v_1} + cost^{v_1-v_2} \wedge pred_{tc}^{v_2} = v_1$

17 $\forall edge_{tc}^{v_1-v_2} \in P : edge_{tc}^{v_1-v_2} \wedge pred_{tc}^{v_2} = v_1$

// Constraints for HARC

18 $\forall edge_{tc}^{v_1-v_2} : edge_{tc}^{v_1-v_2} \Rightarrow edge_{tc.dst}^{v_1-v_2}$

19 $\forall edge_{dst}^{v_1-v_2}$ (excluding static routes): $edge_{dst}^{v_1-v_2} \Rightarrow edge_{all}^{v_1-v_2}$

Figure 5: SMT constraints for finding repairs

Policy constraints. Figure 5 shows the constraints we use for each of the four classes of policies listed in Table 1. In the simplest case, *PC1*, we do not want any path to exist between SRC and DST (constraint 1). The boolean variable $path_{tc}^{v_1-v_2}$ represents a path from v_1 to v_2 in the tcETG for tc. Constraints 2 and 3 inductively define when a path exists.

For *PC2*, we do not want any paths from SRC to DST that do not traverse a waypoint (constraint 4). The boolean variable $nwpath_{tc}^{v_1-v_2}$ represents a path from v_1 to v_2 that does not traverse a waypoint. An inter-device edge contains a waypoint if the corresponding physical link has a waypoint on-path, and an intra-device edge contains a waypoint if traffic is shunted through a waypoint connected to the router as the traffic passes through the router. The boolean variable $wedge^{v_1-v_2}$ denotes for all ETGs whether a particular edge contains a waypoint. For example, $wedge^{C_O-B_I}$ is true for the control plane shown in Figure 2a, because all

ETGs in the HARC (two of which are shown in Figure 3) contain a waypoint on the edge $C_O \rightarrow B_I$. Similar to our constraints for *PC1*, constraints 5 and 6 inductively define when a path without a waypoint exists. If a network operator is unable or unwilling to add waypoints to the network, then we must include additional constraints that ensure all edges without waypoints in the original setup remain in that state (i.e., $\neg wedge^{v_1-v_2}$).

For *PC3*, we require a minimum number (K) of link-disjoint paths, such that up to $K-1$ physical link failures can be tolerated. Consequently, constraints 7–12 are designed to enumerate link-disjoint paths in the ETG. We create K boolean variables for each edge that could exist in the tcETG for tc: $edge1_{tc}^{v_1-v_2}...edgeK_{tc}^{v_1-v_2}$. An edge must exist in the tcETG if it is part of a link-disjoint path (constraint 7). Each link-disjoint path must start at SRC and end at DST (constraints 8 and 9), and each ETG edge in the middle of the path must have an ETG edge that precedes it and exactly one ETG edge that follows it (constraints 10 and 11). Finally, constraint 12 states that an ETG edge (specifically an inter-device ETG edge) that exists in one link-disjoint path cannot exist in any other link-disjoint path. Note that a link-disjoint path (i.e., an edge-disjoint path in the physical network topology) does not directly correspond to an edge-disjoint path in the tcETG, because a single router (i.e., vertex) in the physical topology is represented by multiple vertices in the ETG. For example, the tcETG in Figure 4a contains two link-disjoint paths from S to T ($A \rightarrow C$ and $A \rightarrow B \rightarrow C$), but the corresponding paths in the ETG are not edge-disjoint (they share edges $\text{SRC}_S \rightarrow A_\text{O}$ and $C_\text{I} \rightarrow \text{DST}_C$).

For *PC4*, we require a path-equivalent ETG (§4.1) and an assignment of edge weights such that the shortest path from SRC to DST is P. Consequently, constraints 13–17 are modeled on Dijkstra's shortest path algorithm. The integer variable $cost^{v_1-v_2}$ represents the cost of the edge $v_1 \rightarrow v_2$ across all ETGs in the HARC. Edge costs must be the same across all ETGs, because routing protocols such as OSPF do not allow costs to be customized on a per-traffic-class or per-destination basis. Constraint 13 forces the cost of edges to be positive. The integer variable $scost_{tc}^{v_1}$ represents the cost of the shortest path from SRC to v_1 in the tcETG for tc, and the variable $pred_{tc}^{v_1}$ stores the vertex that immediately precedes v_1 in the shortest path. In other words, the *pred* variables encode the shortest path tree from SRC to all other vertices. Constraints 14 and 15 define the base case for the shortest paths: the cost from SRC to itself is 0, and the vertex preceding SRC on the shortest path to SRC is itself.

The assignment of edge weights is governed by constraint 16. This constraint inductively defines the shortest path from SRC to each vertex (v_2). The first part of the constraint ensures an edge to v_2 ($edge_{tc}^{v_1-v_2}$) exists in the tcETG and checks that there is no other edge to v_2 ($edge_{tc}^{v_3-v_2}$) that results in a shorter path from SRC to v_2. If both of those conditions hold, then we know the cost of the shortest path from SRC to v_2 ($scost_{tc}^{v_2}$) is the cost of the shortest path from SRC to v_1 ($scost_{tc}^{v_1}$) plus the cost of the edge from v_1 to v_2 ($cost^{v_1-v_2}$), and v_1 immediately precedes v_2 on the shortest path from SRC to v_2 (i.e., $pred_{tc}^{v_2} = v_1$). The SMT solver will automatically iterate over combinations of edges and edge weights to find values that satisfy this constraint. Finally, constraint 17 states that each edge in the desired path P must exist in the tcETG, and the edges in P must be in the shortest path from SRC to DST in the order they appear in P.

Other reachability policies can be accommodated using similar constraints. For example, isolation between two traffic classes ($tc1$ and $tc2$) can be encoded using the constraint $\forall edge_{tc1}^{v_1-v_2} : edge_{tc1}^{v_1-v_2} \Rightarrow \neg edge_{tc2}^{v_1-v_2}$, and vice versa.

HARC constraints. In addition to the policy constraints, we need a few constraints that ensure the resulting HARC is well-formed (§4.3). Constraint 18 enforces the requisite hierarchy between the tcETGs and their dETGs, while constraint 19 enforces the hierarchy between the dETGs and aETG. Without these constraints, the SMT solver may produce repairs that cannot be implemented in the actual control plane: e.g., a solution that includes the edge $A_\text{O} \rightarrow C_\text{I}$ in the tcETG for $S \rightsquigarrow T$ (dashed line in Figure 4a) but excludes the edge from the dETG for T (not shown) is invalid, because traditional routing protocols do not allow routing adjacencies to be enabled for only a single traffic class.

5.2 Minimizing ETG modifications

While all satisfying solutions to our SMT formulation (Figure 5) represent a HARC that is policy-compliant, the solution computed by the solver may not result in minimal changes. To help compute minimal repairs, we transform our SMT problem into a MaxSMT problem. A MaxSMT problem consists of a set of hard constraints that *must* be satisfied and a set of soft constraints that should be *maximally* satisfied. In CPR, the hard constraints come from our original SMT formulation; they ensure the solution is correct. The soft constraints are derived from the original HARC produced from the input configurations; they ensure the resulting HARC is as similar as possible to the original. In this section, we present the intuition and definition of a set of soft constraints that seek to minimize the number of lines of configuration changed. Similar sets of constraints can be constructed for other objectives such as minimal number of devices changed; we omit details for brevity.

Relating HARC modifications to configuration changes. When constructing dETGs, we account for all control plane constructs considered in the construction of the aETG (e.g., OSPF adjacencies) plus some additional constructs (e.g.,

static routes and route filters). Similarly, when constructing tcETGs, we account for all control plane constructs considered in the construction of the corresponding dETG plus some additional constructs (e.g., ACLs). If there are no additional constructs that apply to a specific destination or traffic class, then the dETG (or tcETG) will have the same structure as the aETG (or dETG). If the dETG (or tcETG) and aETG (or dETG) contain different sets of edges, then for each edge that exists in one but not the other, there must be a control plane construct that causes the deviation.

Consequently, if we repair a dETG by adding or removing an edge without adding or removing the edge from the aETG, then there must be a control plane construct we add to the configuration (e.g., a static route or route filter) that implements the modeled deviation. The same applies when we add and remove edges to a tcETG without doing the same in the corresponding dETG—although an edge in a tcETG must exist in the dETG for the HARC to be valid (§4.3). This implies that each deviation between a dETG and the aETG, or a tcETG and its corresponding dETG, requires a single configuration change. For example, if i edges present in a dETG are removed from a corresponding tcETG, then we need to add i (applications of) ACLs for the traffic class associated with the tcETG. Similarly, if we add an edge to a dETG without adding the edge to i of the tcETGs, then we need to: (1) change the configurations to reflect the addition of the edge in the dETG—e.g., remove a route filter—and (2) make i configuration changes—e.g., add i deny statements to an ACL—to prevent the i traffic classes from using the newly available path.

In summary, the number of new ways in which a child ETG deviates from its parent ETG as a result of repairs, plus the number of new ways in which a child ETG now aligns with its parent ETG, equals the number of configuration changes required to implement the behavior modeled by the repaired HARC. Since the aETG does not have a parent, any change to the aETG is considered a new deviation or alignment.

Soft constraints. Since each new deviation or alignment between child and parent ETGs requires a configuration change, and our goal is to minimize the number of lines of configuration changed, our soft constraints seek to minimize the number of new deviations and alignments between child and parent ETGs. Alternatively, we can cast this problem as maximizing the number of edges for which child and parent ETGs continue to align or deviate as they do in the original HARC constructed from the input configurations.

Each edge in the child ETG that continues to align with or deviate from the parent provides one unit of *utility*, because it avoids one configuration change. Thus, if we create a soft constraint for every edge in a child ETG that requires the edge to either align with or deviate from its parent, the number of soft constraints will equal the total utility of the solution. Table 2 lists the precise soft constraints we use for each possible edge in each tcETG for all combinations of ETGs the edge currently exists in.

Edge in original			Soft constraint for the edge at each level		
aETG	dETG	tcETG	tcETG	dETG	aETG
✓	✓	✓	$edge_{tc} \leftrightarrow edge_{dst}$	$edge_{dst} \leftrightarrow edge_{all}$	$edge_{all}$
✓	✓		$\neg edge_{tc}$	$edge_{dst} \leftrightarrow edge_{all}$	$edge_{all}$
✓		✓	Invalid HARC		
✓			$edge_{tc} \leftrightarrow edge_{dst}$	$\neg edge_{dst}$	$edge_{all}$
	✓	✓	$edge_{tc} \leftrightarrow edge_{dst}$	$edge_{dst}$	$\neg edge_{all}$
	✓		$\neg edge_{tc}$	$edge_{dst}$	$\neg edge_{all}$
		✓	Invalid HARC		
			$edge_{tc} \leftrightarrow edge_{dst}$	$edge_{dst} \leftrightarrow edge_{all}$	$\neg edge_{all}$

Table 2: Soft constraints for finding minimal repairs

5.3 Scalability

While our MaxSMT formulation can identify a set of correct, minimal HARC modifications, solving the problem for even moderately sized networks is time consuming. For example, computing a repair for a network with 45 routers and 120 *PC3* policies (one per traffic class) requires 40 seconds (Figure 8b); doubling the number of policies (and traffic classes) more than quadruples the solving time. This raises an important question: *can we find correct HARC modifications faster if we tolerate a repair that is close to, but not exactly, minimal?*

To answer this question, we leverage our observation above that doubling the number of traffic classes more than quadruples the solving time. While we cannot ignore some of a network's traffic classes, we can formulate multiple MaxSMT problems, each for a different subset of the network's traffic classes. At one extreme, we can formulate a single MaxSMT problem that covers all traffic classes. On the other, we can formulate a separate MaxSMT problem for each destination and solve them in parallel: e.g., one problem for $R \rightsquigarrow U$, $S \rightsquigarrow U$, and $T \rightsquigarrow U$ in Figure 2a, and a separate problem for $R \rightsquigarrow T$, $S \rightsquigarrow T$, and $U \rightsquigarrow T$. In the absence of *PC4* policies, the solutions will not conflict, because routing can be customized on a per-destination basis using route filters and static routes. *PC4* policies pose a challenge, because link costs cannot be customized on a per-destination basis. However, conflicts can be avoided by dividing traffic classes such that only one of the problems involves *PC4* policies and associated edge weight computations. We cannot formulate MaxSMT problems at finer granularity (e.g., per traffic-class), because we risk producing a HARC that violates the hierarchy requirements—see the end of §5.1 for an example.

In §8, we show that for real network configurations solving a separate MaxSMT problem for each destination results in an order of magnitude reduction in overall solving time without any decrease in the minimality of repairs.

ETG	Edge	Configuration change
tcETG	inter-device	remove tc from ACL
tcETG	intra-device	*invalid modification*
dETG	inter-device	remove dst from route filter (if edge exists in repaired aETG) OR add static route for dst
dETG	intra-device	remove dst from route filter
aETG	inter-device	enable routing
aETG	intra-device	enable route redistribution

Table 3: Translations for edge additions; the inverse changes apply to edge removals

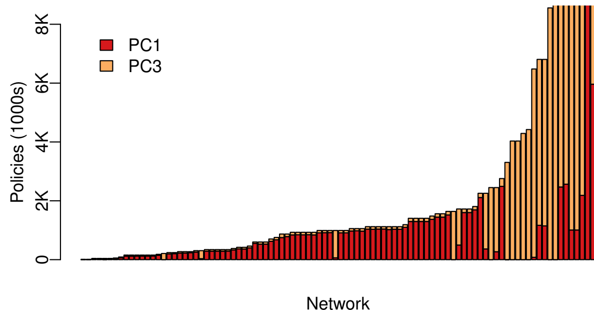


Figure 6: Policy mix in real data center networks

6 TRANSLATING HARC REPAIRS TO CONFIGURATION CHANGES

The final step in CPR is to translate modifications made to the HARC into actual configuration changes. We can determine how the HARC was modified by comparing the original HARC generated from the broken configurations to the repaired HARC represented by the solution to our MaxSMT problem. As discussed in §5.2, every edge we add or remove from an ETG requires a corresponding configuration change.

Determining the type of control plane construct to add, remove, or modify is simplified by the fact that each type of ETG in the HARC considers a slightly broader set of control plane constructs than its parent (§4.3). Consequently, if an edge is added or removed from an ETG but not changed in the parent ETG, then we need to modify one of the control plane constructs that are considered in the construction of the child ETG but not the parent ETG. For example, if we remove an inter-device edge from a tcETG but not from its corresponding dETG, then we need to change an ACL. However, if we remove an inter-device edge from a tcETG *and* its associated dETG, then we need to change a static route or route filter. Table 3 lists the type of configuration change that needs to be made for each type of ETG and edge.

After determining the type of change that needs to be made, we locate the precise stanza to change based on: (1) the traffic class (tc) or destination (dst) associated with the modified ETG, and (2) the process(es) and (for inter-device edges) interfaces associated with the modified edge. We then traverse the substanzas to locate the appropriate line to modify, remove, or insert at. For example, to remove tc from an ACL, we locate the ACL that is applied to the edge's source interface, and we check if the ACL contains a deny statement for tc. If we locate a deny statement that applies only to tc, then we remove it; otherwise, we add a permit statement for tc at the beginning of the ACL. Similarly, to enable routing between two processes, we locate the router stanzas for the processes on each device, and we add a `network` or `neighbor` stanza, for OSPF and BGP processes respectively, that includes the interfaces associated with the edge.

7 IMPLEMENTATION

Our implementation of CPR is written in Java (≈10K LOC). We use Batfish [18] to parse router configurations written in vendor-specific languages (e.g., Cisco IOS) and modify the ARC implementation [20] to generate HARCs based on the parsed configurations. We use the Z3 theorem prover's [3] Java API to encode and solve our MaxSMT formulation. We have made our implementation of CPR open source [1], so that network operators and researchers can leverage it to repair their network and expand its capabilities.

8 EVALUATION

We evaluate CPR along three dimensions: time to compute repairs, minimality of repairs, and CPR-generated repairs versus hand-written repairs.

We use configuration snapshots from 96 real data center networks as well as synthetic configurations for vanilla fat-tree topologies [5]. The real data center configurations come from the same dataset we used in prior works [20, 22]; we filter the dataset to only include networks that have at least one policy change.[7] The resulting set of 96 networks have between 2 and 24 routers[8] (median is 8) and up to 82K traffic classes (median is ≈1K). The dataset does not include a list of desired policies, so we infer the policies a network satisfies in a particular snapshot using ARC's verification algorithms [20]. We only consider policies of type *PC1* and *PC3*, because we do not know the location of waypoints and cannot infer which paths an operator prefers. Figure 6 shows the mix of policies for each network; the networks are stored by the total number of policies. The majority of the networks have a policy for every traffic class; no traffic class has multiple policies, because a traffic class cannot both be always blocked (*PC1*) and always reachable (*PC3*).

[7]Many networks in the original dataset have only non-routing-related changes (e.g. password changes or updates to Simple Network Management Protocol (SNMP) settings).

[8]The networks also contain dozens of switches. We exclude them because they operate at a lower layer of the network stack that ARC does not capture.

To compute repairs, we feed the inferred policies and the configurations from the *preceding* snapshot into CPR.

We also generate configurations for fat-tree topologies of varying port counts [5]. All routers run OSPF. We include ACLs on all core switches to block or permit certain traffic classes, such that hosts in different pods are always blocked (*PC1*) or always reachable (*PC3*), respectively. We also include waypoints on half of the core–aggregation links, and block traffic on the remainder, such that hosts in different pods always traverse a waypoint (*PC2*). Finally, we assign lower costs to the links between the first core switch and the connected aggregation switches to induce primary paths (*PC4*). We break the configurations by inverting the ACLs and assigning lower costs to the links of a different core switch. We have publicly released the code for generating the configurations [1].

All experiments are conducted on servers with 10-core Intel E5 2.4GHz CPUs and 128GB of RAM. We set a time limit of 8 hours on all experiments.

8.1 Time to Compute HARC Repairs

We first evaluate the time required to compute HARC repairs.

Real data centers. We compute HARC repairs for each of the 96 real data center networks using both a single MaxSMT problem that encompasses all traffic classes (*maxsmt-all-tcs*) and multiple MaxSMT problems that each encompass one destination (*maxsmt-per-dst*). Figure 7 shows the time required to repair each network at each problem granularity; we order the networks by the number of policies. We observe that *maxsmt-all-tcs* takes more than an hour in 58% of the networks, and for 30% of the networks does not even finish in the time limit we set (8 hours). In contrast, computing repairs separately for each group of traffic classes with the same destination (*maxsmt-per-dst*), reduces the computation time by one to two orders of magnitude. Using this approach, repairs for 86% of the networks were computed in less than a minute and 99% completed in less than an hour.

Several factors contribute to this substantial decrease. First, each MaxSMT problem has fewer boolean variables and constraints, because it encompasses fewer traffic classes and policies. A simpler problem is faster to solve. Second, we can ignore destinations for which there are no policy violations, thereby reducing the number of problems we need to solve. Finally, using multiple MaxSMT problems provides an opportunity for parallelism: running 10 MaxSMT problems in parallel, we can compute repairs for 98% of the networks in less than a minute and all complete in less than an hour. Thus, our approach for improving scalability (§5.3) offers substantial performance improvements.

The differences in computation time across networks are due to several factors. The time required to compute repairs

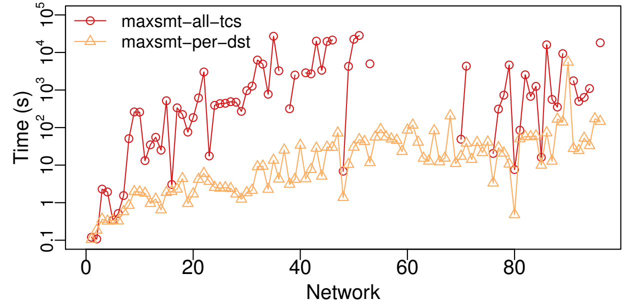


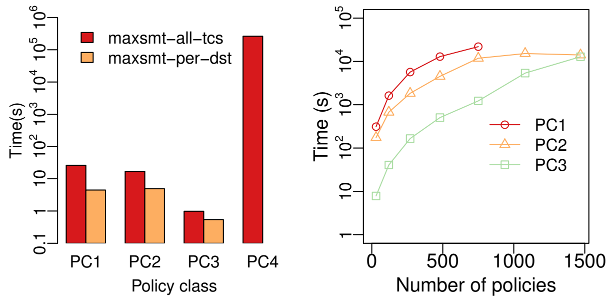
Figure 7: Time required to compute repairs for real data center networks

using *maxsmt-all-tcs* is most strongly correlated with the number of policies that need to be satisfied (Pearson correlation coefficient of 0.49), because additional constraints must be added for each policy (§5.1). There is also a weak correlation (0.2) with network size. With *maxsmt-per-dst*, the time to compute repairs is most strongly correlated with the number of policies that are violated in the original configurations (correlation coefficient of 0.34), because we only need to formulate and solve a MaxSMT problem for destinations for which there is at least one violated policy.

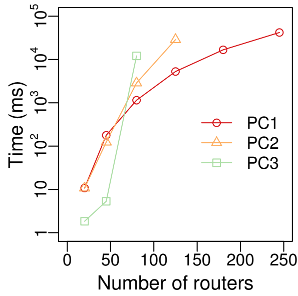
Synthetic fat-tree configurations. To better understand which factors affect CPR's performance, we conduct three different experiments using our synthetic fat-tree configurations.

In the first experiment, we vary the classes of policies the network must satisfy, while keeping the network size (a 4-port fat-tree with 20 routers) and number of policies (12) constant. Figure 8a shows the time required to compute repairs for each type of policy using *maxsmt-all-tcs* and *maxsmt-per-dst*; we exclude *maxsmt-per-dst* results for *PC4*, because we cannot formulate multiple MaxSMT problems that involve this policy class (§5.3). We observe that always reachable (*PC3*) policies are the fastest to repair while primary path (*PC4*) policies are the slowest. Primary path policies are substantially more complex to repair, because the possible values of the cost variables ($cost^{v_1-v_2}$) are virtually limitless, whereas the constraints for the other policies only involve boolean variables. We also again observe that using *maxsmt-per-dst* results in an order-of-magnitude improvement in repair times compared to *maxsmt-all-tcs*.

Next, we vary the number of policies the network must satisfy, while keeping the type of policies and network size (a 6-port fat-tree with 45 routers) constant. Figure 8b shows the time required to compute repairs for three of the policy classes using *maxsmt-per-dst*; we exclude *PC4* for the reason noted above. We see an exponential increase in repair times as the number of policies increases. This stems from the fact that each new policy adds additional boolean variables to



(a) Policy class **(b) Number of policies**



(c) Network size

Figure 8: Impact of different factors on the time required to compute repairs

the problem (e.g., $path_{tc}^{v_1-v_2}$ for *PC1*, $nwpath_{tc}^{v_1-v_2}$ for *PC2*, and $edgek_{tc}^{v_1-v_2}$ for *PC3*). Each new variable doubles the space of possible solutions (although the number of solutions tried by the solver increases by much less due to the way it navigates the solution space). For *PC1* and *PC2* the increase tapers off as as we approach the maximum number of policies the network can support, which is dictated by the maximum number of hosts the fat-tree can support. This stems from the fact that the number of allowed paths dwindles as more traffic classes must be blocked or routed through a waypoint, thereby giving the solver fewer viable options to explore.

Finally, we vary the size of the network, while keeping the type and number of policies (30) constant. Figure 8c shows the time required to compute repairs for three of the policy classes using *maxsmt-per-dst*; we again exclude *PC4* for the reason noted above. For *PC1* and *PC2*, we again see an exponential increase in repair times as the network size increases, because each new routing process and physical link adds additional edge possibilities ($edge_{tc}^{v_1-v_2}$). For *PC3*, the increase is more drastic, because K additional edge variables are added to the problem for each new physical link.

8.2 Minimality of Repairs

Next, we evaluate the minimality of repairs computed by CPR under the two different granularities of MaxSMT formulations. From Figure 9, we observe that computing repairs

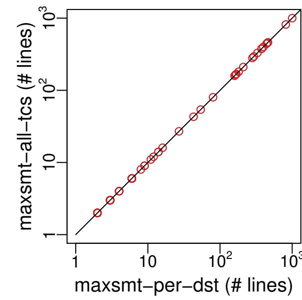


Figure 9: Number of lines of configuration changed using multiple versus a single MaxSMT problem

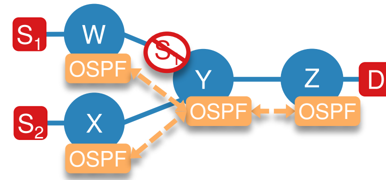


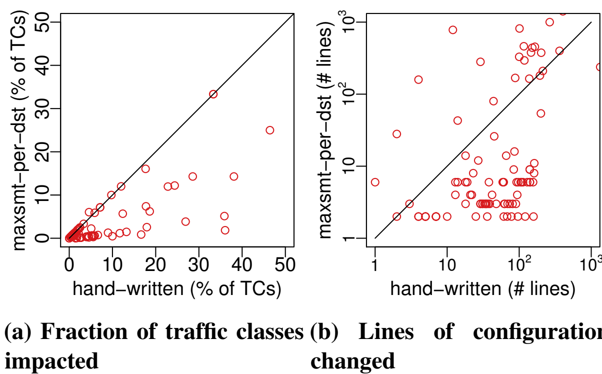
Figure 10: Example network that satisfies "$S_1 \rightarrow D$ is always blocked" and violates "$S_2 \rightarrow D$ is always blocked"

using *maxsmt-per-dst* always results in the same number of lines of configuration changed compared to computing repairs over all traffic classes (*maxsmt-all-tcs*). Thus, in practice, solving multiple smaller MaxSMT problems to boost performance (§5.3) does not come at a cost of reduced minimality.

8.3 Comparison with Hand-written Repairs

Finally, we compare repairs produced by CPR with repairs hand-written by network operators. We extract the latter by "diff'ing" successive configuration snapshots. Some changes made by operators have no bearing on routing or forwarding—e.g., updates to router login credentials or Simple Network Management Protocol (SNMP) settings; we ignore all non-routing and non-forwarding related differences between snapshots. We compare the CPR-produced and hand-written repairs along three dimensions: number of traffic classes impacted, number of lines of configuration changed, and time required. For brevity, we only present results for *maxsmt-per-dst*; the results for *maxsmt-all-tcs* are similar.

Traffic classes impacted. Figure 11a shows the fraction of traffic classes (TCs) impacted by CPR-produced versus hand-written repairs; each point corresponds to a single pair of successive configuration snapshots. In 60% of the cases, less than 5% of a network's traffic classes are impacted by repairs. However, hand-written repairs impact *more* traffic classes than CPR-produced repairs in 53% of the total cases (and the same number of traffic classes in the remaining 47% of



(a) Fraction of traffic classes impacted **(b) Lines of configuration changed**

Figure 11: CPR-produced versus hand-written repairs

cases), despite both types of repairs realizing the same set of policies. This stems from CPR's minimality goals: CPR avoids changing an ETG even if a change to the ETG would have no bearing on policy compliance for the corresponding traffic class. Consequently, CPR minimizes changes more than necessary. For example, consider the unrepaired network in Figure 10 that satisfies the policy "$S_1 \rightarrow D$ is always blocked" but violates the policy "$S_2 \rightarrow D$ is always blocked." An operator may choose to disable the routing adjacency between Y and Z, which impacts both traffic classes, whereas CPR may choose to add an ACL on Z to block all incoming traffic from S_2, which only impacts the $S_2 \rightarrow D$ traffic class. Both repairs require changing the same number of lines of configuration, and both result in a policy-compliant network, but the operator's repair impacts twice as many traffic classes as the CPR's repair. This implies CPR could be more lax in its minimality objectives, potentially allowing CPR to find repairs faster.

Number of lines changed. Figure 11b shows the number of lines of configuration changed in CPR-produced versus hand-written repairs. We observe that CPR-produced repairs require changing the same or fewer number of lines of configuration in 79% of the cases. This trend partially stems from our previous observation that CPR-produced repairs impact fewer traffic classes than hand-written repairs. More importantly, it indicates that CPR is able to identify simpler repairs than human operators.

The cases where CPR-produced repairs require changing more lines than hand-written repairs are the result of unoptimized ACL rules. CPR currently translates each HARC modification in isolation, without considering whether translations could be merged—e.g., a single ACL rule could encompass changes for multiple traffic classes. Improving CPR's translation process, e.g., using firewall rule optimization algorithms [19], is part of our planned future work.

Time to develop a repair. It is difficult to quantify how long it takes a human to repair a network, because detecting violations, localizing the problem, and designing a fix are often intermingled, and operators' actions to complete these steps are rarely logged [22]. Additionally, a substantial fraction of the repair time is often spent on change management processes, which involve peer reviewing repairs before they are deployed. Consequently, we use the number of traffic classes impacted and the number of lines of configuration changed in a hand-written repair as proxy for the time taken for a human to repair the network, because we expect larger repairs take longer to write.

Interestingly, there is no correlation between the time taken by CPR to compute a repair and the number of traffic classes impacted or number of lines of configuration changed in a hand-written repair: both have a Pearson correlation coefficient of -0.02. Thus, CPR may be faster than humans at computing some repairs and slower for others. A detailed study of when CPR beats humans at computing repairs is an interesting topic we plan to explore in future work.

9 DISCUSSION

In this section, we discuss a few of CPR's qualitative limitations and directions for future work.

Minimality versus simplicity of repairs. CPR's objective is to find repairs that minimize the number of changes made to the network. This objective is driven by the fact that more complex networks are more difficult to manage [9]. However, a minimal repair may not always be the least complex way to repair the network. For example, enabling route redistribution requires only a single line of configuration, but route redistribution is notorious for making networks significantly more difficult for an operator to manage [31]. In the future, we plan to explore how we can produce repairs that are both minimal and easy for network operators to understand.

Protocols and features modeled by HARC. ARC, and by extension HARC, only models the basic features of the most common routing protocols (RIP, OSPF, and eBGP), along with access control lists, route filters, static routes, and acyclic route redistribution [20]. ARC does not model more advanced protocol features (e.g., OSPF areas and BGP local preference), other common routing protocols (e.g., iBGP, IS-IS, EIGRP, and LDP), or layer-2 protocols (e.g., spanning tree). Consequently, CPR is restricted to repairing networks that use the common protocols and features supported by ARC. However, any improvements made to ARC will directly benefit CPR and allow it repair a larger range of networks. Moreover, as long as the semantics of ARC remain unchanged, no changes are required to CPR to take advantage of future improvements in ARC.

10 RELATED WORK

CPR's vision is similar to that of Wu et al. [47] and Hojjat et al. [25]. However, they focus on repairing control applications and forwarding rules, respectively, for software-defined networks (SDNs), rather than configurations for distributed control planes. Repairing distributed control planes is more challenging, because repairs are constrained by the route computation and selection algorithms supported by standard protocols (e.g., OSPF computes least-cost paths using Dijsktra's algorithm). Furthermore, Wu et al. base repairs on observed traffic [47], and Hojjat et al. base repairs on the network's current failure state [25], so new problems may arise if new traffic patterns emerge or the set of available links changes. In contrast, CPR bases repairs on a specification (i.e., policies) and considers all possible failure scenarios, so the control plane is guaranteed to operate correctly[9] until the policies change (which may necessitate further repairs).

Several prior works [37, 14, 40, 23] have used constraint solving to generate program repairs. DirectFix [37] is the most similar to CPR, insofar as it represents a program as a circuit and uses MaxSMT to identify a minimal set of circuit connections that must be added or removed to satisfy the target semantics. However, DirectFix's notion of minimality is based on syntactic similarity—which has also been emphasized in other work [12, 45, 42, 30]—whereas CPR is concerned with the size of the change (in terms of number of devices and lines of configuration). Nonetheless, we plan to explore syntactic similarity of configuration changes in the future, as it can make the repaired configurations easier for network operators to understand.

In addition to constraint solving, program repair has been conducted using abstract interpretation [33, 42], games [24], mutation [13], and genetic algorithms [6, 32]. Several of these approaches offer better scalability than constraint solving, and may allow for even faster computation of repairs. In the future, we plan to explore the application of such techniques to control plane repairs.

Synthesizing a network control plane directly from policies [48, 8, 39, 15] can avoid bugs in the first place. However, this requires a wholesale replacement of the network's current control plane, which requires significant overhead and network downtime.

11 CONCLUSION

Manually repairing distributed network control planes to conform to a diverse set of policies, under all failures, is a daunting task. Not only do network operators need to reason about correctness across routers, traffic classes, and policies, they also need to consider the complexity of the repair. Fortunately, we have shown that it is possible to *automatically generate correct, minimal control plane repairs* using a carefully constructed encoding of the control plane's semantics and MaxSMT-based constraint solving. In particular, we introduced a new hierarchical abstract representation for control planes (HARC) that is well suited for network repair, and we presented a MaxSMT formulation that encodes the requisite characteristics a network's HARC must and should satisfy to obtain a repair that satisfies all policies through changes to a minimal number of lines of configuration. Detailed evaluation of our system using real configurations from 96 data center networks showed CPR produces repairs for 98% of the networks in less than a minute, and these repairs required the same or fewer configuration changes than hand-written repairs in 79% of the networks.

[9]Assuming the specification is complete.

12 ACKNOWLEDGEMENTS

We thank Saw Lin for his assistance with evaluation. Thanks also to the anonymous reviewers and our shepherd Brad Karp for their insightful feedback. This work is supported by National Science Foundation grant CCF-1637427.

REFERENCES

[1] https://bitbucket.org/uw-madison-networking-research/arc.

[2] Cisco IOS configuration fundamentals command reference. http://www.cisco.com/c/en/us/td/docs/ios/fundamentals/command/reference/cf_book.pdf.

[3] The z3 theorem prover. https://github.com/Z3Prover/z3.

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

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[6] A. Arcuri. On the automation of fixing software bugs. In *International Conference on Software Engineering (ICSE)*, 2008.

[7] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *SIGCOMM*, 2017.

[8] 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 *SIGCOMM*, 2016.

[9] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[10] D. Burton and P. L. Toint. On an instance of the inverse shortest paths problem. *Math. Program.*, 53:45–61, 1992.

[11] D. Caldwell, S. Lee, and Y. Mandelbaum. Adaptive parsing of router configuration languages. In *IEEE Internet Network Management Workshop (INM)*, 2008.

[12] L. D'Antoni, R. Samanta, and R. Singh. Qlose: Program repair with quantiative objectives. In *International Conference on Computer Aided Verification (CAV)*, 2016.

[13] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[14] F. Demarco, J. Xuan, D. L. Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA)*, 2014.

[15] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev. Network-wide configuration synthesis. Technical Report 1611.02537, arXiv, 2016.

[16] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. D. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[17] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[18] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

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

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

[21] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. Technical Report TR1838, University of Wisconsin-Madison, 2016.

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

[23] D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using SAT. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011.

[24] A. Griesmayer, R. Bloem, and B. Cook. Repair of boolean programs with an application to C. In *International Conference on Computer Aided Verification (CAV)*, 2006.

[25] H. Hojjat, P. Rümmer, J. McClurg, P. Cerný, and N. Foster. Optimizing horn solvers for network repair. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2016.

[26] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[27] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[28] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[29] E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In *Computer Aided Verification (CAV)*, 2015.

[30] R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 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 *SIGCOMM*, 2008.

[32] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering (ICSE)*, 2012.

[33] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012.

[34] R. Mahajan, D. Wetherall, and T. E. Anderson. Understanding BGP misconfiguration. In *SIGCOMM*, 2002.

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

[36] G. S. Malkin. Rip version 2. STD 56, RFC Editor, November 1998. http://www.rfc-editor.org/rfc/rfc2453.txt.

[37] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *International Conference on Software Engineering (ICSE)*, 2015.

[38] J. Moy. Ospf version 2. STD 54, RFC Editor, April 1998. http://www.rfc-editor.org/rfc/rfc2328.txt.

[39] S. Narain, G. Levin, S. Malik, and V. Kaul. Declarative infrastructure configuration synthesis and debugging. *Journal of Network and Systems Management*, 16(3):235–258, Sept. 2008.

[40] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *International Conference on Software Engineering (ICSE)*, 2013.

[41] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4). RFC 4271, RFC Editor, January 2006. http://www.rfc-editor.org/rfc/rfc4271.txt.

[42] R. Samanta, J. V. Deshmukh, and E. A. Emerson. Automatic generation of local repairs for boolean programs. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2008.

[43] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: Scalable symbolic execution for modern networks. In *SIGCOMM*, 2016.

[44] Y.-W. E. Sung, X. Tie, S. H. Wong, and H. Zeng. Robotron: Top-down network management at facebook scale. In *SIGCOMM*, 2016.

[45] C. von Essen and B. Jobstmann. Program repair without regret. In *International Conference on Computer Aided Verification (CAV)*, 2013.

[46] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE)*, 2009.

[47] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated bug removal for software-defined networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[48] Y. Yuan, R. Alur, and B. T. Loo. NetEgg: Programming network policies by examples. In *Workshop on Hot Topics in Networks (HotNets)*, 2014.

[49] P. Zave, R. A. Ferreira, X. K. Zou, M. Morimoto, and J. Rexford. Dynamic service chaining with dysco. In *SIGCOMM*, 2017.

[50] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. A survey on network troubleshooting. Technical Report TR12-HPNG-061012, Stanford University, June 2012.