Synthesis of Fault-Tolerant Distributed Router Configurations

KAUSIK SUBRAMANIAN, LORIS D'ANTONI, and ADITYA AKELLA, University of Wisconsin-Madison, USA

Operators of modern networks require support for diverse and complex end-to-end policies, such as, middlebox traversals, isolation, and traffic engineering. While Software-defined Networking (SDN) provides centralized custom routing functionality in networks to realize these policies, many networks still deploy "legacy" control planes running distributed routing protocols like OSPF and BGP because these protocols are scalable and robust to failures. However, realization of policies by distributed control plane configurations is manual and error-prone. We present Zeppelin, a system for automatically generating policy-compliant control planes that also behave well under majority of small network failures. Zeppelin differs from existing approaches in that it uses policy-compliant paths to guide the synthesis process instead of directly generating policy-compliant configurations. We show that Zeppelin synthesizes highly resilient and policy-compliant configurations for real topologies with up to 80 routers.

CCS Concepts: • **Networks** → **Network manageability**; **Routing protocols**;

Additional Key Words and Phrases: Zeppelin; Synthesis; Fault Tolerance; Network Management; Routing protocols; Hierarchical network control plane

ACM Reference Format:
Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2018. Synthesis of Fault-Tolerant Distributed Router Configurations. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 1, Article 22 (March 2018), 26 pages. https://doi.org/10.1145/3179425

1 INTRODUCTION

Programming networks to correctly forward flows according to user- and application-induced policies is difficult and error-prone [19, 33]. At least three common characteristics of network policies are to blame: (1) A network may need to satisfy several types of policies, including reachability (i.e., which endpoints can communicate), isolation (i.e., which flows cannot share links), service chaining (i.e., which "middleboxes", e.g., firewalls or load balancers, must be traversed), resilience (e.g., number of available backup paths), and traffic engineering (e.g., minimizing average link utilization). (2) The network must provide certain guarantees in the event of failures. Ideally, every set of forwarding paths (i.e., data plane) installed in the network should conform to the above policies, otherwise performance, security, or availability problems may arise. (3) Most policies are global—i.e., they concern end-to-end paths, not individual devices/links.

The global nature of network policies is one motivation for software-defined networking (SDN). SDN allows paths to be centrally computed over a global view of the network. However, ensuring paths are correctly computed and installed in the presence of failures is difficult in practice, even

Authors' address: Kausik Subramanian; Loris D'Antoni; Aditya Akella, University of Wisconsin-Madison, 1210 W Dayton St, Madison, WI, 53706, USA, sskausik08@cs.wisc.edu, loris@cs.wisc.edu, akella@cs.wisc.edu.

2476-1249/2018/3-ART22 $15.00
https://doi.org/10.1145/3179425

if the SDN controller is distributed [1]. Traditional control planes rely on distributed protocols such as Open Shortest Path First (OSPF) and Border Gateway Protocol (BGP) to compute paths; these protocols typically employ variants of least cost path computation, and react to failures by recomputing least cost paths and installing forwarding state in routers that induces the paths. In contrast to centralized SDN, traditional control planes offer greater fault tolerance; but, determining the appropriate distributed realization of policies is hard [4].

Our high-level goal is to develop a system to *automate the process of creating a correct and failure-resilient distributed realization of policies in a traditional control plane.* To be useful, such a system must satisfy a few key requirements. (1) It must handle a wide range of commonly-used policies—including reachability, service chaining, and traffic engineering—to meet applications' diverse security and compliance requirements. (2) It must ensure that configurations are resilient to network malfunctions such as link failures. (3) It must provide support for realizing hierarchical control planes— where a network is split into several "domains" atop which a hierarchy of intra- and inter-domain control plane configurations is deployed—to ensure scalability for large networks [20]. (4) To improve manageability and network cost-effectiveness [7, 15], it must ensure that configurations obey certain general rules-of-thumb [6], e.g., limiting the number of lines of configurations and the use of certain configuration constructs.

Thus, our work contrasts with prior efforts which suffer from one or more drawbacks: they generate SDN- or BGP-specific control planes for a limited range of policies (e.g., peering) [3, 4, 18, 24, 26, 29, 32]; do not attempt to be resilient to failures [12]; do not support generating hierarchical control planes [4]; and do not enable generation of simple-to-manage network configurations.

The problem of synthesizing router configurations for which the distributed control plane is resilient to failures and generates policy-compliant paths is computationally hard. Even generating a set of policy-compliant paths for an SDN is computationally hard—e.g., enforcing isolated paths is NP-complete. While it is possible to develop algorithms for individual policies, accommodating multiple policies and the above requirements is extremely difficult.

An attractive possibility, motivated by recent progress in program synthesis, is to use Satisfiability Modulo Theories (SMT) solvers to automatically search the space of distributed configurations for a suitable "solution"— i.e., concrete router configurations that satisfy input policies and the aforementioned four requirements. SMT solvers provide support for constraint solving for propositional logic (SAT) and linear rational arithmetic (LRA); these powerful theories can encode properties on network paths and configurations. Thus, an SMT-search based approach can, in theory, provide support for multiple policies and different manageability requirements. This approach also decouples the requirements from the underlying search, and thus, could be extended to support new policies.

However, using SMT solvers in our context is non-trivial. To infer the concrete set of paths induced by network configurations, one has to incorporate into synthesis complex concepts—e.g., reasoning about shortest path algorithms requires constraints in complex theories (SAT and LRA). Even with recent advances in SMT solving, approaches that directly generate configurations from policies do not scale to moderately-sized networks or sets of policies [12]. Furthermore, generating resilient control planes needs reasoning about how protocols react to failures, which further complicates an already intractable synthesis problem. Control plane hierarchy and manageability requirements further complicate the problem.

In this paper, we present Zeppelin, a system that overcomes the above challenges in SMT-based distributed configuration synthesis. Zeppelin uses a two-phased approach for tractability (Figure 1) that does not attempt to generate a policy-compliant control plane in a single step. First, Zeppelin uses Genesis [30] to synthesize paths—i.e., the network forwarding state—that are compliant with given policies, such as, waypoints and isolation. Zeppelin then generates

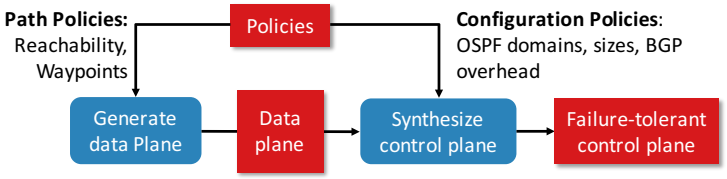


Fig. 1. Two-phase process for generating a control plane with failure-tolerance properties

intra-domain (shortest-path OSPF) and inter-domain (BGP) router configurations that induce the forwarding state synthesized by Genesis *and* provide high resilience. Zeppelin caters for moderate-sized enterprises and multi-tenant datacenters which require support for diverse policies.

We consider three settings with progressively more complex policies and resilience requirements and show how Zeppelin, using its two-phase approach, can effectively generate highly-resilient and policy-compliant solutions.

First, we consider a setting in which the operator of a hierarchically structured network wants to enforce a large diverse set of complex policies (waypoint traversal, isolation, traffic engineering, etc). The operator also desires a simple notion of *connectivity-resilience*—i.e., under most common link failures, even if operator-specified policies (like waypoints) are not enforced, majority of packets can still reach their destination. In this case, Zeppelin synthesizes OSPF configurations using linear constraint solving to compute link weights, and uses the unsatisfiable cores of failed solving attempts to judiciously place a small number of static routes. Zeppelin synthesizes BGP configurations directly from the domain mapping (i.e., which router belongs to what domain) and the paths. Using these techniques, Zeppelin can generate configurations that are 10% more resilient than configurations that only use static routes.

Second, we consider a setting in which the operator wants to generate configurations which reduce the number of policy violations occurring under common link failure scenarios. We call this notion *policy-resilience*. Since synthesizing policy-resilient configuration is very difficult in the general case, we focus on a restricted class of policies: for traffic class (src-dst subnet pair), we allow the operator to specify a set of waypoints that packets must traverse before reaching their destination. We modify our linear constraints to ensure that at least two paths that traverse the waypoints have path cost (e.g., OSPF path cost) lower than any path not going through a waypoint, thus providing 1-resilience under failure. Using this, Zeppelin can generate configurations that are 140% more policy-resilient than configurations generated with our first technique (i.e., reduces policy violations under failures by 140%).

Finally, we consider a setting where the operator can specify bounds on the number and sizes of domains her control plane is organized into, and the ability to assign routers to different domains. We present a stochastic search technique that leverages this flexibility to look for ways to assign routers to different domains so that the synthesized configurations have even higher resilience. Through this, Zeppelin can further improve the resilience of the configurations by 10%. We show that Zeppelin can be naturally used to bound or optimize different configuration metrics such as static routes and BGP configurations' size to improve network manageability.

Our experiments show that, thanks to its two-phase approach, Zeppelin synthesizes connectivity-resilient configurations for medium-sized datacenter topologies in < 10 *minutes* and policy-resilient configurations in under an hour. Notably, Zeppelin's performance is *2-3 orders of magnitude* faster than the state-of-art network configuration synthesis system SyNET [12], which uses a direct synthesis approach based on SMT.

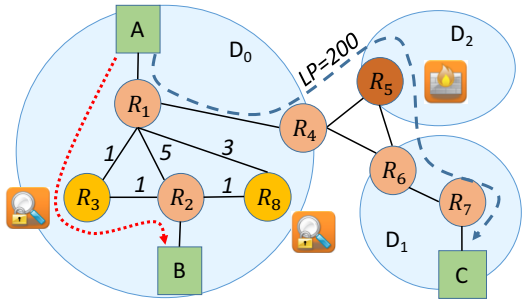


Fig. 2. Example demonstrating the two-phase synthesis approach. Genesis produces policy-compliant paths (red and blue), which are used as input by Zeppelin to synthesize the OSPF and BGP configurations.

Contributions. We make the following contributions.

- Zeppelin, a framework for that enforces policies in "traditional" (OSPF and BGP) networks by synthesizing highly-resilient router configurations. Zeppelin uses concrete paths to guide the synthesis instead of directly generating policy-compliant configurations.
- An algorithm for synthesizing policy-compliant configurations with few statically assigned routes. This algorithm yields configurations with high network connectivity even in the presence of link failures (§ 4).
- An algorithm for synthesizing policy-compliant configurations that is specialized for waypoint policies. This algorithm yields configurations that have high policy compliance even in the presence of link failures (§ 5.2).
- A stochastic search mechanism for finding partitions of the network into multiple routing domains which yield configurations with higher resilience (§ 6).
- An implementation of Zeppelin together with an evaluation of its algorithms for different workloads, and comparison with a state-of-the-art configuration synthesis tool (§ 7).

2 OVERVIEW

Before formally defining the problem tackled by Zeppelin and the algorithms, we present an overview of Zeppelin using an end-to-end example. Consider the hierarchical network in Figure 2 which is split into three OSPF domains, and these domains communicate using BGP. Host A talks to hosts B and C with the following policies: traffic to B must go through a firewall at router R_3, traffic to C goes through an IDS at R_5 and traffic to B and C do not share any links (We discuss the policies supported in detail in Section 3.2). Zeppelin synthesizes OSPF and BGP router configurations satisfying the policies. Illustrating our two-phase approach (Figure 1), Zeppelin first uses Genesis [30], a policy-enforcement framework for SDNs to find policy-compliant paths (indicated by the blue and red paths). Note that there exists multiple solutions for the given policies, we show how the control plane is synthesized using one of the solutions:

- For traffic $A \rightarrow B$, the path is entirely inside a single OSPF domain, so Zeppelin has to find OSPF weights such that the shortest path is $R_1 \rightarrow R_3 \rightarrow R_2$. From the OSPF weights shown in Figure 2, we can see that the weight of path $R_1 \rightarrow R_3 \rightarrow R_2$ (2) is smaller than that of path $R_1 \rightarrow R_2$ (5).
- For traffic $A \rightarrow C$, the path traverses through multiple domains, so Zeppelin configures BGP such that routing across domains follows the Genesis path. In this case, we need traffic to go through a longer path in terms of AS path length; Zeppelin sets R_4's BGP local preference for route C from R_5 to a higher value (200), so that R_5 is preferred over R_6.

Term	Description	Term	Description
$T = (V, L)$	Topology T of routers V and links L	$D(r_1, r_2)$	Shortest OSPF distance from r_1 to r_2
$\Theta(r) = \theta$	Router r belongs to domain θ	$G(\theta, \lambda) = \{g_1, g_2\}$	Set of exit gateway routers for λ in domain θ
$SR(r, \lambda) = \{r_1\}$	Static route $r \rightarrow r_1$ for destination λ	$LP(r, r', \lambda)$	Local preference at r to choose r' route to λ
$W(r_1, r_2)$	OSPF weight of link $r_1 \rightarrow r_2$	$IF(r_2, \lambda) = \{r_1\}$	r_2 will not advertise λ to r_1 using iBGP

Fig. 3. Different configuration parameters in Zeppelin

- Zeppelin can find better control planes in terms of how the network is split into domains. In this case, Zeppelin will suggest moving R_5 to domain D_1. This simplifies R_4's BGP configuration as the local preference entry for C is not required.
- With a backup firewall at R_8, Zeppelin automatically synthesizes resilient OSPF configurations for D_0 such that if either $R_1 - R_3$ or $R_3 - R_2$ link fails, traffic to B goes through the firewall at R_8.

Zeppelin is fundamentally different from state-of-art approaches for policy enforcement in legacy networks: Propane [4] generates BGP configurations from peering and path requirements, while Fibbing [31] uses fake advertisements to centrally control OSPF routing. SyNET [12] targets BGP and OSPF configuration synthesis, however, it does not cleanly tackle domain assignments or provide support for resilience. Zeppelin has more general policy coverage, supports automatic domain assignment (which is manual and difficult today), and can provide fault-tolerant configurations, all in one system.

3 PROBLEM DEFINITION

In this section, we formally define the problems addressed in this paper. We first describe the representation of different routing protocols; we model static routes, OSPF shortest-path routing, and BGP preference-based routing. Then, we define what it means for a configuration to meet a given set of policies and present the configuration synthesis problems we tackle. Finally, we formalize two resilience metrics called connectivity-resilience and policy-resilience; the goal of Zeppelin is optimizing these metrics.

3.1 Routing Model

We represent the physical router topology as a directed graph $T = (V, L)$, where V is the set of routers and $L \subseteq V \times V$ is the set of links. Throughout the paper we assume T is fixed. We use the neighbour function $N(s) = \{s' \mid (s, s') \in L\}$ to denote the set of neighbour routers of s. A path $\pi = (r_1, r_2)(r_2, r_3) \cdots (r_{n-1}, r_n) \in L^*$ is a loop-free valid path if a router is not visited more than once, denoted by $valid(\pi)$. We also represent the path π as $r_1 \rightarrow r_2 \rightarrow \cdots \rightarrow r_n$. We write $l \in \pi$ when the path π contains the link l. We define $\Lambda \subset IP$ to denote the set of destination IP addresses; distributed protocols make forwarding decisions based on the destination address/subnet.

Hierarchical control planes partition the network into multiple connected components called domains. We define a router domain assignment function $\Theta : V \mapsto \mathbb{N}$ which maps each router to a domain (denoted by a number). We assume each domain uses OSPF as the intra-domain routing protocol and BGP as the inter-domain routing protocol. We summarize the notations in Figure 3.

Static Routes. These are statically configured next-hop routes with higher priority over those routes computed by OSPF and BGP.

OSPF. In Open Shortest Path First (OSPF) routing, traffic from a router r to a destination λ is forwarded across the shortest path from r to λ. We define the OSPF routing function $\Re_{ospf}$:

$V \times \Lambda \mapsto 2^V$ so that $\Re_{ospf}(r, \lambda)$ describes the next-hop for router r which lies on the shortest path for a destination λ.

BGP. BGP is a path-vector inter-domain routing protocol that connects different domains (or ASes). BGP routers can be configured with a local preference (LP) to assign higher priorities to specific routes for a particular destination. The Internal BGP (iBGP) protocol is used to exchange external BGP routes among BGP routers belonging to the same domain. Informally, iBGP propagates local preferences of the routes to all the BGP routers within the same domain. We can configure iBGP filters (IF) to prevent a router from advertising a route.

A BGP router receives eBGP and iBGP routes; it chooses a route with highest local preference, and if there is a tie, it chooses the route that traverses fewer domains [27]. We assume all ties are broken using these criteria. After route selection, if a eBGP route is preferred, the route is redistributed to OSPF. Therefore, for a domain θ, only BGP routers which redistribute routes for destination λ to OSPF belong to the gateway router set $G(\theta, \lambda)$. We define a partial BGP routing function $\Re_{bgp} : V \times \lambda \mapsto V$ such that $\Re_{bgp}(r, \lambda)$ describes the next-hop BGP router for the BGP router r when forwarding traffic for a destination λ.

OSPF+BGP+SR. We now describe how routing happens in hierarchical networks with all the previously described protocols used together. We express the complete network configuration C as a tuple $(T, \Theta, W, LP, IF, SR)$. We define the routing function $\Re^C : V \times \Lambda \mapsto 2^V$ so that $\Re^C(r, \lambda)$ describes the next-hop router for the router r when forwarding traffic for a destination λ. The priority order based on administrative distance [11] is SR > BGP > OSPF. Piecing together the different routing protocols and their interactions, the routing function $\Re^C$ for the hierarchical domain network configuration C is defined as follows:

$$\Re^C(r, \lambda) = \begin{cases} SR(r, \lambda) & \text{if } SR(r, \lambda) \neq \emptyset, \\ \Re^C_{bgp}(r, \lambda) & \text{if } r \in G(\Theta(r), \lambda), \\ \Re^C_{ospf}(r, \lambda) & \text{otherwise.} \end{cases}$$

Induced paths. We assume a finite set of packet classes $PC = \{0, \ldots, C_{pc}\}$ and map each reachability policy that requires the existence of a path between two endpoints to a unique integer in PC. The rest of the policies specify properties on these paths. Each packet class pc is associated with a tuple $(s_{pc}, d_{pc}, \lambda_{pc})$ which specifies the path from s_{pc} to d_{pc} for destination IP λ_{pc}.

Definition 3.1 (Induced Paths). Given a configuration C, the set of paths induced by the configuration C for the packet class pc:

$$\mathfrak{P}^C(pc) = \{\pi = (u_1, v_1) \ldots (u_n, v_n) \mid valid(\pi) \wedge u_1 = s_{pc} \wedge v_n = d_{pc} \wedge \forall i.\ v_i \in \Re^C(u_i, \lambda_{pc})\}$$

Given a set of packet classes PC, the set of paths induced by C is defined as $\mathfrak{P}^C(PC) = \cup_{pc \in PC} \mathfrak{P}^C(pc)$.

3.2 Policy Support

One of the foremost tasks in network management in enterprise and multi-tenant datacenters—i.e., where different entities ("tenants") share the datacenter's infrastructure (compute, network etc.)—is programming networks to forward traffic in a manner consistent with user- and application-induced high-level policies for performance and security considerations. Unlike SDN, "traditional" networks are heterogeneous and run different kinds of protocols, and thus, operators require support to enforce and/or optimize different configuration structure and properties. We classify Zeppelin's policy support into two categories: (1) *path policies*: high-level intents on paths of different traffic classes, and (2) *configuration policies*: low-level intents on the deployed router configurations.

Figure 4a describes the common path policies supported by Zeppelin. Operators can specify policies in a declarative manner using a high-level policy language. Given a set Ψ of path policies,

Policy	Description
Reachability	There is a path from router s to router t for subnet λ
Reachability + Waypoints	The path from s to t for destination λ traverses a set of waypoints
Isolation	Paths of two reachability policies do not share links
Traffic Eng.	Minimize total/max link utilization

(a) Path Policies

Policy	Description
Count	No. of OSPF domains: $c_1 \leq N_D \leq c_2$
Domain Size	Limit number of routers in a domain: $l \leq ds \leq u$
BGP Enable	Enable BGP on routers B (hardware constraint)
Static Route sc	Limit number of static routes: $sc < SC$
BGP Config. Overhead bc	Upper bound the number of local preference entries and iBGP filters $bc < BC$
Cost Minimization	Minimize user-defined cost $expr(sc, bc)$

(b) Configuration Policies

Fig. 4. Zeppelin Policy Support

we say that a set of paths Π is policy-compliant with respect to Ψ, denoted $\Pi \models \Psi$, if the paths in Π satisfy all the policies in Ψ [30].

Reachability and Waypoints. This policy enables network communication between pairs of a tenant's virtual instances (VM), applications, or hosts. The tenant may wish that the flow between two of her end hosts, or from another tenant, must traverse specific middleboxes, which we also refer to as "waypoints".

Isolation. Tenants may require various Quality-of-Service (QoS) or security guarantees since the underlying infrastructure is shared among tenants. In the extreme, a tenant could require that her flows are not affected in any manner by any other tenant by strictly isolating the path of the tenant's flows from others' flows.

Traffic Engineering. While support for the above policies can be used to satisfy tenant requirements, network operators need to carefully manage constrained resources. Operators may also want to balance load on their network infrastructure. This is often done by optimizing a network-wide objective such as total or maximum utilization of network links due to traffic induced by all tenants.

The high-level path policies are enforced by different low-level configuration constructs pertaining to routing protocols like OSPF and BGP, and ideally, the operator requires support to impose constraints on the structure and properties of the deployed configurations. Figure 4b describes all the configuration policies supported by Zeppelin for hierarchical control planes.

OSPF Domains and Sizes. We support a policy which restricts how many OSPF domains a configuration might have. This policy is useful for avoiding situations resulting in too many domains in the hierarchical split which can be difficult to administer. Another policy allows the operator to bound the size of each OSPF domain because OSPF does not scale gracefully with network size.

Resource Constraints. Certain routers may not be suited to run BGP due to resource constraints. Thus, the operator can specify what set of routers $B \subseteq V$ are BGP-compatible.

Configuration Metrics. Zeppelin provides ways to specify upper bounds on the number of static routes sc and BGP configuration overhead bc which we define as the sum of local preference entries and iBGP filters. While static routes are a powerful tool, it is desirable to limit their usage since they can lead to undesired routing behaviors such as routing loops. Similarly, it is desirable to limit the number of BGP configurations since they increase the complexity of the network. Zeppelin can also try to minimize certain expressions over the quantities sc, and bc—e.g., $max(sc, bc)$.

3.3 Synthesis of policy-compliant configurations

We now define when a configuration C is policy-compliant, present our synthesis problems, and introduce two definitions of resilience we will use throughout the paper.

Definition 3.2 (Policy-Compliance and Synthesis). Given a set of path policies Ψ and a set of configuration policies P, a configuration C is policy-compliant with (Ψ, P), written as $C \models (\Psi, P)$, if the set of induced paths satisfies Ψ—i.e., $\mathfrak{P}^C(PC) \models \Psi$—and C satisfies the policies in P. The *configuration synthesis problem* is to find, given Ψ and P, a configuration C that is policy compliant with (Ψ, P).

Our approach will proceed in two phases, one of which solves the following sub-problem.

Definition 3.3 (Path-Compliance and Synthesis). Given configuration policies P and a set of paths Π over packet classes PC, a configuration C is path-compliant with (Π, P), if $\mathfrak{P}^C(PC) = \Pi$ and C satisfies the policies in P. The *path-compliance synthesis problem* is to find, given P and Π, a configuration C that is path-compliant with (Π, P).

1-Resilience metrics. One of the main goals of this paper is to generate configurations that are highly resilient to failures. In this work, we focus on resilient configurations for single link failures. [1] For a packet class pc and configuration C, we define the set of links *affecting* pc as $\mathbb{L}(pc) = \{l \mid \exists \pi \in \mathfrak{P}^C(pc).\ l \in \pi\}$. For a single failure of a link l, a packet class pc is affected if only if $l \in \mathbb{L}(pc)$. Given a configuration $C = ((V, L), \Theta, W, LP, IF, SR)$ and a link $l \in L$, we write C_l to denote the configuration $C = ((V, L \setminus \{l\}), \Theta, W, LP, IF, SR)$ in which the link l has been removed from the topology. We present two different resilience metrics we will use in the following sections.

The first metric describes how good a given configuration is at preserving policy-compliance under an arbitrary single link failure. The score measures, for each policy pc, what percentage of the links affecting pc causes a policy to still hold when failing.

Definition 3.4. The *policy-resilience* score of a configuration $C = ((V, L), \Theta, W, LP, IF, SR)$ is defined as:

$$PR(C) = \frac{\sum_{\forall pc} |\{l \mid l \in \mathbb{L}(pc) \ \wedge \ C_l \models (\Psi_{pc}, P)\}|}{\sum_{\forall pc} |\mathbb{L}(pc)|}$$

where $C \models (\Psi_{pc}, P)$ means that C is compliant for the set of policies Ψ_{pc} that affect packet class pc.

In general, generating configurations with high policy-resilience is hard, since there may be only a few paths that satisfy a certain policy. Therefore, we also consider a second more relaxed metric, which describes how good a given configuration is at preserving connectivity (but not policy compliance) upon a single link failure.

Definition 3.5. The *connectivity-resilience* score of a configuration $C = ((V, L), \Theta, W, LP, IF, SR)$ is defined as follows:

$$CR(C) = \frac{\sum_{\forall pc} |\{l \mid l \in \mathbb{L}(pc) \ \wedge \ \mathfrak{P}^{C_l}(pc) \neq \emptyset\}|}{\sum_{\forall pc} |\mathbb{L}(pc)|}$$

Ideally, operators desire a resilience score of 1 (perfectly resilient for policy-compliance or preserving connectivity). However, achieving this objective is in practice difficult due to the increased number of configurations one needs to consider, and sometimes impossible due to the structure of the network and the nature of policies. In practice, Zeppelin can synthesize configurations with high resilience scores and operators can run it to generate different configurations until a threshold score is crossed.

[1] We do not consider multiple link failures as they occur with lower probability [16].

4 FROM POLICIES TO CONNECTIVITY-RESILIENT CONFIGURATIONS

We now present an algorithm for synthesizing distributed configurations that adhere to path policies like the one described in Figure 4a. In general, no existing tool for synthesizing distributed configurations can handle this heterogeneous set of these policies—e.g., isolation—and, even less so, generate resilient configurations. Due to the complexity of the problem, we focus on synthesizing configurations that are *policy-compliant* and have high *connectivity-resilience*.

4.1 A two-phase approach

Existing approaches to configuration synthesis try to directly generate configurations from the given policies [12], but a direct approach leads to scalability issues as well as limitation in the set of supported policies. Instead of directly generating configurations from policies, Zeppelin uses a two-phase approach: first it generates policy-compliant paths and then it synthesizes distributed configurations that realize these paths and have high connectivity resilience.

For the first phase, we use Genesis [30], a network management system which synthesizes SDN forwarding tables enforcing some input policies. Some of the policies supported by Genesis are specified in Figure 4a. Given a set of policies, Genesis generates a set of constraints so that solutions to these constraints are forwarding tables that can be used to extract the policy-compliant paths.

However, we need to modify the Genesis constraints to integrate with Zeppelin. Legacy protocols only support destination-based forwarding, unlike OpenFlow switches [21]. A router cannot forward a subnet's traffic to different routers based on source (or other packet headers). Thus, we add additional constraints to ensure the paths generated by Genesis follow destination-based forwarding: paths obtained from Genesis for a destination will form a directed tree.

Let us now look at how we solve the path-compliance problem. Formally, we are given a set of paths Π, a topology $T = (V, L)$, a domain-assignment function Θ, and we want to find functions of the configuration $C = (\Theta, W, LP, IF, SR)$ such that $\mathfrak{P}^C(PC) = \Pi$. In a software-defined OpenFlow [21] network, we can program switch rules to forward traffic to a particular next-hop switch. Therefore, a trivial solution to the path-compliance problem in SDNs is to install forwarding rules along the policy-compliant paths [30]. Static routes provide the functionality of specifying the next-hop router, thus, a solution to the path-compliance problem is to simply add a static route to $SR(\lambda)$ for every link l in set of the paths Π. Similar to static routes, Fibbing [31] used fake advertisements to achieve fine-grained control over OSPF forwarding.

Under no failures, the trivial solution of using static routes ensures policy-compliance, as traffic follows the paths provided by Genesis. However, this solution uses more static routes than necessary and, in the presence of failures, static routes may create routing loops that make certain connections unreachable, hence reducing connectivity-resilience. Thus, we set our goal to synthesizing configurations which are policy-compliant when there are no network failures, and improving connectivity-resilience by using fewer static routes.

We first show how to synthesize path-compliant intra-domain configurations and then extend to inter-domain configurations.

4.2 Synthesizing Path-compliant Intra-domain Configurations

In this section, we show how to synthesize path-compliant intra-domain configurations that use few static routes. The problem of optimally placing static routes is NP-hard (Theorem A.1), therefore, we propose a non-optimal greedy strategy. We first show how to solve the problem when there exists a solution with no static routes and then extend our technique to greedily add static routes when needed. We use efficient off-the-shelf linear programming solvers for solving the system of constraints which are presented in the following sections.

4.2.1 Intra-domain Synthesis without Static Routes.

In the absence of static routes, OSPF routers use edge weights to choose the shortest weighted path for each pair of endpoints. The problem of synthesizing the weight function W that realizes an input set of paths Π is a variation of the so-called *inverse shortest path* problem [8]. For a destination IP λ, we call ξ_λ the directed tree of T obtained by only keeping the nodes and edges that are traversed by paths in Π for λ; the root of the tree is the destination router connected to λ. This destination tree property is due to the modifications to Genesis to support OSPF's destination-based forwarding. We define $\Delta = \{\xi_\lambda \mid \lambda \in \Lambda\}$ to be the set of all destination trees.

Given a set of input paths Π, Zeppelin generates a set of linear constraints to find weights for each directed link in the domain. The constraints use the variable $W(r_1, r_2)$ and $D(r_1, r_2)$ to denote the weight and shortest distance respectively. We add the equation $D(s, s) = 0$. Equation (1) guarantees that $D(s, t)$ is smaller or equal to the shortest distance from s to t.

$$\forall s, t.\ \forall r \in N(s).\ D(s, t) \leq W(s, r) + D(r, t) \tag{1}$$

Intuitively, the shortest path connecting s to t must traverse through one of the neighbor routers of s, and thus, distance can be defined inductively as the shortest among distances from the neighbors.

For each destination tree $\xi_\lambda \in \Delta$, we add equations to ensure that the input paths with destination λ are the shortest ones. Notice that, if a path π is the shortest path between its endpoints, every subpath of π also has to be the shortest path between its endpoints. For each tree ξ_λ, we define the neighbor $N_{\xi_\lambda}(s)$ to denote the next-hop neighbor of router s in the destination tree ξ_λ. We denote the router directly connected to λ (the root of the tree ξ_λ) by R_λ.

For a path π, $\sum_\pi W$ denotes the sum of weights of all links in π. The following equation ensures that, for any node s in ξ_λ, the path $(s, r_1) \cdots (r_n, R_\lambda)$ in ξ_λ is the *unique shortest path*:

$$\forall s \in \xi_\lambda.\ \forall r \in N(s) \setminus N_{\xi_\lambda}(s).\ \sum_{(s,r_1)\cdots(r_n,R_\lambda)} W < W(s, r) + D(r, R_\lambda) \tag{2}$$

These constraints guarantee that the sum of the weights belonging to the path from s to R_λ in ξ_λ is strictly smaller than any other path that goes to R_λ via a node n'. Note that, while $D(r, R_\lambda)$ can be smaller than the actual shortest distance from r to R_λ, $D(r, R_\lambda)$ is used to upper bound the sum of edge weights in ξ_λ, and thus, the solution to the edge weights will ensure paths in ξ_λ are the shortest.

Soundness and Completeness. OSPF synthesis is sound—i.e., weights satisfying constraints (1) and (2) will ensure all paths in Π are the unique shortest paths. OSPF synthesis is complete as well—i.e., if all paths in Π are the unique OSPF shortest paths, the weights will satisfy the constraints.

4.2.2 Intra-domain Synthesis with Static Routes.

If the system of equations presented in Section 4.2.1 admits a solution, the values of the $W(s, t)$ variables are the weights we are trying to synthesize, otherwise the intra-domain synthesis problem cannot be solved without static routes. Consider the paths in Figure 5 between s and t for λ_1 and λ_2. There exists no solution as the following is required for λ_1: $W(s \to r_1 \to t) < W(s \to t)$, and vice-versa for λ_2. To steer the traffic for a particular destination to a next-hop router not on the shortest path, we install a static route.

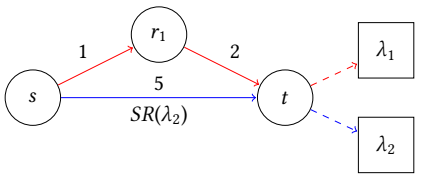


Fig. 5. Example illustrating the use of static routes.

Since the problem of minimizing the number of static routes is NP-hard, we opt to place static routes using a best-effort approach. Our algorithm starts by

trying to synthesize a solution that does not use static routes using the equations proposed in §4.2.1. In the case of a failure, the algorithm uses the "proof of unsatisfiability"—i.e., the unsatisfiable core—generated by the constraint solver to greedily add a small set of static routes. Certain equations are eliminated to model the added static routes and the approach is repeated until a solution is found.

Intuitively, an unsat core or IIS (Irreducible Inconsistent Subsystem) [10] is a subset of the input constraints such that, if all constraints except those in the IIS are removed, the resulting set of linear equations is still unsatisfiable. Moreover, the set is irreducible—i.e., removing any one constraint from the IIS produces a satisfiable set of constraints. In our case, an IIS cannot consist of only constraints from Equation (1) as these constraints admit a trivial solution with all variables set to 0. Therefore, an IIS must contain some constraint of the form given in Equations (2). Let us denote this set of constraints as $\Psi_S(s, \lambda)$:

$$[\sum_{l_i=(r_1,r_2)} W(r_1, r_2) < W(s, r) + D(r, R_\lambda)] \in \Psi_S(s, \lambda)$$

This constraint is added to ensure that router s forwards to traffic to next-hop $N_{\xi_\lambda}(s)$ and not some neighbor router r, based on OSPF weights, but the path through r is causing the unsatisfiability. To remove this inequality from the set of constraints, we add the static route $(s, N_{\xi_\lambda}(s))$ to $SR(\lambda)$. As a result of adding a static route, Zeppelin removes the $\Psi_S(s, \lambda)$ constraints as router s will forward λ traffic to next-hop $N_{\xi_\lambda}(s)$ irrespective of the OSPF distances to the destination; the unsatisfiability caused by this IIS is eliminated. However, the new set of constraints may still be unsatisfiable due to other IISes. We repeat the procedure and add static routes until we obtain a satisfiable set of constraints. In each iteration, there can be more than one way to place a static route and Zeppelin picks one randomly.

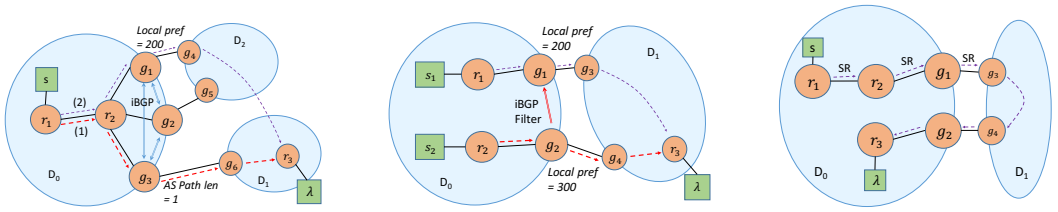
Soundness and Completeness. OSPF synthesis with static routes is sound—i.e, the configuration $C(W, SR)$ will induce the paths Π provided by Genesis. As far as completeness goes, Zeppelin is guaranteed to find a solution if there are no constraints on the number of static routes (use static routes along each path). If there is a configuration policy upper bounding the number of static routes, the synthesis procedure is not guaranteed to find a compliant configuration, if one exists. Since there can only be finitely many solutions for paths and possible SR assignments, we are eventually guaranteed to find a solution by repeating Genesis and OSPF-SR synthesis multiple times and choosing different solutions.

4.3 Synthesizing Path-compliant Inter-domain Configurations

Next, we extend our algorithm to solve the path-compliance problem in the presence of multiple domains. Our algorithm determines the BGP configuration and static routes to ensure that the paths will traverse the correct domains, and new constraints required to OSPF synthesis for the inter-domain scenario.

Configuring BGP and Static Routes. In this section, we describe how to configure BGP routers so that the routes chosen by BGP (and redistributed into OSPF) can yield the policy-compliant paths Π given as input. Note that, the BGP $\rightarrow$ OSPF redistribution can sometimes lead to explosion of OSPF routing tables. However, in our datacenter setting, BGP is used in a restricted sense for management ease and BGP routers are not connected to the external internet directly, thus, only internal prefixes are redistributed.

To enforce the policy-complaint paths, we need to take into account three different cases, which are illustrated in Figure 6. In the first case, path to destination λ exits the domain through a single BGP gateway, illustrated in Figure 6(a). For path (1)—i.e., the red one— to be the chosen path, it is already on the shortest AS path, therefore, the gateway for λ in domain D_0 is g_3, i.e., $G(0, \lambda) = g_3$. However, if path (2)—i.e., the purple one— is the path from Genesis, it has to traverse more domains.



(a) Single BGP Gateway (b) Multiple BGP Gateways (c) Static Routes

Fig. 6. Examples of BGP and static route configurations for inter-domain routing given domain assignment Θ.

Thus, we set local preference for route for λ to g_1 to 200, i.e., $LP(g_1, g_4, \lambda) = 200$. After iBGP route distributions, the gateway route for λ will be through g_1.

In the second case, two paths from different sources to destination λ exit the domain through two different gateways. In Figure 6(b), the path from r_1 to λ exits domain D_0 through gateway router g_1, while the path from r_2 to λ exits through g_2. In this case, both these routes need to be redistributed to the OSPF domain. By assigning local preferences such that $LP(g_1, g_3, \lambda) < LP(g_2, g_4, \lambda)$, g_2 will redistribute a route for λ to the OSPF domain and the traffic from r_2 will correctly exit through g_2. To prevent the traffic from r_1 to exit through g_2, we add an iBGP filter for the connection $g_2 \longrightarrow g_1$ for destination IP λ. Thus, g_1 will redistribute the route it received from g_3. For this example, $G(0, \lambda) = \{g_1, g_2\}$.

Finally, we describe a case in which standard BGP cannot enforce the desired path, therefore requiring the use of static routes. Consider the path for destination λ shown in Figure 6(c) which has a domain loop in the path ($D_0 \longrightarrow D_1 \longrightarrow D_0$), thus, this path cannot be enforced by conventional BGP routing. To circumvent this issue, Zeppelin uses static routes (illustrated in Figure 6(c)). To find the minimal number of static routes to be used in such cases, we find the longest path suffix that has no domain loops (from the start of this path fragment, BGP+OSPF can be used for routing), and add static routes from the source to the start of this path suffix.

Modified OSPF Synthesis. When BGP routes for destination λ are redistributed by multiple gateways to an OSPF domain, Zeppelin uses a modified OSPF synthesis algorithm from §4.2. In case of multiple gateways, an OSPF router will choose the closest BGP gateway in terms of OSPF distance for λ. For instance, a router r will choose gateway g_1 over g_2 if the OSPF distance from r to g_1 is strictly lesser than the distance from r to g_2. We add new constraints on OSPF weights (details omitted for brevity) to ensure that any router on the path is closest to the gateway in the path.

5 FROM WAYPOINT POLICIES TO POLICY-RESILIENT CONFIGURATIONS

The algorithm presented in the previous section generates configurations which guarantees enforcement of complex policies under no failure scenarios, and at the same time, achieves good connectivity-resilience by using fewer static routes. However, the generated configurations do not provide any guarantees as to whether the policies are satisfied when failures occur in the network. In this section, we present a different algorithm that, for a restricted set of policies, can synthesize configurations with high *policy-resilience*. In particular, we focus on *waypoint policies* which play a crucial role in enterprise and datacenter networks for security, performance, and auditing.

5.1 Problem Setup

We consider policies of the form λ: `s >> 𝕎 >> t` where λ is a destination IP subnet, s and t are the source and destination routers respectively, and $\mathbb{W} \subseteq V$ is a set of routers called the *waypoint*

set. Each policy is mapped to a packet class and we say that a configuration complies to a policy for *pc* if all induced paths in $\mathfrak{P}^C(pc)$ traverses some waypoint $w \in \mathbb{W}$. Here, $\mathbb{W}$ can be the set of physical replicas of, for example, a firewall. Given a set of such waypoint policies, our goal is to find a policy-compliant configuration $C = (T, \Theta, W, LP, IF, SR)$ with high policy-resilience—i.e., the configuration must be waypoint-compliant under single-link failures.

We present an algorithm that solves this problem when the network has a single domain; when the network has multiple domains, we assume that all waypoints belong to the same domain and use our new algorithm in the domain which contains the waypoints. This assumption can be realized through Network Function Virtualization [14, 23], which provides waypoint replication and flexible waypoint placement in the network.

5.2 Intra-domain Policy-Resilient Waypoint Compliance

Genesis provides support for link-isolation, which can be used in conjunction with the waypoint policy to generate multiple disjoint paths for a particular destination which are waypoint-compliant. For our case of 1-resilience, Zeppelin uses Genesis to generate two link-disjoint paths, where each path traverses through one of the waypoints in $\mathbb{W}$. Link-disjointedness ensures that a single link failure will not disable both these paths and we could guarantee policy-resilience by ensuring that, under any link failure, traffic to λ traverses one of these two paths. However, enforcing the control plane to always route through one of the two paths is difficult and overly restrictive. Instead, we relax our constraints to guarantee that the two paths are shorter than any non-compliant path in the network. The existence of two such paths still guarantees that traffic under any single link failure, traverses a waypoint.

5.2.1 *Waypoint-Compliance Constraints.*

We now show how to generate constraints that guarantee that a single waypoint-compliant path is shorter than any path which does *not* traverse a waypoint. We define $D(s, t, \mathbb{W})$ to be the distance between s and t for paths that *do not* traverse any waypoint $w \in \mathbb{W}$. We call $D(s, t, \mathbb{W})$ the *non-waypoint distance*. We add constraints to represent these distances by considering a network topology where all the waypoints $w \in \mathbb{W}$ are removed:

$$\forall s, t \in V \setminus \mathbb{W}.\ \forall r \in N(s) \setminus \mathbb{W}.\ D(s, t, \mathbb{W}) \leq W(s, r) + D(r, t, \mathbb{W}) \tag{3}$$

Using these equations, $D(s, t, \mathbb{W})$ is upper bounded by the actual shortest non-waypoint distance from s to t.

Given a destination tree ξ_λ, we can use non-waypoint distances to enforce waypoint-compliance. Consider a router in r such the waypoint $w \in \mathbb{W}$ is downstream to r in ξ_λ—i.e., there exists a path from r to w in ξ_λ. The path from r must traverse through a waypoint in $\mathbb{W}$. If the sum of weights of the edges of the path in the tree $r \rightarrow^+_{\xi_\lambda} R_\lambda$ is strictly smaller than the non-waypoint distance from r to R_λ, then the path from r to R_λ is guaranteed to traverse a waypoint $w \in \mathbb{W}$. These constraints are expressed as:

$$\forall r' \in N(s) \setminus N_{\xi_\lambda}(r).\ \sum_{r \rightarrow^+_{\xi_\lambda} R_\lambda} W < W(r, r') + D(r', R_\lambda, \mathbb{W}) \tag{4}$$

These equations do *not* guarantee that router r will forward traffic to $N_{\xi_\lambda}(r)$. Consider the example in Figure 7(a) where traffic for λ must traverse through one of the waypoints in $\{r_1, r_2\}$. In this case, Genesis provided the path $s \rightarrow r_1 \rightarrow t$, but the shortest path is $s \rightarrow r_2 \rightarrow t$ which is waypoint-compliant.

Soundness and Completeness. With no static routes, if edge weights satisfy constraints (3) and (4), then paths for packet classes will traverse one of the waypoints. However, these constraints do not ensure completeness—i.e., a solution could be waypoint-compliant while violating some of the

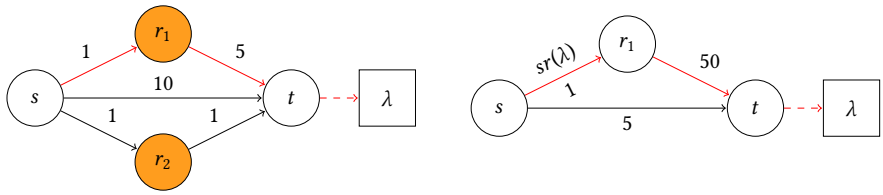


Fig. 7. Example of waypoint-compliant edge weights and example of a routing loop caused by a static route.

constraints, specifically constraint (4): while the weight of the Genesis path may be greater than the non-waypoint distance, the actual shortest path could be waypoint-compliant.

5.2.2 Avoiding Routing Loops.

Algorithm 1 presents the OSPF synthesis algorithm with static routes. If one of the constraints in Equation (4) is part of an unsatisfiable core, Zeppelin adds the static route $(r, N_{\xi_\lambda}(r))$ to $SR(\lambda)$ to eliminate the equations at router r. Constraints (4) do not guarantee that the path generated by Genesis is indeed the shortest path, so adding static routes can lead to undesired behaviors like routing loops. Consider the example configuration shown in Figure 7(b). If link $r_1 \longrightarrow t$ fails, traffic will oscillate between s and r_1 due to SR and OSPF and finally be dropped.

Whenever Zeppelin adds a static route to resolve an unsat-core, it removes the constraints pertaining to the static route and *adds* new constraints to prevent routing loops. Suppose Zeppelin added a static route (sr_1, sr_2) for destination λ where $sr_2 = N_{\xi_\lambda}(sr_1)$ (we do not add static routes on any other links except ξ_λ). Zeppelin needs to ensure that, for any router $r \in \xi_\lambda$ that does not lie in the downstream path $sr_2 \rightarrow^+_{\xi_\lambda} R_\lambda$, the shortest path from sr_2 to R_λ does not traverse through r. For e.g., in Figure 7(b), there is a routing loop because the shortest path from r_1 to t goes through upstream router s.

Formally, Zeppelin adds constraints to ensure that the weight of path $sr_2 \rightarrow^+ R_\lambda$ is strictly smaller than any path from sr_2 that traverses a router r' not in the downstream path from sr_2:

$$\forall r' \in \xi_\lambda.\ sr_2 \not\rightarrow^+_{\xi_\lambda} r'.\ \sum_{sr_2 \rightarrow^+ R_\lambda} W < D(sr_2, r') + D(r', R_\lambda) \tag{5}$$

If one of these constraints is part of an unsatisfiable core, then there is a routing loop caused from sr_2. Zeppelin rectifies this by adding a static route $(sr_2, N_{\xi_\lambda}(sr_2))$ to $SR(\lambda)$. Algorithm 1 describes the unsat-core learning algorithm for waypoint-compliance.

Soundness and Completeness. Algorithm 1 is sound—i.e., for each packet class, there is a path which goes to one of the waypoint in the set. Algorithm 1 is not complete—i.e., not guaranteed to find C with a given bound on number of static routes.

5.2.3 Configurations for Two Paths.

To guarantee policy-resilience, Zeppelin uses Genesis to generate two waypoint-compliant edge-disjoint paths π_1 and π_2 from s to t and adds the following constraints to ensure that the weights of both paths are strictly shorter than the non-waypoint distance for λ.

$$\sum_{\pi_1} W < D(s, t, \mathbb{W}) \ \wedge \ \sum_{\pi_2} W < D(s, t, \mathbb{W}) \tag{6}$$

Notice that the constraints do not enforce an order between the weights of π_1 and π_2. Since the paths are edge-disjoint, if a single link fails, at most one of the paths is affected. For a OSPF configuration with no static routes, all paths for a policy will be waypoint-compliant under failures.

Algorithm 1 OSPF Waypoint Synthesis with Static Routes

```
1:  procedure OSPF_W_SYNTH(Π)
2:      Ψ_D : Distance (1) and non-waypoint distance constraints (3)
3:      Ψ_W : Waypoint constraints for Π (4)
4:      Ψ_R = ∅ : Routing Loop avoidance constraints (5)
5:      Ψ = Ψ_D ∪ Ψ_W ∪ Ψ_R
6:      while Ψ is unsat do
7:          Extract unsat core uc from LP Solver
8:          Pick static route sr(sr_1, sr_2, λ) from uc constraints
9:          Add sr to static routes SR
10:         Ψ = Ψ \ (Ψ_W(sr_1, λ) ∪ Ψ_R(sr_1, λ))
11:         Ψ = Ψ ∪ Ψ_R(sr_2, λ)
12:     Obtain W from solution model of Ψ
13:     return Configuration C(W, SR)
```

Soundness and Completeness. If there are no static routes, OSPF weights satisfying constraints (3), (4) and (6) ensure waypoint-compliance under any arbitrary single link failure. For completeness, the same observations as in Section 5.2.1 also apply to this case.

5.2.4 Resilience with Static Routes.

Providing policy-resilience with static routes is challenging because static routes do not react to failures. When static routes are used, we relax our definition of waypoint-compliance to ensure at least one of the induced paths for the policy traverses the waypoint. [2] We now present our approach to providing policy-resilience with static routes.

Zeppelin only adds static routes on the paths obtained from Genesis. Consider the example in Figure 8 where $\pi_1 = s \rightarrow r_0 \rightarrow r_1 \rightarrow t$ and $\pi_2 = s \rightarrow r_2 \rightarrow t$ are two edge disjoint paths connecting the routers s to t. In this example, π_1 is the path selected by the configuration and it uses a static route for $r_0 \rightarrow r_1$. Ideally, we want to ensure that if any link on π_1 fails, then the network must switch over to π_2. When link $r_1 \rightarrow t$ fails, router s forwards to r_0 (path weight 5: $s \rightarrow r_0 \rightarrow r_2 \rightarrow t$) over r_2 (path weight 6: $s \rightarrow r_2 \rightarrow t$). Because of the static route at r_0, traffic is forwarded to r_1 which sends it back to r_0 through r_3, causing a routing loop.

The problem is caused by the fact that π_1—i.e., the path chosen by the configuration—contains a static route. Upon a failure, even though there exists another waypoint-compliant path, π_2, the configuration will not switch to it. To avoid this problem, Zeppelin synthesizes configurations that split traffic among the two paths.

However, simply adding the constraint $\sum_{\pi_1} W = \sum_{\pi_2} W$ does not ensure the traffic will be split (see Figure 8), due to the presence of static routes in the path. Intuitively, a static route is added to override OSPF and take a longer path, thus $\sum_{\pi_1} W$ is not the actual shortest path in the network. Thus, we need to estimate the weight of the shortest paths from the source corresponding to π_1 and π_2 and equate them to split the traffic.

In our running example, the distance between s to t corresponding to π_1 is equal to $W(s, r_0) + D(r_0, t) = 1 + (1 + 1 + 1)$ where r_0 is the position of the first static route. From s to r_0, since there are no static routes in the path, the OSPF distance can be estimated using the sum of edge weights, while the distance from r_0 to t is estimated using $D(r_0, t)$. Concretely, given $\pi_1 = (s, r_1)(r_1, r_2) \ldots (r_m, t)$, let r_α denote the first router in the path which has a static route—i.e., $\forall i < \alpha.(r_i, r_{i+1}) \notin SR(\lambda)$ for

[2] Can be realized by two mechanisms: 1) A router which has multiple routes for λ will split the traffic among these routes, and 2) The waypoint can mark certain fields in the packet header, and the network operator can employ simple edge-based checks at the destination (for e.g., in the hypervisor) to detect if a packet traversed the waypoint or not. Thus, only packets which traversed a waypoint-compliant path would be accepted and forwarded to the tenant.

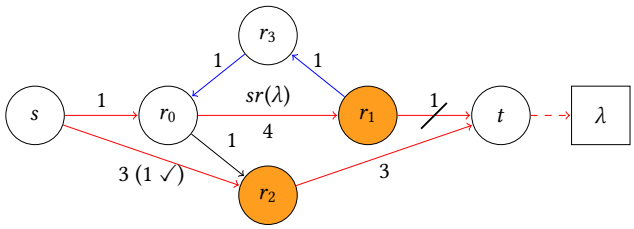


Fig. 8. Example non-resilient configuration for the red paths provided by Genesis for $\mathbb{W} = \{r_1, r_2\}$. If link $r_1 \rightarrow t$ fails, a routing loop is formed at $r_0 \rightarrow r_1 \rightarrow r_3 \rightarrow r_0$, no traffic reaches t. The configuration is 1-resilient waypoint-compliant when the $s \rightarrow r_2$ weight is set to 1.

destination IP λ. We estimate the actual shortest OSPF distance between s to t corresponding to π_1 as the sum of weights till r_α plus the distance from r_α to t.

To enable splitting amongst π_1 and $\pi_2 = (s, s_1)(s_1, s_2) \ldots (s_n, t)$, we would like to ensure that the estimated OSPF distance of π_2 (with first static route at s_β) is smaller or equal to the estimated OSPF distance corresponding to π_1, and vice-versa. A first attempt at doing so is to use the following constraint.

$$\sum_{(s,s_1)\ldots(r_{\beta-1},r_\beta)} W + D(r_\beta, t) \quad \leq \quad \sum_{(s,r_1)\ldots(r_{\alpha-1},r_\alpha)} W + D(r_\alpha, t) \tag{7}$$

By virtue of distance constraints (1), $D(r_\beta, t)$ is upper-bounded by the actual shortest distance between r_β and t. However, the distance constraints do not impose a lower bound on $D(r_\beta, t)$ and a constraint of the form $D(r_\beta, t) \leq E$, where E is some expression, will also not impose a lower bound on the value of $D(r_\beta, t)$. Instead this may result in lower D and incorrect W values. Hence, we need to use $\sum_{\pi_2} W$ (which is an upper bound) for the OSPF distance corresponding to π_2:

$$\sum_{\pi_2} W \quad \leq \quad \sum_{(s,r_1)\ldots(r_{\alpha-1},r_\alpha)} W + D(r_\alpha, t) \tag{8}$$

Since we cannot exactly express the weights of the actual routes seen at the source router, the above constraint does not guarantee that traffic will be split amongst π_1 and π_2. Similar to the routing loop avoidance constraints, we add the above constraints lazily when a static route is added to one of the paths for λ. However, if one of these constraints is part of an unsatisfiable core, we cannot add a static route to eliminate this unsatisfiability. Zeppelin lazily eliminates these constraints from the system of equations depending on whether these constraints are part of an unsat-core which cannot be eliminated by a static route. Thus, the presented approach does not provide provable resilience guarantees. However, our experiments show that Zeppelin generates highly resilient configurations (§7.2).

6 INCREASING RESILIENCE THROUGH DOMAIN ASSIGNMENT

Many networks today have well-defined routing hierarchies to realize a variety of administrative policies or division of responsibilities. For example, a campus network may be hierarchically divided into multiple routing domains (e.g., OSPF Areas) corresponding to different departments. Often, these hierarchies are realized through careful planning and require detailed configurations—e.g., to determine which routers to include in a particular OSPF routing domains, how many such domains to create, how big to make each domain etc. Unfortunately, this need for careful planning causes the hierarchical division to be painstaking and error-prone, and makes changes to the hierarchies—e.g., to accommodate more hosts or new policies—difficult to implement.

In such settings, Zeppelin helps by automating domain creation/restructuring. It can automatically find good routing domain hierarchies with increased resilience and policy compliance. We

envision that operators will synthesize "one-shot" domain assignments for their changing input policies at coarse-grained timescales (days/weeks) when there are significant changes to their networks or policies; operators need not re-synthesize assignments for low-level policy changes.

In this section, we present an algorithm that searches the space of possible domain assignments Θ to find one that meets all configuration policies and increases connectivity-resilience by reducing the number of static routes. Note that a simple variant of this problem: finding a domain assignment with zero static routes is NP-complete (Theorem A.3). Therefore, we opt for a greedy stochastic search. Zeppelin uses Markov Chain Monte Carlo (MCMC) sampling methods, specifically the Metropolis-Hasting algorithm, a common technique used in several optimization problems [28].

6.1 Searching Assignments with MCMC

MCMC sampling is a technique for drawing elements from a probability density function in direct proportion to its value. In our setting, MCMC searches the space of domain assignments and, if we assign higher probabilities to domain assignments with lower cost, MCMC will explore good configurations more *often* than bad ones. For MCMC to work we need to provide a transition function that lets us move from one domain assignment to another with a certain probability and a cost function that assigns costs (and therefore probabilities) to domain assignments.

In our setting, each domain assignment Θ has an associated cost $c(\Theta, e)$ where $e = expr(sc, bc)$ is the expression we are trying to minimize—e.g., if we are trying to minimize the number of static routes $e = sc$. The search starts by setting the current domain assignment to some random domain Θ_0. The following process then repeats. Given the current domain assignment Θ, compute a new domain assignment Θ' by randomly moving a gateway router r from one domain to another—i.e., $\Theta(r) \neq \Theta'(r)$ and for every $r' \neq r$, $\Theta(r') = \Theta'(r')$. If $c(\Theta', e) \leq c(\Theta, e)$, then Θ' becomes the current domain assignment. If $c(\Theta', e) > c(\Theta, e)$, then Θ' becomes the current domain assignment with probability $Pr(\Theta \rightarrow \Theta') = exp(-\beta \times (c(\Theta', e) - c(\Theta, e))$ (where β is a positive constant) while Θ continues being the current domain assignment with probability $1 - Pr(\Theta \rightarrow \Theta')$.

The algorithm always accepts a new proposal Θ' that has cost lower than Θ. If Θ' has a higher cost than Θ, the proposal is accepted with probability inversely proportional to how far the costs of Θ and Θ' are. This ensures that the algorithm does not get stuck at local minima, but explores proposals with small cost differences with higher probability.

6.2 The Cost of a Domain Assignment

We now look at how the cost $c(\Theta, e)$ is computed; this amounts to substituting in $e = expr(sc, bc)$ the values of sc and bc for the assignment Θ. The techniques presented in Section 4.3 provide a way to synthesize BGP configurations and inter-domain static routes for a given domain assignment and can be used to efficiently compute the quantities bc. However, given a domain assignment, computing the minimal number of intra-domain static routes is hard. Since we want MCMC to explore as many assignments as possible, we present a heuristic technique for estimating the number of static routes.

Given two paths π and π' for destinations λ and λ', we define these paths form a diamond if these paths intersect at two routers (r_1 and r_2) without any common router in between. If the paths π and π' completely lie in the same domain, the presence of a diamond implies that there are two different shortest paths between r_1 and r_2, which means that at least one static route is required. On the other hand, if r_1 and r_2 lie in different domains, no static routes are required to resolve this diamond. Before starting the MCMC search, Zeppelin precomputes the set of all diamonds induced by the paths Π. For each domain assignment Θ, Zeppelin estimates the number of intra-domain static routes by counting the number of diamonds formed by every pair of paths which lie inside the same domain.

While diamonds in the same domain definitely require static routes, there might be sets of paths that do not contain diamonds but still require static routes; these are not taken into account in our estimate. However, our diamond-based estimate can be computed efficiently and our experiments show that reductions in cost lead to lesser static routes and increased connectivity-resilience (§7.3).

7 EVALUATION

We implemented a prototype of Zeppelin in Python. For a given workload, Zeppelin outputs Quagga template configurations [25]. We use the Gurobi solver [17] for linear constraints generated by Zeppelin. In this section, we evaluate Zeppelin using enterprise-scale data center fat-tree topologies [2] of different sizes. Experiments were conducted on a 32-core Intel-Xeon 2.40GHz CPU machine and 128GB of RAM. Specifically, we ask the following questions.

Q1: How does Zeppelin perform on different workloads? (§7.1)
Q2: How resilient are Zeppelin's configurations? (§7.2)
Q3: Can Zeppelin get higher resilience by trying different domain assignments? (§7.3)
Q4: How does Zeppelin compare with the state-of-the-art? (§7.4)

7.1 Single Domain End-to-end Performance

We evaluate the end-to-end performance of Zeppelin on a fat-tree topology with 45 routers and a single OSPF domain. The size of the topology is consistent with operator preferences to restrict the size of a domain to under 50 routers. PC refers to the algorithm that handles complex policies and synthesizes policy-compliant configurations with few static routes (§4) and 1-WC (resp. 2-WC) refers to the algorithm that handles waypoint policies and can synthesize policy-compliant configurations with high policy-resilience from 1 path (resp. 2 paths) per waypoint policy (§5).

We consider two classes of policies. We evaluate algorithm PC to handle simple reachability policies and complex isolation policies that existing tools cannot handle; Our second class consists of waypoint policies of the form described in §5; for this class we evaluate 1-WC and 2-WC. For each algorithm we report the time for Genesis to generate policy-compliant forwarding paths and time Zeppelin takes to generate configurations from these paths. We set a timeout of 2,000 seconds for each workload.

Reachability Policies. We generate reachability policies (each of which corresponds to a path) with randomly generated endpoints. Figure 9(a)(a) shows the synthesis time for PC for varying number of paths. For these workloads, Zeppelin can synthesize configurations for 80 policies in 62 seconds, out of which configuration synthesis from paths takes 52 seconds on average.

Isolation Policies. We generate policies to ensure tenant-isolation in a multi-tenant topology, i.e., each tenant's traffic will be isolated from traffic from other tenants, and a tenant cannot interfere with any other tenant. This is achieved by adding isolation policies amongst different tenant's traffic. We have n tenant groups (between 1 to 4), each group comprised of g destinations (20). Figure 9(a)(b) shows the synthesis time for PC. The x-axis shows the paths $n * g$.

As the number of tenants increase, time to synthesize the data plane increases exponentially because finding isolated paths is NP-complete. Also, these paths are longer than if we only had reachability policies. Similarily, time taken to synthesize OSPF configurations increases exponentially with the number of tenants as PC needs to add more static routes and requires more iterations of the unsat-core learning procedure. For this workload, Zeppelin can synthesize configurations for 4 tenants, each with 20 destinations in 507 seconds, where Genesis takes 217 seconds for synthesizing paths. Note that synthesizing configurations is harder for isolation than reachability because of longer paths provided by Genesis.

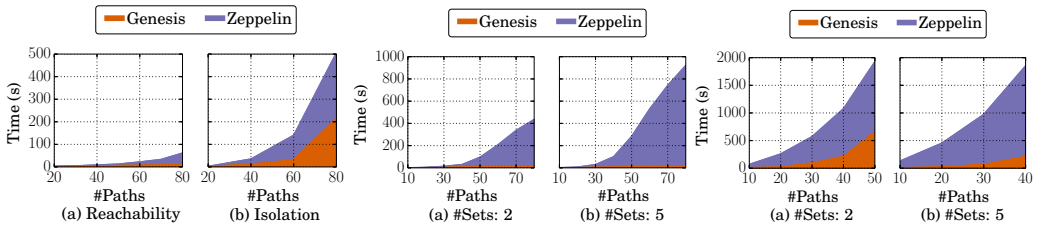


Fig. 9. End-to-end synthesis time for waypoint policy workloads for varying number of paths and different number of waypoint sets.

Waypoint Policies. We generate waypoint policy workloads with varying number of destination subnets and waypoint sets as follows. We consider workloads with 2 and 5 unique waypoint sets where each set contains 3 randomly picked routers. Each waypoint set can be viewed as a class of replicated middleboxes. We randomly generate policies that map each subnet to one of the waypoint sets.

Figure 9(b) and (c) show the end-to-end synthesis time for varying number of destination paths and the number of unique waypoint sets (2 and 5). Since synthesizing paths for waypoint policies can be done independently, Genesis synthesis time is small compared to OSPF synthesis. The synthesis times of 1-WC and 2-WC increase with the number of waypoint sets because the algorithms need to add constraints corresponding to $D(s, t, \mathbb{W})$ for each set. Hence, computing the unsat-core in each iteration to find static routes is more expensive. For 5 waypoint sets and 40 destinations, Zeppelin takes 120 seconds on average to synthesize waypoint-compliant configurations and 1800 seconds to synthesize policy-resilient configurations.

Q1: Zeppelin can synthesize policy-compliant configurations for medium size topologies in less than 10 minutes and policy-resilient configurations in less than an hour.

7.2 Resilience of Intra-domain Configurations

Connectivity-resilience. We measure the connectivity-resilience of the configurations generated using the algorithm PC on the reachability workload described in Section 7.1. For our baseline, we use configurations where paths are enforced using only static routes and OSPF weights are assigned randomly. Figure 10 shows the results. A point above the diagonal denotes a benchmark in which Zeppelin generated a configuration with higher connectivity resilience than the baseline algorithm. PC provides an average connectivity-resilience score of 0.97 over the baseline score of 0.88. Moreover, PC places on average 9.3% of the static routes than required by the baseline.

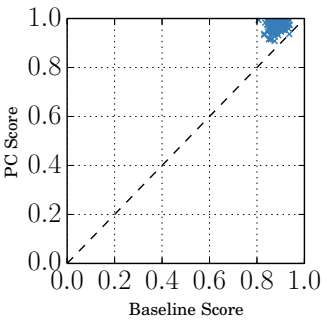
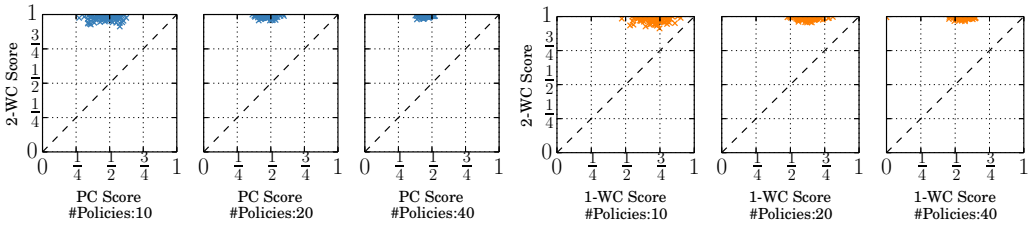


Fig. 10. Connectivity-resilience: PC vs baseline.

Policy-resilience. We measure the policy-resilience of the configurations generated using algorithms 1-WC and 2-WC on the waypoint workload described in §7.1 with 5 waypoint sets. For each workload, we run each algorithm 3 times with a different random seed, and report the most-resilient configuration it obtains. For our baseline, we use the configurations generated by the algorithm PC, which only tries to reduce the number of static routes.



(a) 2-WC vs. PC (b) 2-WC vs. 1-WC

Fig. 11. Policy-resilience scores of 2-WC, 1-WC, and PC synthesis for varying waypoint workloads.

Figure 11 shows the results for varying number of policies (10, 20 and 40), each policy mapped to a particular destination IP address. For the plot on the left (resp. right) a point above the diagonal denotes a benchmark in which 2-WC generated a configuration with higher policy-resilience than PC (resp. 1-WC). For 40 policies, 2-WC can synthesize configurations with high policy-resilience (average 0.98) over PC (average 0.41) and 1-WC (average 0.51). Although not as good as 2-WC, 1-WC is able to provide higher policy-resilience than PC thanks to the relaxed constraints that do not require the path given by Genesis to be the shortest one.

Q2: Zeppelin **can synthesize highly resilient configurations**.

7.3 Dynamic Domain Assignment Performance

In this section, we measure whether when Zeppelin is allowed to explore different domain assignments, the MCMC algorithm presented in §6 can generate configurations with higher resilience. We consider an 80 router fat-tree topology and run the MCMC sampling for 600s—i.e., more than 100,000 iterations—to minimize the cost function $max(sc, bc)$. Using this cost function, Zeppelin tries to jointly decrease number of BGP local preferences and static routes. For the input, we generate n (between 200 and 1,000) random paths for $n/4$ destination IPs, with random path length between 3 and 10. We require Zeppelin to split the network into 5 OSPF domains each with size in range between 4 and 10. We conduct each experiment 20 times and report averages and standard deviations. For each MCMC run, we collect the domain assignments with worst and best scores. For these assignments, we use the algorithm PC to compute path-compliant configurations. Figure 12 shows, for varying number of paths, the average ratios between the best- and worst-score configurations in terms of BGP configuration overhead, number of static routes, and connectivity-resilience. For smaller workloads, the algorithm reduces the BGP configuration overhead, but increases static routes, leading to worse connectivity-resilience. For larger workloads, the algorithm can reduce both static routes and BGP local preference entries by 0.3× and improve the connectivity-resilience by 0.1×.

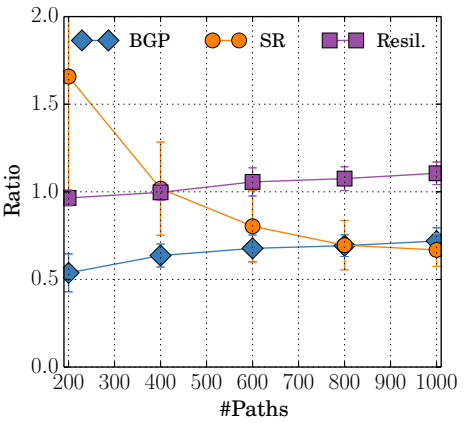


Fig. 12. MCMC Evaluation for varying number of paths.

Topology	#Classes	SyNET	Zeppelin	Speedup
Internet2	1	9.0s	0.02s	**450×**
	5	21.3s	0.03s	**710×**
	10	49.3s	0.03s	**1,643×**
G3	1	9.4s	0.03s	**313×**
	5	19.8s	0.03s	**660×**
	10	39.9s	0.03s	**1,330×**

Fig. 13. Comparison of synthesis times of Zeppelin and SyNET for OSPF + Static route workloads.

Q3: When allowed to try different domain assignments, Zeppelin **synthesizes configurations with higher connectivity-resilience mainly for workloads with large number of paths**.

7.4 Comparison to SyNET

SyNET [12] is a network-wide configuration synthesis system which supports multiple routing protocols (OSPF and BGP) and static routes. In this section, we compare SyNET's performance with Zeppelin for OSPF + static route configuration synthesis workloads obtained from the authors of SyNET.

For these experiments, we use the two network topologies the SyNET's authors made available: (1) Internet2: a US-based network with 9 routers, and (2) G3: A 3×3 grid topology. The routing requirements define the forwarding behavior for 1, 5, and 10 traffic classes as follows: for each router and traffic class the workload specifies the next-hop router and the protocol for forwarding: OSPF or static route. Thus, for a topology with n routers and m traffic classes, the workload contains $n \times m$ SyNET requirements. We combine these requirements to generate paths to provide as input for Zeppelin's OSPF synthesis. Since the workload specifies which routers had static routes installed for the different classes, we add these static routes (and remove the corresponding constraints), and use the algorithm presented in Section 4.2.1 to find the OSPF weights.

Figure 13 shows Zeppelin outperforms SyNET for all the workloads and topologies, achieving a best case 1,643× speedup for Internet2 topology with 10 traffic classes. In each of these experiments, Zeppelin invokes the LP solver once and does not need to add more static routes than specified in the input requirements, which matches with the SyNET output (if more static routes were required, SyNET would have returned no solution for these experiments).

Q4: Zeppelin **is able to achieve 2-3 orders of magnitude speedup over state-of-the-art tools**.

8 RELATED WORK

Centralized control. RCP [9], supported logically central BGP configuration. The more recent Fibbing [31] system provides centralized control over distributed routing by creating fake nodes and fake links to steer the traffic in the network through paths that are not the shortest, similar to how Zeppelin uses static routes. Both approaches face the issue of forwarding loops in the network during failures. Fibbing and Zeppelin differ in the specific algorithms to solve the synthesis problem. Zeppelin does offer key advantages: first, by using static routes for steering, we do not increase the control traffic in the network, unlike the fake advertisements used in Fibbing. Second, Fibbing by design targets a single domain and does not take into account domain decomposition and inter-domain routing.

Configuration synthesis. ConfigAssure [22] uses a combination of logic programming and SAT solving to synthesize network configurations for security, functionality, performance and reliability requirements specified as constraints; but it does not support any notion of policy- or connectivity-resilience or hierarchical domain splitting. Fortz et. al [13] tackle the problem of optimizing OSPF

weights for performing traffic engineering, but their work is tailor-made to just this specific problem. Propane [4, 5] tackles the specific problem of synthesizing BGP configurations for concrete and abstract topologies to ensure network-wide objectives hold even under failures. Propane is suited to specify preferences on paths and peering policies among different autonomous systems. Propane translates policies to a graph-based intermediate representation, which is then compiled to device-level BGP configurations. The automata-based compilation and resilience algorithms for Propane are tailored for its underlying technology (BGP configurations with support for local preferences, MEDs and communities), in contrast to our LP-based algorithms which synthesize hierarchical BGP, OSPF and static routing configurations which have a different forwarding model than BGP.

SyNET [12] tackles network-wide configuration synthesis (Definition 3.2) by modeling the behavior and interactions of the routing protocols as a stratified Datalog program, and using SMT to synthesize the Datalog input such that the fixed point of the Datalog program (which represents the network's forwarding state after convergence) satisfies certain policies or path requirements. While both systems can take paths as input requirements, Zeppelin uses a two-phase approach where OSPF weights are synthesized separately using LP-solvers—which are faster and parallelizable—rather than directly solving the whole configuration synthesis problem using SMT solvers. Moreover, SyNET's approach does not deal with resilience, a key aspect we tackle in this paper, and SyNET does not attempt to minimize the number of static routes, which can cause undesirable behaviors like routing loops. Finally, SyNET supports routers that run both OSPF and BGP protocols and can be configured with static routes, which does not fit naturally into a hierarchical structure where some routers only run OSPF and not BGP. In contrast, our stochastic MCMC algorithm can look for dynamic domain assignments with constraints on OSPF domain sizes, lower BGP configuration complexity and higher resilience.

Policy languages. Zeppelin uses the Genesis [30] framework to synthesize paths which comply with reachability, waypoint and isolation policies. Genesis is tailored for software-defined networks comprised of OpenFlow [21] switches, whose forwarding tables can be directly programmed to specify the next-hop switch for different classes of traffic based on the paths provided by Genesis. Zeppelin's algorithms cater for legacy control planes running OSPF and BGP which lack the programmability of SDNs. Genesis also provides resilience in a different manner than Zeppelin. Genesis precomputes and installs backup paths (generated using isolation policies) in the SDN dataplane, while Zeppelin synthesizes OSPF weights and static routes such that, the distributed protocols converge to a policy-compliant path even under failures. Unlike Genesis, which can provide resilience for arbitrary k-link failures, Zeppelin currently supports only $k = 1$ link failure scenarios.

In the future, Zeppelin could use other policy language frameworks as a front-end (with less rich policy support but better performance). In Merlin [29], data planes that adhere to policies expressed using regular expressions and min and max bandwidth guarantees are synthesized using mixed integer linear programming (ILP). NetKAT [3] is a domain-specific language and logic for specifying and verifying network packet-processing functions for SDN, based on Kleene algebra with tests (KAT). NetKAT can express certain network-wide policies like reachability and waypoints. However, both Merlin and NetKAT do not support link-isolation to produce edge-disjoint paths, which is needed for waypoint-compliance in §5.

9 CONCLUSION

We presented Zeppelin, a system for automatically synthesizing highly-resilient distributed hierarchical control planes—i.e., OSPF and BGP router configurations—from high-level policies. For hierarchical control planes, we devised algorithms based on our OSPF-BGP interaction model. In practice, there exists other hierarchical models, these routing models can be integrated into

Zeppelin's modular two-phase approach. Similarly, we can modify OSPF synthesis to incorporate multi-path routing for load-balancing etc. Finally, resilience is an important requirement, and extending Zeppelin to support other notions of policy-resilience for $k > 1$ link failures is subject for future work.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Brighten Godfrey, Aaron Gember-Jacobson, Raajay Viswanathan, and Shuchi Chawla for their insightful feedback and suggestions. Kausik, Loris and Aditya are supported by the Wisconsin Institute on Software-defined Datacenters of Madison and grants from Google and National Science Foundation (CCF-1637516, CNS-1302041, CNS-1330308, CNS-1345249).

REFERENCES

[1] Aditya Akella and Arvind Krishnamurthy. 2014. A Highly Available Software Defined Fabric. In *HotNets*.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, Vol. 38. ACM, 63–74.

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

[4] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitu Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proceedings of the ACM SIGCOMM 2016 Conference on SIGCOMM (SIGCOMM '16)*.

[5] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2017. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 437–451.

[6] Theophilus Benson, Aditya Akella, and David Maltz. 2009. Unraveling the Complexity of Network Management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. USENIX Association, Berkeley, CA, USA, 335–348. http://dl.acm.org/citation.cfm?id=1558977.1559000

[7] Theophilus Benson, Aditya Akella, and Aman Shaikh. 2011. Demystifying Configuration Challenges and Trade-offs in Network-based ISP Services. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. ACM, New York, NY, USA, 302–313. https://doi.org/10.1145/2018436.2018471

[8] Peter Broström and Kaj Holmberg. 2009. Compatible weights and valid cycles in non-spanning OSPF routing patterns. *Algorithmic Operations Research* 4, 1 (2009), 19–35.

[9] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. 2005. Design and Implementation of a Routing Control Platform. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05)*. USENIX Association, Berkeley, CA, USA, 15–28. http://dl.acm.org/citation.cfm?id=1251203.1251205

[10] John W Chinneck. 2007. *Feasibility and Infeasibility in Optimization:: Algorithms and Computational Methods*. Vol. 118. Springer Science & Business Media.

[11] Cisco. 2013. What Is Administrative Distance. http://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/15986-admin-distance.html. (2013).

[12] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2017. Network-wide Configuration Synthesis. In *29th International Conference on Computer Aided Verification, Heidelberg, Germany, 2017 (CAV'17)*.

[13] Bernard Fortz and Mikkel Thorup. 2000. Internet traffic engineering by optimizing OSPF weights. In *INFOCOM 2000. Nineteenth annual joint conference of the IEEE computer and communications societies. Proceedings. IEEE*, Vol. 2. IEEE, 519–528.

[14] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 163–174. https://doi.org/10.1145/2619239.2626313

[15] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. 2015. Management Plane Analytics. In *IMC*.

[16] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. ACM, New York, NY, USA, 350–361. https://doi.org/10.1145/2018436.2018477

[17] Gurobi. 2017. Gurobi Optimization. http://www.gurobi.com/. (2017).
[18] Victor Heorhiadi, Michael K Reiter, and Vyas Sekar. 2016. Simplifying software-defined network optimization using SOL. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 223–237.
[19] Ratul Mahajan, David Wetherall, and Thomas E. Anderson. 2002. Understanding BGP misconfiguration. In *SIGCOMM*.
[20] David A. Maltz, Geoffrey Xie, Jibin Zhan, Hui Zhang, Gísli Hjálmtýsson, and Albert Greenberg. 2004. Routing Design in Operational Networks: A Look from the Inside. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '04)*. ACM, New York, NY, USA, 27–40. https://doi.org/10.1145/1015467.1015472
[21] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74. https://doi.org/10.1145/1355734.1355746
[22] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. 2008. Declarative Infrastructure Configuration Synthesis and Debugging. *J. Netw. Syst. Manage.* 16, 3 (Sept. 2008), 235–258. https://doi.org/10.1007/s10922-008-9108-y
[23] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: a framework for NFV applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 121–136.
[24] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2486001.2486022
[25] Quagga. 2017. Quagga Routing Suite. http://www.nongnu.org/quagga/. (2017).
[26] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. 2013. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 109–114.
[27] Yakov Rekhter, Tony Li, and Susan Hares. 2005. *A border gateway protocol 4 (BGP-4)*. Technical Report.
[28] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150
[29] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. 2014. Merlin: A Language for Provisioning Network Resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. ACM, New York, NY, USA, 213–226. https://doi.org/10.1145/2674005.2674989
[30] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2017. Genesis: Synthesizing Forwarding Tables for Multi-tenant Networks. In *POPL*. ACM.
[31] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. 2015. Central Control Over Distributed Routing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 43–56. https://doi.org/10.1145/2785956.2787497
[32] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. 2015. Scenario-based Programming for SDN Policies. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '15)*. ACM, New York, NY, USA, Article 34, 13 pages. https://doi.org/10.1145/2716281.2836119
[33] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. 2012. *A Survey on Network Troubleshooting*. Technical Report TR12-HPNG-061012. Stanford University.

A PROOFS

Theorem A.1 (Hardness of synthesis). *Given a network with a single domain, and a positive number k, the problem generating a path-compliant configuration with at most k static routes is NP-complete.*

Proof. We show that the decision version of the minimum vertex cover problem, i.e., there exists a vertex cover of size $\leq k$, which is NP-complete, reduces to finding a set of static routes of size $\leq k$ and OSPF weights for a network with only one domain. The latter is also in NP, so after the reduction we can conclude that it is also NP-complete.

Let $G = (V, E)$ be an instance of the minimum vertex cover problem. A set of vertices $VC \subseteq V$ is the vertex cover if $\forall (v_1, v_2) \in E.\ v_1 \in VC \vee v_2 \in VC$.

We now show how to construct a topology $T = (R, L)$ and a corresponding set of paths Π that can be enforced by configuration C which requires static routes SR such that $|SR| \leq k$ if and only if the corresponding $VC(SR)$ is a vertex cover of the graph G and $|VC(SR)| \leq k$.

Construction. For every vertex $v \in V$: add a vertex r_v. For every edge $(u, v) \in E$: add two vertices s_{uv} and t_{uv} to R. Add edges connecting $s_{uv} \rightarrow r_u$, $s_{uv} \rightarrow r_v$, $r_u \rightarrow t_{uv}$ and $r_v \rightarrow t_{uv}$. Figure 14 illustrates this construction.

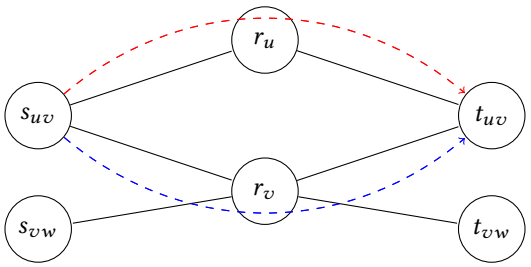


Fig. 14. Construction for reduction to Vertex Cover.

If there is another edge $(v, w) \in E$, then s_{vw} and t_{vw} have an edge connecting to r_v (shown in Figure 14).

For each edge $(u, v) \in E$, we add two paths in Π: $s_{uv} \rightarrow r_u \rightarrow t_{uv}$ for destination host d_u and $s_{uv} \rightarrow r_v \rightarrow t_{uv}$ for destination host d_v. (dashed paths in Figure 14).

We now prove that if there exists a set of static routes SR such that $|SR| \leq k$ such that the resulting configurations are path-compliant for Π, then there exists a vertex cover VC of G such that $|VC| \leq k$.

For each static route $sr \in SR$, the static route has to be placed at the source, either going to r_u or r_v (structure of topology T). We construct a set $VC(SR)$ by adding the vertex v based on the endpoints of each static route $sr \in SR$. To show that $VC(SR)$ is a vertex cover of G, we first prove Lemma A.2.

Lemma A.2. *For each diamond formed by the input paths, atleast 1 static route on one of the edges of the paths of the diamond is required to find a valid solution to the OSPF edge weights.*

Proof. Given two paths π_1 and π_2 for destinations d_u and d_v, we define these paths form a diamond if these paths intersect at two routers (s_{uv} and t_{uv}) without any common router in between. Consider the following diamond constructed by paths π_1: $s_{uv} \rightarrow r_u \rightarrow t_{uv}$ for destination d_u and π_2: $s_{uv} \rightarrow r_v \rightarrow t_{uv}$ for destination d_v. Let us assume there exists a solution for the OSPF edge weights without any static routes.

We add the following inequality to make π_1 is the shortest path from s_{uv} to t_{uv} by ensuring π_1 is shorter than the path from s_{uv} to t_{uv} via r_v:

$$W(s_{uv}, r_u) + W(r_u, t_{uv}) < W(s_{uv}, r_v) + W(r_v, t_{uv}) \tag{9}$$

Since π_2 is also the shortest path from s_{uv} to t_{uv}, the linear inequality added is:

$$W(s_{uv}, r_v) + W(r_v, t) < W(s_{uv}, r_u) + W(r_u, t_{uv}) \tag{10}$$

Since there are no static routes on the edges of π_1 and π_2, none of the above equations are eliminated. Adding equations 9 and 10 yields the inequality $0 < 0$, which is inconsistent and therefore, no solution to the edge weights exists for this system of equations, which contradicts our assumption. Therefore, for each diamond formed by the input paths, atleast 1 static route on one of the edges of the paths of the diamond is required to find a valid solution to the OSPF edge weights. □

For every edge $(u, v) \in E$, the constructed paths from s_{uv} to t_{uv} form a diamond. Thus, by Lemma A.2, the diamond created by the paths corresponding to each edge in G requires atleast one static route to eliminate the inconsistency caused by the diamond. If a static route's endpoints contains r_u, we put u in $VC(SR)$ and similarily for r_v. Edge (u, v) is covered since atleast one static route is added, thus, atleast one of $\{u, v\}$ is in $VC(SR)$. Thus, if SR eliminates all diamond inconsistencies to find a solution to the OSPF weights, the corresponding set $VC(SR)$ covers all edges in E. Therefore, $VC(SR)$ is a vertex cover.

Thus, by finding a set of static routes SR such that $|SR| \leq k$ such that all the diamond inconsistencies are eliminated, and there exists OSPF weights W such that the configurations forward traffic along Π, we can find a vertex cover VC for graph G such that $|VC| \leq k$.

This transformation is polynomial, the constructed network topology T has $|V|+2|E|$ nodes, $4|E|$ links and $2|E|$ paths. Therefore, OSPF configuration synthesis with number of static routes $\leq k$ is NP-complete. Thus, OSPF synthesis with minimal number of static routes is NP-hard. □

Theorem A.3 (Hardness of domain assignments). *Given a network and set of paths Π, the problem of generating a domain assignment for which there exists a configuration that is path-compliant with Π with no static routes is NP-complete.*

Proof. We show the k-graph coloring problem, which is NP-complete reduces to finding a domain assignment such that the number of static routes is 0. The latter is also in NP, so after the reduction we can conclude that it is also NP-complete.

Let $G = (V, E)$ be an instance of the k-graph coloring problem. Formally, we need to find a coloring $C : V \mapsto \{1, 2, \ldots k\}$ such that for $\forall (u, v) \in E. C(u) \neq C(v)$.

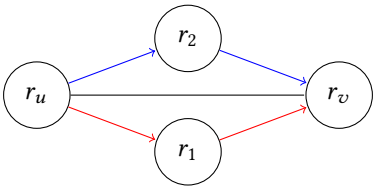


Fig. 15. Construction for reduction to Graph Coloring.

Construction. Let us consider a network topology $T = (R, L)$. For each $v \in V$, add a router $r_v \in R$. To ensure any domain assignment to the r_v's is valid, we add link to connect every router (each domain must be contiguous).

For every edge $(u, v) \in E$, we add r_1, r_2 to R, and add paths: $r_u \rightarrow r_1 \rightarrow r_v$ and $r_u \rightarrow r_2 \rightarrow r_v$ with different destinations to Π. We can notice that these two paths form a diamond.

Suppose we find a domain assignment Θ with k domains such that the number of static routes used is zero. Since, the count of static routes is 0, for two routers r_u, r_v which have a diamond, $\Theta(r_u) \neq \Theta(r_v)$. This is because, if $\Theta(r_u) = \Theta(r_v)$, then atleast one static route would be required to eliminate the diamond (Lemma A.2). Therefore, for every edge in $(u, v) \in E$, the routers r_u and r_v belong to different domains, therefore u and v have different colors. There, the k-graph coloring problem reduces to finding a domain assignment with zero static routes. Therefore, the path-compliance synthesis problem is NP-complete. □

Received November 2017, revised January 2018, accepted March 2018.