Tiramisu: Fast Multilayer Network Verification

Anubhavnidhi Abhashkumar, *University of Wisconsin - Madison;*
Aaron Gember-Jacobson, *Colgate University;* Aditya Akella,
University of Wisconsin - Madison

https://www.usenix.org/conference/nsdi20/presentation/abhashkumar

This paper is included in the Proceedings of the
17th USENIX Symposium on Networked Systems Design
and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Open access to the Proceedings of the
17th USENIX Symposium on Networked
Systems Design and Implementation
(NSDI '20) is sponsored by

Tiramisu: Fast Multilayer Network Verification

Anubhavnidhi Abhashkumar*, Aaron Gember-Jacobson†, Aditya Akella*
*University of Wisconsin - Madison**, *Colgate University*†

Abstract: Today's distributed network control planes are highly sophisticated, with multiple interacting protocols operating at layers 2 and 3. The complexity makes network configurations highly complex and bug-prone. State-of-the-art tools that check if control plane bugs can lead to violations of key properties are either too slow, or do not model common network features. We develop a new, general multilayer graph control plane model that enables using fast, property-customized verification algorithms. Our tool, Tiramisu can verify if policies hold under failures for various real-world and synthetic configurations in $<$ 0.08s in small networks and $<$ 2.2s in large networks. Tiramisu is 2-600X faster than state-of-the-art without losing generality.

1 Introduction

Many networks employ complex topologies and distributed control planes to realize sophisticated network objectives. At the topology level, networks employ techniques to virtualize multiple links into logically isolated broadcast domains (e.g., VLANs) [8]. Control planes employ a variety of routing protocols (e.g., OSPF, eBGP, iBGP) which are configured to exchange routing information with each other in intricate ways [9, 21]. Techniques to virtualize the control plane (e.g., virtual routing and forwarding (VRF)) are also common [8].

Bugs can easily creep into such networks through errors in the detailed configurations that the protocols need [9, 21]. In many cases, bugs are triggered when a failure causes the control plane to reconverge to new paths. Such bugs can lead to a variety of catastrophic outcomes: the network may suffer from reachability blackholes [5]; services with restricted access may be rendered wide open [4]; and, expensive paths may be selected over inexpensive highly-preferred ones [4].

A variety of tools attempt to verify if networks could violate important policies. In particular, *control plane analyzers* [6, 12–14, 23, 29] proactively verify if the network satisfies policies against various environments, e.g., potential failures or external advertisements. State-of-the-art examples include: graph-algorithm based tools, such as ARC [14] which models all paths that may manifest in a network as a series of weighted digraphs; satisfiability modulo theory (SMT) based tools, such as Minesweeper [6] which models control planes by encoding routing information exchange, route selection, and failures using logical constraints/variables; and, explicit-state model checking (ESMC) based tools, such as Plankton [23][1] which models routing protocols in a custom language, such that an explicit state model checker can explore the many possible data plane states resulting from the control plane's execution.

Unfortunately, these modern control plane tools still fall short because they make a hard trade-off between performance and generality (§2). ARC abstracts many low level control plane details which allows it to leverage polynomial time graph algorithms for verification, offering the best performance of all tools. But the abstraction ignores many network design constructs, including commonly-used BGP attributes, and iBGP. While these are accounted for by the other classes of tools [6, 23] that model control plane behavior at a much lower level, the tools' detailed encoding renders verification performance extremely poor, especially when exploring failures (§8). Finally, all existing tools ignore VLANs, and VRFs.

This paper seeks a fast general control plane verification tool that also accounts for layer 2.5 protocols, like VLANs.

We note that today's trade-off between performance and generality is somewhat artificial, and arises from an unnatural coupling between the control plane encoding and the verification algorithm used. For example, in existing graph-based tools, graph algorithms are used to verify the weighted digraph control plane model. In SMT-based tools, the detailed constraint-based control plane encoding requires a general constraint solver to be used to verify any policy. ESMC-based tools' encoding forces a search over the many possible data plane states, mirroring software verification techniques that exhaustively explore the execution paths of a general program.

The key insight in our framework, Tiramisu, is to decouple encoding from verification algorithms. Tiramisu leverages a *new, rich encoding* for the network that models various control plane features and network design constructs. The encoding allows Tiramisu to use different *custom verification algorithms* for different *categories of policies* that substantially improve performance over the state-of-the-art.

Tiramisu's network model uses graphs as the basis, similar to ARC. However, the graph model is multi-layered and uses multi-attribute edge weights, thereby capturing dependencies among protocols (e.g., iBGP depending on OSPF-provided paths) and among virtual and physical links, and accounting for filters, tags, and protocol costs/preferences.

For custom verification, Tiramisu notes that most policies studied in the literature can be grouped into three categories (Table 1): (i) policies that require the actual path that manifests in the network under a given failure; (ii) policies that care about certain quantitative properties of paths that may manifest (e.g., maximum path length); and, finally, (iii) policies that merely care about whether a path exists. Tiramisu leverages the underlying model's graph structure to develop performant verification algorithms for each category.

To verify category (i) policies, Tiramisu efficiently solves the stable paths problem [15] using the *Tiramisu Path Vector Protocol* (*TPVP*). *TPVP* simulates control plane computa-

[1] Plankton was developed contemporaneously with our system

tions across multiple devices and interdependent protocols by operating on the Tiramisu network model. *TPVP*'s domain-specific path computation is faster than the general search strategies used in SMT solvers, making Tiramisu orders of magnitude faster in verifying category (i) policies. *TPVP* also outperforms ESMC-based tools, because *TPVP*'s use of routing algebra [16, 25] atop the Tiramisu rich graph allows it to compute paths in one shot, whereas ESMC-based tools emulate protocols, and explore their state, one at a time in order to account for inter-protocol dependencies.

For category (ii), Tiramisu's insight is to use integer linear program (ILP) formulations that only model the variables that are relevant to the policy being verified. Tiramisu significantly outperforms SMT- and ESMC-based tools, whose computation of specific paths requires them to (needlessly) explore a much larger variable search space.

For category (iii), Tiramisu uses a novel graph traversal algorithm to check path existence. The algorithm invokes canonical depth-first search on multiple Tiramisu subgraphs, to account for tags (e.g., BGP communities) whose use on a path controls the path's existence. For such policies, Tiramisu matches ARC's performance for simple control planes; but Tiramisu is much more general.

Finally, for many of the policies in categories (i) and (ii), Tiramisu leverages the underlying model's graph structure to develop a graph algorithm-based accelerator for further verification speed-up. We show how to use a variant of a dynamic programming-based algorithm for the k shortest paths problem [31] to curtail needlessly exploring paths that manifest under many not-so-relevant failure scenarios.

We implemented Tiramisu in Java [2] and evaluated it with many real and synthetic configurations, spanning campus, ISP, and data center networks. We find that Tiramisu significantly outperforms the state-of-the-art, and is more general—some of the networks have layer-2/3 features that Minesweeper and Plankton don't model. Using category-specific algorithms on complex networks, Tiramisu verified category i, ii, and iii policies in *60*, *80* and *3ms*, respectively. Compared to Minesweeper, Tiramisu is *80X*, *50X*, and *600X* faster for category i, ii, and iii policies, respectively. On iBGP networks, Tiramisu outperforms Plankton by up to *300X* under failures. Tiramisu's *TYEN* acceleration improves performance by *1.3-3.8X*. Tiramisu scales well, providing verification results in ∼100ms per traffic class for networks with ∼160 routers.

2 Motivation

In this section, we provide an overview of state-of-the-art control plane verifiers. We then identify their key drawbacks that motivate Tiramisu's design.

2.1 Existing control plane verifiers

State-of-the-art control plane verifiers are based on three different techniques: graph algorithms [14], symbolic model checking [6,12,29], and explicit-state model checking [13,23]. We review the most advanced verifier in each category.

Graph algorithms: ARC [14] models a network's control plane using a set of directed graphs. Each graph encodes the network's forwarding behavior for a specific traffic class–i.e., packets with specific source and destination subnets. In the graphs, vertices represent routing processes (i.e. instances of routing protocols running on specific devices); directed edges represent possible network hops enabled by the exchange of advertisements between processes; edge weights encode OSPF costs or AS path lengths. ARC verifies a policy by checking a simple graph property: e.g, *src* and *dst* are always blocked if they are in separate components. By leveraging graph algorithms, ARC offers orders-of-magnitude better performance [14] than simulation-based verifiers [13]. However, ARC's simple graph model does not cover: widely used layer-3 protocols (e.g., iBGP), any layer-2 primitives (e.g., VLANs), and many protocol attributes (e.g., BGP community).

Symbolic model checking: Minesweeper [6] verifies a network's policy compliance by formulating and solving a satisfiability modulo theory (SMT) problem. The SMT constraints encode the behavior of the network's control and data planes (M) as well as the negation of a target policy ($\neg P$). If the SMT constraints ($M \wedge \neg P$) are unsatisfiable, then the policy is satisfied. To provide extensive network design coverage, Minesweeper uses many variables, which results in a large search space. Minesweeper verifies all policies on this large general SMT model. To verify for k failures, Minesweeper's SMT solver may, in the worst case, enumerate all possible combinations of k-link failures.

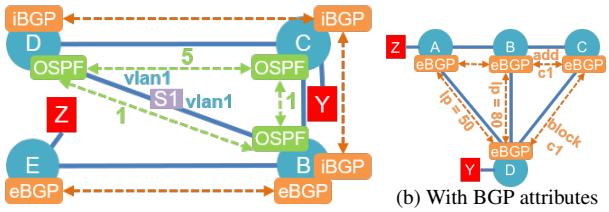
Explicit-state model checking: Plankton [23] models different control plane protocols (e.g., OSPF, BGP) using the Path Vector Protocol (PVP) [15] in a modeling language (Promela). It tracks dependencies among protocols defined in a network's control plane based on packet equivalence classes (PECs). It then uses an explicit state model checker, SPIN, to search on the overall state space of this model and find a state that violates a policy. To speedup verification, Plankton runs multiple independent SPIN instances in parallel. Plankton uses other optimizations including device equivalence to reduce the failure scenarios to explore. Despite these optimizations, Plankton still enumerates the state space of many failure scenarios, and checks them sequentially (§8).

2.2 Challenges

We now identify three high-level challenges that affect existing verifiers' performance and/or correctness.

Cross-layer dependencies. Consider the network in Figure 1a. Router B is an eBGP peer of router E, and router C is an iBGP peer of routers B and D. B and D are connected to switch $S1$ on VLAN 1. All routers except E belong to the same OSPF domain; the cost of link C–D is 5 and others is 1.

In this network, C learns a route to E through its iBGP neighbor B; the route's next hop is the IP address of B's loopback interface. In order for C to route traffic through B,



(a) With protocol dependencies

(b) With BGP attributes

Figure 1: Example networks

C must compute a route to B using OSPF. The computed path depends on the network's failure state. In the absence of failures, *OSPF* prefers path $C \rightarrow B$ (cost 1). When the link B–C fails, *OSPF* prefers a different path: $C \rightarrow D \rightarrow B$ (cost 6). Unfortunately, traffic for E is dropped at D, because D never learns a route to E; E is not in the same OSPF domain, and routes learned (by C) via iBGP are not forwarded. If B and D were iBGP peers, or B redistributed its *eBGP*-learned routes to *OSPF*, then this blackhole would be avoided.

ARC's simplistic graph abstraction cannot model iBGP and thus cannot model iBGP-OSPF dependencies. Hence, ARC cannot be used to verify policies in this network. Minesweeper can model iBGP, but its encoding is inefficient. To model iBGP, Minesweeper creates n additional copies of the network where n represents number of routers running iBGP. Each copy models forwarding towards the next-hop address associated with each iBGP router. This increases Minesweeper's SMT model size by nX, which significantly degrades its performance. In Plankton, the iBGP-OSPF dependency is encoded as a dependency between PECs, which prevents Plankton from fully parallelizing its SPIN instances. Hence, Plankton loses the performance benefits of parallelism.

Cross-layer dependencies also impact other network scenarios. Assume the $B-S1$ link in Figure 1a was assigned to VLAN 2. Now B and D are connected to the same switch $S1$ on different VLANs; internally, $S1$ runs two *virtual switches*, one for each VLAN. Hence, traffic between B and D cannot flow through switch $S1$. By default, ARC, Minesweeper, and Plankton, assume layer-2 connectivity. Thus, according to these verifiers, B and D are reachable and traffic can flow between them, which is incorrect.

The overall theme is that protocols "depend" on each other—e.g., iBGP depends on OSPF, BGP and OSPF depend on VLANs, etc.—and these dependencies must be *fully*, *correctly* and *efficiently* modeled.

Protocol attributes. Consider the network in Figure 1b. All routers (A–D) run eBGP. B adds community "c1" to the advertisements it sends to C, and D blocks all advertisements from C with community "c1". Additionally, D prefers routes learned from B over A by assigning local preference (lp) values 80 and 50, respectively.

The path that D uses to reach A depends on communities and local preference. There are three physical paths from D to A: (*i*) $D \rightarrow A$, (*ii*) $D \rightarrow B \rightarrow A$, and, (*iii*) $D \rightarrow C \rightarrow B \rightarrow A$. However, since router B adds community "c1" to routes advertised to C, and D blocks advertisements from C with this community, path *iii* is unusable. Furthermore, path *ii* is preferred over the shorter path *i* due to local preference.

Since ARC uses Dijkstra's algorithm to compute paths, it can only model additive path metrics like OSPF cost and AS-path length; it cannot model non-additive properties such as local preference, communities, etc. Hence, ARC incorrectly concludes that path *iii* is valid and (shortest) path *i* is preferred. Although Minesweeper and Plankton can model these attributes, they suffer from other drawbacks mentioned earlier.

Failures. Assume there are no communities in the network in Figure 1b, and routers A and D are configured to run OSPF in addition to eBGP. This network can tolerate a single link failure without losing connectivity.

According to ARC, traffic from D to A can flow through four paths: $D_{bgp} \rightarrow C_{bgp} \rightarrow B_{bgp} \rightarrow A_{bgp}$, $D_{bgp} \rightarrow B_{bgp} \rightarrow A_{bgp}$, $D_{bgp} \rightarrow A_{bgp}$, and $D_{ospf} \rightarrow A_{ospf}$. To evaluate the network's failure resilience, ARC calculates the *min-cut* of this graph, which is 3, and concludes that it can withstand two arbitrary, simultaneous link failures. This is incorrect because edges $D_{ospf} \rightarrow A_{ospf}$ and $D_{bgp} \rightarrow A_{bgp}$ are both unusable when the physical link D–A fails.

As mentioned in §2.1, Minesweeper and Plankton enumerate multiple failure scenarios and this makes them very slow to verify for failures. For example, in this 5-link network, to *verify reachability with 1 failure*, Minesweeper (and Plankton without optimization) may explore 5 failure scenarios before establishing reachability.

An overall issue is that irrespective of the policy, Minesweeper and Plankton need to compute the actual path taken in the network. ARC, on the other hand, represents policies as graph properties, e.g. mincut, connectivity, etc. It then uses fast polynomial time algorithms to compute these properties. However, ARC's simplistic graph abstraction cannot model all network features. Our goal is to create a tool that combines the network coverage of Minesweeper and Plankton, with the performance benefits of ARC.

3 Overview

Motivated by the inefficiencies and coverage limitations of existing network verifiers (§2), we introduce a new network verification tool called Tiramisu.Tiramisu is rooted in a rich graph-based network model that captures forwarding and failure dependencies across multiple routing and switching layers (e.g., BGP, OSPF, and VLANs). Since routing protocols are generally designed to operate on graphs and their constituent paths, graphs offer a natural way to express the route propagation, filtering, and selection behaviors encoded in device configurations and control logic. Moreover, graphs admit efficient analyses that allow Tiramisu to quickly reason about important network policies—including reachability, path lengths, and path preferences—in the context of multi-link failures. In this section, we highlight how Tiramisu's graph-based model and verification algorithms address the challenges discussed in §2. Detailed descriptions, algorithms, and proofs of cor-

Category	Policy	Meaning	Comments
i: compute path with *TPVP*	PREF	Path Preference	Can use *TYEN*
	MC	Multipath Consistency	
ii: compute actual numeric graph property with ILP	KFAIL	Reachability < K failures	Can use *TYEN*
	BOUND	All paths have length < K	
	EB	All paths have equal length	
iii: identify connectivity wth *TDFS*	BLOCK	Always Blocked	
	WAYPT	Always Waypointing	
	CW	Always Chain of Waypoints	
	BH	No Blackhole	

Table 1: Policies Verified

rectness, are presented in later sections.

3.1 Graph-based network model

To accurately and efficiently model cross-layer dependencies, Tiramisu constructs two inter-related types of graphs: *routing adjacencies graphs* (RAGs) and *traffic propagation graphs* (TPGs). The former allows Tiramisu to determine which routing processes may learn routes to specific destinations. The latter models more detail, especially, all the prerequisites for such learning to occur—e.g., OSPF must compute a route to an iBGP neighbor in order for the neighbor to receive BGP updates. Our verification algorithms run on the TPGs.

Routing adjacencies graph. A RAG (e.g, Figure 2a) encodes routing *adjacencies*. Two routing processes are adjacent if they are configured as neighbors (e.g., BGP) or configured to operate on interfaces in the same layer-2 broadcast domain (e.g., OSPF). A RAG contains a vertex for each routing process, and a pair of directed edges for each routing adjacency. Tiramisu runs a domain-specific "tainting" algorithm on the RAG to determine which routing processes may learn routes for a given prefix p.

Traffic propagation graph. In addition to routing adjacencies, propagation of traffic requires: (*i*) a route to the destination, (*ii*) layer-2 and, in the case of BGP, layer-3 routes to adjacent processes, and (*iii*) physical connectivity. A TPG (e.g., Figure 2b) encodes these *dependencies*. Vertices are created for each VLAN on each device and each routing process. Directed edges model the aforementioned dependencies as follows: (*i*) a VLAN vertex is connected to an OSPF/BGP vertex associated with the same router if, according to the RAG, the process may learn a route to a given subnet; (*ii*) an OSPF vertex is connected to the vertices for the VLANs on which it operates, and a BGP vertex is connected to an OSPF and/or VLAN vertex associated with the same router; (*iii*) a VLAN vertex is connected to a vertex for the same VLAN on another device if the devices are physical connected. Additionally, multi-attribute edge labels are assigned to edges to encode filters and "costs" associated with a route. With this structure, Tiramisu is able to correctly model a much wider range of networks than state-of-the-art graph-based models [14]. All verification procedures operate on the TPG.

3.2 Verification algorithms

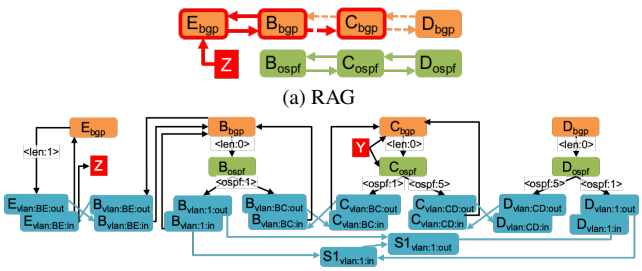
Tiramisu's verification process is rooted in the observation that network policies explored in practice and in academic research (Table 1) fall into three categories: (*i*) policies concerned with the actual path taken under specific failures—e.g., path preference (PREF); (*ii*) policies concerned with quantitative path metrics—e.g., how many paths are present (KFAIL) and bounds on path length (BOUND); and (*iii*) policies concerned with the existence of a path—e.g., blocking (BLOCK) and waypointing (WAYPT). Verifying policies in the first category requires high fidelity modeling of the control plane's output—namely, enumeration/computation of precise forwarding paths—whereas verifying policies in the last category requires low fidelity modeling of the control plane's output—namely, evidence of a single plausible path. For optimal efficiency, Tiramisu's core insight is to introduce category-specific verification algorithms that operate with the minimal level of fidelity required for accurate verification.

TPVP. To verify category *i* policies, Tiramisu efficiently solves the stable paths problem [15] using the *Tiramisu Path Vector Protocol* (*TPVP*). *TPVP* extends Griffin's "simple path vector protocol" (SPVP) [15]. In TPVP, each TPG node consumes the multi-dimensional attributes of outgoing edges, and uses simple arithmetic operations (based on a routing algebra [26]) to select among multiple available paths. In simple networks (that use a single routing protocol) *TPVP* devolves to a distance vector protocol, which computes paths under failures in polynomial time. For such networks, TPVP is comparable to ESMC tools [23], but is faster than general SMT-based tools [6]. For general networks with dependent control plane protocols, TPVP is faster than SMT-based tools, because it uses a domain-specific approach to computing paths, compared to an SMT-based general search strategy. TPVP also beats ESMC tools by emulating the control plane in one shot as opposed to emulating the constituent protocols and exploring their state space one at a time to account for inter-protocol dependencies.

ILP. For category *ii* policies, Tiramisu leverages general integer linear program (ILP) encodings that compute network flows through the TPG, rather than precise paths. The ILPs only consider protocol attributes that impact whether paths can materialize (e.g., BGP communities), and they avoid path computation. Thus, Tiramisu is much faster than state-of-the-art approaches that always consider all attributes (e.g., BGP local preferences) and enumerate actual paths [6, 23] (§8).

TDFS. Finally, *Tiramisu depth-first search* (*TDFS*) is a novel polynomial-time graph traversal algorithm to check for the existence of paths and verify category *iii* policies. *TDFS* makes a constant number of calls to the canonical DFS algorithm. Each call is to a subgraph of the TPG that models the interaction between tag-based (e.g., BGP-community-based) route filters along a path that control if a path can materialize.

Tiramisu further improves over state-of-the-art verifiers for some category *i* and *ii* policies by avoiding unnecessary path computation. Specifically, we note that some category *i* and *ii* properties require knowing *when* certain paths are taken (i.e., after how many link failures, or after how many more-preferred paths have failed). For such properties, ex-



(a) RAG

(b) TPG

Figure 2: Graphs for network in Figure 1a

haustively exploring all failures by running *TPVP* for each scenario, while sufficient, is overkill. To avoid enumerating not-so-useful failure scenarios, Tiramisu leverages the graph structure of the network model to run a variant of Yen's dynamic programming based algorithm for k-shortest paths [31], to directly compute a preference order of paths that manifest under arbitrary failures. Our variant, *TYEN*, invokes *TPVP* in a limited fashion from within Yen's execution, minimizing path exploration. For PREF over k paths, we simply use *TYEN* to compute the top-k paths over the TPG. Likewise, we use *TYEN* to accelerate KFAIL, a category *ii* policy. Note that *TYEN* can only be applied to networks whose path metrics are monotonic [16].

4 Tiramisu Graph Abstraction

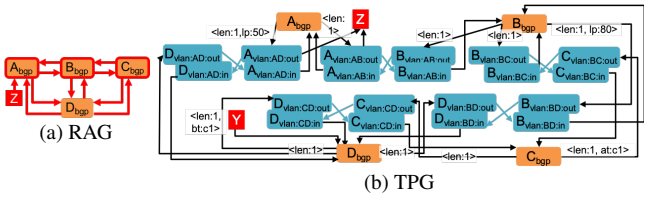
In this section, we describe in detail the two types of graphs used in Tiramisu: *routing adjacencies graphs* (RAGs) and *traffic propagation graphs* (TPGs). TPGs are partially based on RAGs, and both are based on a network's configurations and physical topology.

4.1 Routing adjacencies graphs

RAGs encode routing adjacencies to allow Tiramisu to determine which routing processes may learn routes to specific IP subnets. Tiramisu constructs a RAG for each of a network's subnets. For example, Figures 2a, and 3a show the RAGs for subnet Z for the networks in Figure 1.

Vertices. A RAG has a vertex for each routing process. For example, B_{bgp} and B_{ospf} in Figure 2a represent the BGP and OSPF processes on router B in Figure 1a. A RAG also has a vertex for each device with static routes for the RAG's subnet.

Edges. A RAG contains an edge for each routing adjacency. Two BGP processes are adjacent if they are explicitly configured as neighbors. Two OSPF process are adjacent if they are configured to operate on router interfaces in the same Layer 2 (L2) broadcast domain (which can be determined from the topology and VLAN configurations). These adjacencies are represented using pairs of directed edges—e.g., $E_{bgp} \leftrightarrows B_{bgp}$ and $B_{ospf} \leftrightarrows C_{ospf}$—since routes can flow between these processes in either direction. However, if two processes are iBGP neighbors then a special pair of directional edges are used—e.g., $B_{bgp} \dashrightarrow C_{bgp}$—because iBGP processes do not forward routes learned from other iBGP processes. A routing adjacency is also formed when one process (the redistributor)



(a) RAG

(b) TPG

Figure 3: Graphs for network in Figure 1b

distributes routes from another process on the same device (the redistributee). This is encoded with a unidirectional edge from redistributee to redistributor. Vertices representing static routes may be redistributees, but will not have any other edges.

Taints. To determine which routing processes may learn routes to specific destinations, Tiramisu runs a "tainting" algorithm on the RAG. All nodes that originate a route for the subnet associated with the RAG (including vertices corresponding to static routes) are tainted. Then taints propagate freely across edges to other vertices, with one exception: when taints traverse an iBGP edge they cannot immediately traverse another iBGP edge. For example, in Figure 2a, E_{bgp} is tainted, because it originates a route for Z. Then taints propagate from E_{bgp} to B_{bgp} to C_{bgp}, but not to D_{bgp}. No OSPF vertices are tainted, because no OSPF processes originate a route for Z and no processes are configured to redistribute routes.

The tainting algorithm assumes all configured adjacencies are active and no routes are filtered. However, for an adjacency to be active in the real network, certain conditions must be satisfied: e.g., B_{ospf} must compute a route to C's loopback interface and vice versa in order for B_{bgp} to exchange routes with C_{bgp}. These dependencies are encoded in TPGs.

4.2 Traffic propagation graph

A process P on router R can advertise a route for subnet S to an adjacent routing process P' on router R' if all of the following dependencies are satisfied: (*i*) P learns a route for S from an adjacent process, or P is configured to originate a route for S; (*ii*) neither P or P' filters the advertisement; (*iii*) another process/switch on R learns a route to R', or R is connected to the same subnet/layer-2 domain as R'; and (*iv*) R is physically connected to R' through a sequence of one or more links. TPGs encode these dependencies. Tiramisu constructs a TPG for every pair of a network's IP subnets. Figures 2b and 3b show the TPGs that model how traffic from subnet Y is propagated to subnet Z for the networks in Figure 1.

Vertices. A TPG's vertices represent the routing information bases (RIBs) present on each router and the (logical) traffic ingress/egress points on each router/switch. Each routing process maintains its own RIB, so the TPG contains a vertex for each process: e.g., B_{bgp} and B_{ospf} in Figure 2b correspond to the BGP and OSPF processes on router B in Figure 1a.

Traffic propagates between routers and switches over VLANs. Consequently, the TPG contains a pair of ingress/egress vertices for each of a device's VLANs: e.g., $S1_{v1:in}$ and $S1_{v1:out}$ in Figure 2b correspond to the VLAN on switch $S1$ in Figure 1a. Tiramisu creates implicit VLANs for pairs of directly connected interfaces: e.g., $B_{vlan:BE:in}$,

$B_{vlan:BE:out}$, $E_{vlan:BE:in}$, and $E_{vlan:BE:out}$ in Figure 2b correspond to the directly connected interfaces on routers B and E in Figure 1a.

The TPG also includes vertices for the source and destination (target) of the traffic being propagated: e.g., Y and Z, respectively, in Figure 2b.

Edges. A TPG's edges reflect the virtual and physical "hops" the traffic may take. Edges model dependencies as follows:

- **Layers 1 & 2:** For each VLAN V on device D, the egress vertex for V on D is connected to the ingress vertex for V on device D' if an interface on D participating in V has a direct physical link to an interface on D' participating in V: e.g., $B_{vlan1:out} \rightarrow S1_{vlan1:in}$ in Figure 2b corresponds to the physical link between B and $S1$ in Figure 1a. Also, the ingress vertex for V on D is connected to the egress vertex for V on D to model L2 flooding: e.g., $S1_{vlan1:in} \rightarrow S1_{vlan1:out}$.
- **OSPF's dependence on L2:** The vertex for OSPF process P on router R is connected to the egress vertex for VLAN V on R if P is configured to operate on V: e.g., $D_{ospf} \rightarrow D_{vlan:CD:out}$ and $D_{ospf} \rightarrow D_{vlan:1:out}$ in Figure 2b model OSPF operating on router D's VLANs in Figure 1a.
- **BGP's dependence on connected and OSPF routes:** For each peer N of the BGP process P on router R, an edge is created from the vertex for P to the egress vertex for VLAN V on R if N's IP address falls within the subnet assigned to V: e.g., $E_{bgp:B} \rightarrow E_{vlan:BE:out}$ in Figure 2b models the BGP process on router E communicating with the adjacent process on directly connected router B in Figure 1a. If no such V exists, then the vertex for P is connected to the vertex for OSPF process P' on R: e.g., $B_{bgp} \rightarrow B_{ospf}$ models the BGP process on B communicating with the adjacent process on router C (which operates on a loopback interface) via an OSPF-computed path.
- **Routes to the destination:** Every VLAN ingress vertex on router R is connected to the vertex for process P on R if the vertex for P in the RAG is tainted: e.g., $B_{vlan:BC:in} \rightarrow B_{bgp}$ in Figure 2b is created due to the taint on B_{bgp} in Figure 2a, which models that fact that the BGP process on B may learn a route to subnet Z from the adjacent process on E.
  If the destination subnet T is connected to R and at least one routing process on R originates T, then every VLAN ingress vertex on R is connected to the vertex for T: e.g., $E_{vlan:BE:in} \rightarrow Z$ in Figure 2b.
  If the source subnet S is connected to R and the vertex for process P on R is tainted in the RAG, then the vertex for S is connected to the vertex for P: e.g., $Y \rightarrow C_{bgp}$ in Figure 2b.

Filters. As mentioned in §4.1, a RAG may overestimate which processes learn a route to a subnet due to route and packet filters not being encoded in the RAG. A TPG models filters using two approaches: edge pruning and edge attributes.

Tiramisu uses edge pruning to model prefix- or neighbor-based filters. A BGP process P may filter routes imported from a neighbor P' (or P' may filter routes exported to P) based on the advertised prefix or neighbor from (to) whom the route is forwarded. Tiramisu models such a filter by removing from the vertex associated with P, the outgoing edge that corresponds to P'. For example, if router B in Figure 1a had an import filter, or router E had an export filter, that block routes for Z, then edge $B_{bgp} \rightarrow B_{vlan:BE:out}$ would be removed from Figure 2a. Note that import and export filters are both modeled by removing an outgoing edge from the vertex associated with the importing process. OSPF is limited to filtering incoming routes based on the advertised prefix. Tiramisu models such route filters by removing all outgoing edges from the vertex associated with the OSPF process where the filter is deployed. Lastly, packets sent (received) on VLAN V can be filtered based on source/destination prefix. Tiramisu models such packet filters by removing the outgoing (incoming) edge that represents the physical link connecting V to its neighbor.

Tiramisu uses edge attributes to model tag- (e.g., BGP community- or ASN-) based filters. If a BGP process P filters routes imported from a neighor P' (or P' filters routes exported to P) based on tags, then Tiramisu adds a "blocked tags" attribute to the outgoing edge from the vertex associated with P that corresponds to P'. For example, the edge $D_{bgp} \rightarrow D_{vlan:CD:out}$ in Figure 3b is annotated with $bt = \{c1\}$ to encode the import filter router D applies to routes from C in Figure 1b. Edges can also include "added tags" and "removed tags" attributes: e.g., $C_{bgp} \rightarrow C_{vlan:BC:out}$ is annotated with $at = \{c1\}$ to encode the export filter router B applies to routes advertised to C. Notice that tag actions defined in import and export filters are both added to an outgoing edge from the vertex associated with the importing process.

Costs/preferences. Each routing protocol uses a different set of *metrics* to express link and path costs/preferences. For example, OSPF uses link costs, and BGP uses AS-path length, local preference (lp), etc. Similarly, administrative distance (*AD*) allows routers to choose routes from different protocols. Hence a single edge weight cannot model the route selection decisions of all protocols. Tiramisu annotates the outgoing edges of OSPF, BGP, and VLAN ingress vertices with a *vector of metrics*. Depending on the edge, certain metrics will be null: e.g., OSPF cost is null for edges from BGP vertices.

5 Category (i) Policies

We now describe how Tiramisu verifies policies that require knowing the actual path taken in the network under a given failure (§3.2). One such policy is path preference (PREF). For example, $P_{pref} = p1 \gg p2 \gg p3$, states when path $p1$ fails, $p2$ (if available) should be taken; and when $p1$ and $p2$ both fail, $p3$ (if available) should be taken. A path can become unavailable if a link or device along the path fails. We model such failures by removing edges (between egress and ingress VLAN vertices) or vertices from the TPG. To verify PREF, we need to know what alternate paths materialize, and whether a materialized path is indeed the path meant to be

taken according to preference order. Similar requirements arise for verifying multipath consistency (MC).

In this section, we introduce an algorithm, *TPVP*, to compute the actual path taken in the network. *TPVP* can be used to exhaustively explore failures to verify category (i) policies, but it is slow. We show how to accelerate verification using a dynamic-programming based graph algorithm.

5.1 Tiramisu Path Vector Protocol

Griffin et al. observed that BGP attempts to solve the *stable paths problem* and proposed the Simple Path Vector Protocol (SPVP) for solving this problem in a distributed manner [15]. Subsequently, Sobrinho proposed routing algebras for modeling the route computation and selection algorithms of *any* routing protocol in the context of a path vector protocol [25]. Griffin and Sobrinho then extended these algebras to model routing across administrative boundaries—e.g., routing within (using OSPF) and across (using BGP) ASes [16]. However, they did not consider the dependencies between protocols within the same administrative region—e.g., iBGP's dependence on an OSPF. Recently, Plankton addressed these dependencies by solving multiple stable paths problems, and proposed the Restricted Path Vector Protocol (RPVP) for solving these problems in a centralized manner [23].

Below, we explain how the *Tiramisu Path Vector Protocol* (*TPVP*) leverages routing algebras and extends SPVP to compute paths using our graph abstraction. Since SPVP was designed for Layer 3 protocols, *TPVP* works atop a simplified TPG called the Layer 3 TPG (L3TPG). Tiramisu uses path contraction to replace L2 paths with an L3 edge connecting L3 nodes, to model the fact that advertisements can flow between routing processes as long as there is at least one L2 path that connects them. Recall that the incoming edges of VLAN egress vertices and the outgoing edges of VLAN ingress vertices include (non-overlapping) edge labels (§4.2); these are combined and applied to the L3 edge(s) that replace the L2 path(s) containing these L2 edges.

Routing algebras. Routing algebras [16, 25] model routing protocols' path cost computations and path selection algorithms. An algebra is a tuple $(\Sigma, \preceq, L, \oplus, O)$ where:

- Σ is a set of *signatures* representing the multiple metrics (e.g., AS-path length, local pref, ...) associated with a path.
- $\preceq$ is the *preference* relation over signatures. It models route selection, ranking paths by comparing multiple metrics of multiple path signatures in a predefined order (e.g., first compare local pref, then AS-path length, ...)
- L is set of *labels* representing multi-attribute edge-weight.
- $\oplus$ is a function $L \times \Sigma \to \Sigma$, capturing how labels and signatures combine to form a new signature; i.e., $\oplus$ models path cost computations. $\oplus$ has multiple operators, each computing on certain metrics.[2]
- O is the signature attached to paths at origination.

[2]For example, *ADD* operator adds OSPF link costs and AS-path lengths. *LP* operator sets local pref, *TAGS* operator prohibits paths with a tag, etc.

In Tiramisu, path signatures contain metrics from all possible protocols (e.g., OSPF cost, AS-path length, AD, ...), but $\preceq$ and $\oplus$ are defined on a per-protocol basis and only operate on the metrics associated with that protocol. For example, $\oplus_{BGP}$ sets local pref and adds AS-path lengths from a label ($\lambda \in L$), but copies the OSPF cost and AD directly from the input signature ($\sigma \in \Sigma$) to the output signature ($\sigma' \in \Sigma$). Similarly, $\preceq_{BGP}$ compares local pref, AS-path lengths, and OSPF link costs[3] but does not compare AD.

TPVP. *TPVP* (Algorithm 1) is derived from SPVP [15]. For each vertex in the L3TPG, *TPVP* computes and selects a most-preferred path to *dst* based on the (signatures of) paths selected by adjacent vertices. However, *TPVP* extends *SPVP* in two fundamental ways: (*i*) it uses a shared memory model instead of message passing, akin to *RPVP* [23]; and (*ii*) it models multiple protocols in tandem by computing path signatures and selecting paths using routing algebra operations corresponding to different protocols: e.g., $\oplus_{BGP}$ and $\preceq_{BGP}$ are applied at vertices corresponding to BGP processes.

For each peer v of each vertex u (v is a peer of u if $u \to v \in$ L3TPG) *TPVP* uses variables $p_u(v)$ and $\sigma_u(v)$ to track the most preferred path to reach *dst* through v and the path's signature, respectively. Likewise, variables p_u and σ_u represent the most preferred path and its signature to reach *dst* from u.

In the initial state, *TPVP* sets the *path* and *sign* values of all nodes except *dst* to null (line 2). p_{dst} is set to ε and σ_{dst} is set to O, since it "originates" the advertisement (line 3). Similar to *SPVP*, there are three steps in each iteration. First, for each node u, *TPVP* computes all its $p_u(*)$ and $\sigma_u(*)$ values based on the path signatures of its neighbors and outgoing edge labels $\lambda_{u\to *}$ (lines 7–10). It calculates the best path based on the preference relation (line 11). If the current p_u changes from previous iteration, then the network has not converged and the process repeats (lines 12–13).

Theorem 1. *If the network control plane converges,* TPVP *always finds the exact path taken in the network under any given failure scenario.*

We prove Theorem 1 is in Appendix C.1. The proof shows that *TPVP* and the TPG correctly model the permitted paths and ranking function to solve the stable paths problem.

Plankton [23] leverages basic SPVP to model the network. But because basic SPVP cannot directly model iBGP, to verify networks that use iBGP, Plankton runs multiple SPVP instances. As mentioned in §2.1, BGP routing depends on the outcome of OSPF routing. Hence, Plankton runs SPVP multiple times: first for multiple OSPF instances, and then for dependent BGP instances. In contrast, because Tiramisu's *TPVP* is built using routing algebra atop a network model with rich edge attributes, we can bring different dependent routing protocols into one common fold of route computation. Thus, we can analyze iBGP networks, and, generally, dependent protocols, "in one shot" by running a single *TPVP* instance.

[3]OSPF cost is used as a tie-breaker in BGP path selection [10].

Algorithm 1 Tiramisu Path Vector Protocol

```
1:  procedure TPVP(L3TPG)
2:    ∀i ∈ {V − dst} : p_i = ∅, σ_i = ϕ
3:    p_dst = [dst], σ_dst = O                    ▷ dst originates the route
4:    converged = false
5:    while ¬converged do
6:      converged = true
7:      for each u ∈ L3TPG do
8:        for each v ∈ peers(u) do
9:          p_u(v) = edge_{u→v} ∘ p_v              ▷ add edge to path
10:         σ_u(v) = λ_{u→v} ⊕_{type(u)} σ_v
11:         compute p_u and σ_u using ⪯ over σ_u(∗)
12:         if p_u has changed then
13:           converged = false
```

In Appendix D, we show that *TPVP* can verify other policies, like "Multipath Consistency (MC)", that requires materialization of certain paths.

5.2 Tiramisu Yen's algorithm

To verify PREF, Tiramisu runs *TPVP* multiple times to compute paths for different failure scenarios. For example, while verifying P_{pref}, Tiramisu runs *TPVP* for all possible failures (edge removals) that render paths $p1$ and $p2$ unavailable. Then, it checks if *TPVP* always computes $p3$ (if available). While correct, this is tedious and slow overall.

We can accelerate the verification of this policy by leveraging the graph structure of the L3TPG and developing a graph algorithm that avoids unnecessary path explorations/computations. Specifically, we observed that there are similarities between analyzing PREF and finding the k shortest paths in a graph [31]. This is because, in the k shortest paths problem, the k^{th} shortest path is taken only when $k-1$ paths have failed. To avoid enumerating all possible failures of all $k-1$ shorter paths, Yen [31] introduced an efficient algorithm for this problem that uses dynamic programming to avoid failure enumeration. Yen uses the intuition that the k^{th} shortest path will be a small perturbation of the previous $k-1$ shortest paths. Instead of searching over the set of all paths, which is exponential, Yen constructs a polynomial candidate set from the previous $k-1$ paths, in which the k^{th} path will be present.

To accelerate PREF, our *TYEN* algorithm makes two simple modifications to Yen. Yen uses Dijkstra to compute the shortest path. We replace Dijkstra with *TPVP*. Next, we add a condition to check that during the i^{th} iteration, the i^{th} computed path follows the preference order specified in PREF. The detailed description of Yen and *TYEN* are in Appendix A. Note that Yenand hence TYEN, assumes the path-cost composition function is strictly monotonic. Hence, TYEN acceleration can be leveraged only on monotonic networks.

6 Category (ii) Policies

We now describe how Tiramisu verifies policies pertaining to quantitative path metrics, e.g., KFAIL and BOUND. For such policies, Tiramisu uses property-specific ILPs. These ILPs run fast because they abstract the underlying TPG and only model protocol attributes that impact whether paths can materialize (e.g., communities).

6.1 Tiramisu Min-cut

KFAIL states that *src* can reach *dst* as long as there are $< K$ link failures. ARC [14] verifies this policy by computing the min-cut of a graph; if min-cut is $\geq K$, then KFAIL is satisfied.

However, standard min-cut algorithms do not consider how route tags (e.g., BGP communities) impact the existence of paths. For example, in Figure 3b, any path that includes $D_{bgp:c} \rightarrow D_{vlan:CD:out}$ followed by $C_{bgp:b} \rightarrow C_{vlan:BC:out}$ is prohibited due to the addition and filtering of tags on routers B and D, respectively. In this manner, tags prohibit paths with certain combinations of edges in the TPG, making the TPG a "correlated network". It is well known that finding the min-cut of a correlated network is NP-Hard [30]. Note that traffic flows in the direction opposite to route advertisements. Hence, the prohibited paths have tag-blocking edges followed by tag-adding edges.

We propose an ILP which accounts for route tags, but ignores irrelevant edge attributes (e.g., edge costs), to compute the min-cut of a TPG For brevity, we explain the constraints at a high-level, leaving precise definitions to Appendix B.1. Equation numbers below refer to equations in Appendix B.1.

The objective of the ILP is to minimize the number of physical link failures (F_i) to disconnect *src* from *dst*.

$$\textbf{Objective:} \qquad \texttt{minimize} \sum_{i \in pEdges} F_i \qquad (1)$$

Traffic constraints. We first define constraints on reachability. The base constraint states that *src* originates the traffic (Eqn 2). To disconnect the graph, the next constraint states that the traffic must not reach *dst* (Eqn 3). For other nodes, the constraint is that traffic can reach a node if it gets *propagated* on any of its incoming edges (Eqn 4).

Now we define constraints on traffic propagation. Traffic can propagate through an edge e if: it reaches the start node of that edge; the traffic does not carry a tag that is blocked on that edge; and, if the edge represents an inter-device edge, the underlying physical link has not failed. This is represented as shown in Eqn 5.

Tags. We now add constraints to model route tags. The base constraints state that each edge that blocks on a tag forwards that tag, and each edge that removes that tag does not forward it further (Eqn 6 and Eqn 7). For other edges, we add the constraint that edge e forwards a tag t iff the start node of edge e receives traffic with that tag (Eqn 8). Finally, we add the constraint that an edge e carries a blocked tag iff that blocked tag can be added by edge e (Eqn 9).

We prove the correctness of this ILP in Appendix C.2 based on the correctness of Tiramisu's modeling of permitted paths.

Using TYEN for KFAIL with ACLs. This ILP is not accurate when packet filters (ACLs) are in use. ACLs do not influence advertisements. Hence, routers can advertise routes for traffic

that get blocked by ACLs. Recall that during graph creation, Tiramisu removes edges that are blocked by ACLs §4.2. This leads to an incorrect min-cut computation as shown below:

Assume *src* and *dst* are connected by three paths $P1$, $P2$ and $P3$ in decreasing order of path-preference. Also assume these paths are edge disjoint and path $P2$ has a data plane ACL on it. If a link failure removes $P1$, then the control plane would select path $P2$. However, all packets from *src* will be dropped at the ACL on $P2$. In this case, a single link failure (that took down $P1$) is sufficient to disconnect *src* and *dst*. Hence the true min-cut is 1. On the other hand, Tiramisu would remove the offending ACL edge from the graph. The graph will now have paths $P1$ and $P3$, and the ILP would conclude that the min-cut is 2, which is incorrect.

We address this issue as follows. Nodes can become unreachable when failures either disconnect the graph or lead to a path with an ACL. We compute two quantities: (1) N: Minimum failures that disconnects the graph, and, (2) L: Minimum failures that cause the control plane to pick a path blocked by an ACL. The true min-cut value is $min(N,L)$.

First we use our min-cut ILP to compute N. To compute L, in theory, we could simply run TPVP to exhaustively explore k-link failures for $k = 1, 2, ..,$ and determine the smallest failure set that causes the path between *src* and *dst* to traverse an ACL. However, this is expensive.

We can accelerate this process by leveraging *TYEN*, similar to our approach for PREF. We first construct a TPG without removing edges for ACLs. Then, we run *TYEN* until we find the first path with a dropping ACL on it. Say this was the M^{th} preferred path. Then, we remove all edges from the graph that do not exist in any of the previous $M-1$ paths. Next, we use our min-cut ILP to compute the minimal failures L to disconnect this graph. This represents the minimal failures to disconnect previous $M-1$ paths and pick the ACL-blocked path. If $min(L,N) \geq K$ then KFAIL is satisfied.

Overall, to compute KFAIL, the above requires Tiramisu to run (a) *TYEN* to compute M paths; and (b) two min-cut ILPs to compute N and L respectively. Note that to verify KFAIL, Minesweeper will explore all combinations of k link failures.

6.2 Tiramisu Longest path

Always bounded length policy (BOUND) states that for a given K, BOUND is true if under every failure scenario, traffic from *src* to *dst* never traverses a path longer than K hops. Enumerating all possible paths and finding their length is infeasible.

However, this policy can be verified efficiently by viewing it as a variation of computing a quantitative path property, namely the *longest path problem*: for a given K, BOUND is true if the longest path between *src* and *dst* is $\leq K$.

Finding the longest path between two nodes in a graph is also NP hard [19]. To verify BOUND, we thus propose another ILP whose objective is to maximize the number of inter device edges ($dEdges$) traversed by traffic (A_i).

$$\textbf{Objective:} \quad \texttt{maximize} \sum_{i \in dEdges} A_i \qquad (2)$$

We present detailed constraints and a proof of correctness in Appendices B.2 and C.2, respectively.

Constraints. We first add constraints to ensure that traffic flows on one path, *src* sends traffic, and *dst* receives it (Eqn 11 and Eqn 12). For other nodes, we add the flow conservation property, i.e., the sum of incoming flows is equal to the sum of outgoing flows (Eqn 13). Finally, we add constraints on traffic propagation: traffic will be blocked on edge e if it satisfies the tag constraints (Eqn 14).

In Appendix D, we show that a similar ILP can be used to verify other policies of interest—e.g., all paths between *src* and *dst* have equal length (EB).

7 Category (iii) Policies

Finally, we describe how Tiramisu verifies policies that only require us to check for just the existence of a path, e.g., "always blocked" (BLOCK). For these policies, we use a new simple graph traversal algorithm. Tiramisu's performance is fastest when checking for such policies (§8).

Standard graph traversal algorithms, like *DFS*, also do not account for tags. DFS will identify the prohibited path from Figure 3b ($D_{bgp:c} \rightarrow D_{vlan:CD:out}$ followed by $C_{bgp:b} \rightarrow C_{vlan:BC:out}$) as valid, which is incorrect. To support tags, we propose *TDFS*, Tiramisu Depth First Search (Algorithm 2). *TDFS* makes multiple calls to *DFS* to account for tags.

TDFS. As mentioned in §4, edges can add, remove or block on tags. In presence of such edges, the order in which these edges are traversed in a path determines if *dst* is reachable.

TDFS first checks if *dst* is unreachable from *src* according to *DFS* (line 3, 4). If they are reachable, then *TDFS* checks if all paths that connect *src* to *dst* (i) have an edge (say X) that blocks route advertisements and hence traffic (for *dst*) with a tag (line 5 to 6), (ii) followed by an edge (say Y) that adds the tag to the advertisements for *dst* (line 7 to 8), and (iii) has no edge between X and Y that removes the tag (line 9 to 10). If all these conditions are satisfied, then *src* and *dst* are unreachable. If any of these conditions are violated, the nodes are reachable.

We prove the correctness of *TDFS* in Appendix C.3.

The above algorithm naturally applies to verifying BLOCK. It can similarly be used to verify WAYPT ("always waypointing"): after removing the waypoint, if *src* can reach *dst*, then there is a path that can reach *dst* without traversing the waypoint. *TDFS* can also verify "always chain of waypoints (WAYPT)" and "no blackholes (BH)" (Appendix D).

8 Evaluation

We implemented Tiramisu in Java (≈ 7K lines of code) [2]. We use Batfish [13] to parse router configurations and Gurobi [3] to solve our ILPs. We evaluate Tiramisu on a variety of issues:

- How quickly can Tiramisu verify different policies?
- How does Tiramisu perform compared to state-of-the-art?

Algorithm 2 Always blocked with tags

```
1:  procedure TDFS(G, src, dst)
2:    tA, tR, tB ← edges that add, remove, or block on tag (respectively)
3:    if dst is unreachable by DFS (G, src) then
4:      return true (nodes are already unreachable)
5:    if dst is reachable by DFS (G-tB, src) then
6:      return false (∃ path src ⇝ dst where tagged-routes are not blocked)
7:    if ∃e_b ∈ tB s.t. dst is reachable by DFS (G-tA, e_b) then
8:      return false (tag-blocking edges can get ads for dst without tags)
9:    if ∀e_b ∈ tB, e_a ∈ tA: e_a is unreachable by DFS (G-tR, e_b) then
10:     return false (tags always removed before reaching blocking edges)
11:   return true
```

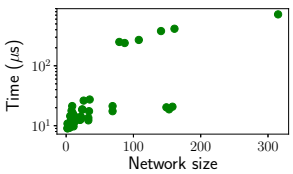


Figure 4: Graph generation time (all networks)

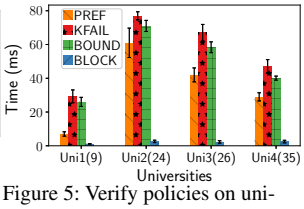


Figure 5: Verify policies on university configs

• How does Tiramisu's performance scale with network size?

Our experiments were performed on machines with dual 10-core 2.2 GHz Intel Xeon Processors and 192 GB RAM.

8.1 Network Characteristics

In our evaluation, we use configurations from (a) 4 real university networks, (b) 34 real datacenter networks operated by a large online service provider, (c) 7 networks from the topology zoo dataset [20], and (d) 5 networks from the Rocketfuel dataset [27]. The university networks have 9 to 35 devices and are the richest in terms of configuration constructs. They include eBGP, iBGP, OSPF, static routes, packet/route filters, BGP communities, local preferences, VRFs and VLANs. The datacenter networks have 2 to 24 devices and do not employ local preference or VLANs. The topology zoo networks have 33 to 158 devices, and the Rocketfuel networks have 79 to 315 devices. The configs for topology zoo and Rocketfuel were synthetically generated for random reachability policies [11, 23] and do not contain static routes, packet filters, VRFs or VLANs. Details on the TPGs for these networks are in Appendix E. The main insight is that the number of routing processes and adjacencies per device varies. Hence, the number of nodes and edges in the TPG do not monotonically increase with network size.

Policies. We consider five polices: (PREF) path preference, (KFAIL) always reachable with $< K$ failures, (BOUND) always bounded length, (WAYPT) always waypointing, and (BLOCK) always unreachable. Recall that: PREF is *category i* and is accelerated by *TYEN* calling *TPVP* from within; KFAIL and BOUND are *category ii* and use ILPs; BLOCK and WAYPT are *category iii* and use *TDFS*.

8.2 Verification Efficiency

We examine how efficiently Tiramisu can construct and verify these TPGs. First, we evaluate the time required to generate the TPGs. We use configurations from all the networks. Figure 4 shows the time taken to generate a *traffic class-specific TPG* for all networks. Tiramisu can generate these graphs, even for large networks, in ≈ 1 *ms*.

Next, we examine how efficiently Tiramisu verifies various policies. Since the university networks are the richest in terms of configuration constructs, we use them in this experiment. Because of VRF/VLAN, Minesweeper, Plankton and ARC cannot model these networks. Figure 5 shows the time taken to verify PREF, KFAIL, BOUND, and BLOCK. Since WAYPT uses *TDFS*, it is verified in a similar order-of-magnitude time as BLOCK. Hence for brevity, we do not show their results. In this and all the remaining experiments, the values shown are the median taken over *100 runs* for *100 different traffic classes*. Error bars represent the std. deviation.

We observe that BLOCK can be verified in less than **3 ms**. Since it uses a simple graph traversal algorithm (*TDFS*), it is the fastest to verify among all policies. In fact, for BLOCK, our numbers are comparable to ARC [14]. The time taken to verify PREF is higher than BLOCK, because *TPVP* and *TYEN* algorithms are more complex, as they run our path vector protocol to convergence to find paths (and in *TYEN*'s case, *TPVP* is invoked several times). Finally, KFAIL and BOUND, both use an ILP and are the slowest to verify. However, they can still be verified in $\approx$ **80 ms** per traffic class.

Although *Uni*2 and *Uni*3 have fewer devices than *Uni*4, their TPGs are larger (§E), so it takes longer to verify policies.

8.3 Comparison with Other tools

Next, to put our performance results in perspective, we compare Tiramisu with other state-of-art verification tools.

Minesweeper [6] In this experiment we use datacenter networks and consider policies PREF, BOUND, WAYPT, and BLOCK. Minesweeper takes the number of failures (K) as input and checks if the policy holds as long as there are $\leq K$ failures. To verify a property under all failure scenarios, we set the value of K to one less than the number of physical links in the network. Figure 6 shows the time taken by Tiramisu and Minesweeper to verify these policies, and Figure 7 shows the speedup provided by Tiramisu.

PREF is the only policy where speedup does not increases with network size. This is because larger networks have longer path lengths and more possible candidate paths, both of which affect the complexity of the *TYEN* algorithm. The number of times *TYEN* invokes *TPVP* increases significantly with network size. Hence the speedup for PREF is relatively less, especially at larger network sizes. For BOUND, the speedup is as high as 50X. For policies that use *TDFS* (BLOCK and WAYPT), Tiramisu's speedup is as high as 600-800X.

Next, we compare the performance of Tiramisu and Minesweeper for the same policies but without failures, e.g. "currently reachable" instead of "always reachable". Tiramisu verifies these policies by generating the actual path using *TPVP*. Figure 8 (a, b, c, and d) shows the speedup provided by Tiramisu for each of these policies. Even for *no failures*,

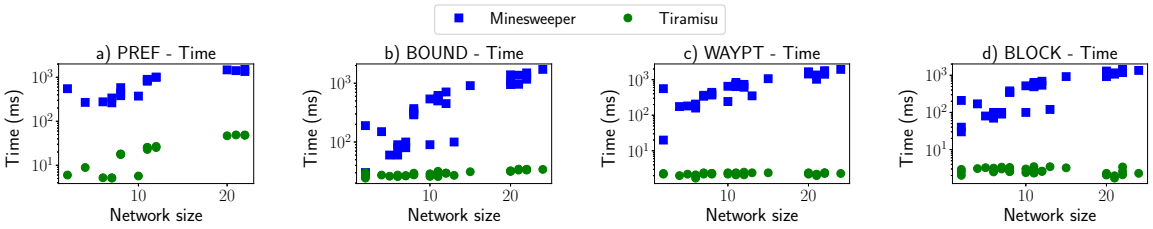


Figure 6: Performance under all failures: Tiramisu vs Minesweeper (datacenter networks)

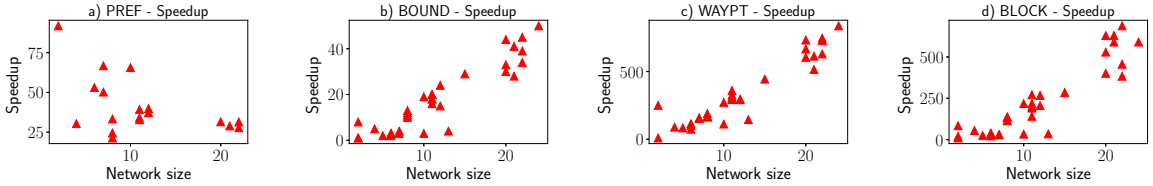


Figure 7: Speedup under all failures: Tiramisu vs Minesweeper (datacenter networks)

Tiramisu significantly outperforms Minesweeper across all policies. Minesweeper has to invoke the SMT solver to find a satisfying solution even in this simple case.

To shed further light on Tiramisu's benefits w.r.t. Minesweeper, we compare the number of variables used by Minesweeper's SMT encoding and Tiramisu's ILP encoding to verify KFAIL and BOUND. We observed that Tiramisu uses 10-100X fewer variables than Minesweeper. For Tiramisu, we also found that BOUND uses fewer variables than KFAIL.

Plankton [23] Next, we compare Tiramisu against Plankton using the Rocketfuel topologies. We generate two sets of configurations: one with only OSPF and one with iBGP and OSPF. We run Plankton with 32 cores to verify reachability with one failure, i.e. k=1 in KFAIL. Figure 9(a) shows the time taken by Plankton (SPVP on 32 cores) and Tiramisu (ILP on 1 core) to verify this policy. Due to dependencies, Plankton (P-iBGP) performs poorly for iBGP networks. It gave an out-of-memory error for large networks. For small networks, Tiramisu (T-iBGP) outperforms Plankton by 100-300X. On networks that run only OSPF, Plankton (P-OSPF) performed better on a single network with 108 routers. Communication with the authors revealed that Plankton may have gotten lucky by quickly trying a link failure that violated the policy. Disregarding that anomaly, Tiramisu (T-OSPF) outperforms Plankton by 2-50X on the OSPF-only networks.

Batfish [13] Data-plane verifiers, like Batfish, generate data planes and verify policies on each generated data plane. ARC [14] showed that Batfish is impractical to use even for small failures. Here, we show that even without failures, Tiramisu outperforms Batfish. We run Batfish with all its optimization, on the datacenter networks to verify reachability without failures. As mentioned earlier, Tiramisu verifies this policy using *TPVP*. Figure 9(b) shows that it outperforms Batfish by 70-100X.

Bonsai Bonsai [7] introduced a compression algorithm to improve scalability of configuration verifiers like Minesweeper, to verify certain policies exclusively under **no failures**. We repeat the previous experiment to evaluate Bonsai (built on Minesweeper). Figure 9(b) shows that Tiramisu still outperforms Bonsai, and can provide speedup as high as 9X.

8.4 Scalability

Here, we evaluate Tiramisu's performance to verify PREF, KFAIL, BOUND, and, BLOCK, on large networks from the topology zoo. Figure 10 shows the time taken to verify these policies. Tiramisu can verify these policies in $<$ **0.12 s**.

For large networks, time to verify PREF (*TYEN*) is as high as BOUND. Again, this is due to larger networks having longer and more candidate paths. Large networks also have high diversity in terms of path lengths. Hence, we see more variance in the time to verify PREF compared to other policies.

For large networks, the time to verify KFAIL is significantly higher than other policies. This happens because KFAIL's ILP formulation becomes more complex, in terms of number of variables, for such large networks. As expected, verifying BLOCK is significantly faster than all other policies, and it is very fast across all network sizes.

Impact of *TYEN* acceleration. In our analysis of PREF and KFAIL, we invoke *TYEN*'s acceleration. To evaluate how much acceleration *TYEN* provided, we now measure the time taken to verify PREF on the Rocketfuel and datacenter networks, with and without *TYEN*'s optimization. Our main conclusions are (i) on small networks ($<$ 20 devices), *TYEN* provides a acceleration as high as 1.4X, and (ii) on large networks ($>$ 100 devices), *TYEN* provides acceleration as high as 3.8X.

Evaluation Summary. By decoupling the encoding from algorithms, Tiramisu uses custom property-specific algorithms with graph-based acceleration to achieve high performance.

9 Extensions and limitations

Although we describe Tiramisu's graphs in the context of BGP and OSPF, the same structure can be used to model other popular protocols (e.g., RIP and EIGRP). Additionally, virtual routing and forwarding (VRFs) can be modeled by replicating routing process vertices for each VRF in which the process participates. To verify policies for different traffic classes, Tiramisu generates a TPG per traffic class. To reduce the number of TPGs, we can compute non-overlapping packet equivalence classes (PEC) [17, 23] and create a TPG per PEC.

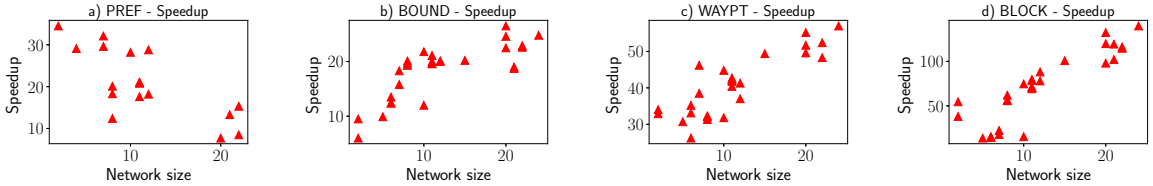


Figure 8: Speedup under no failures: Tiramisu vs Minesweeper (datacenter networks)

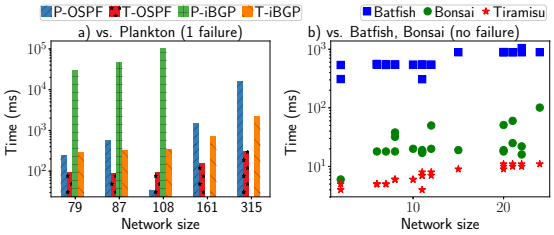


Figure 9: Comparisons with Plankton, Batfish, and Bonsai

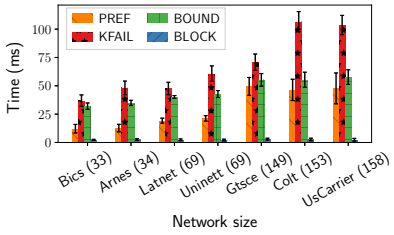


Figure 10: Performance on scale (topology zoo)

Unlike SMT-based tools, Tiramisu does not symbolically model advertisements. Consequently, Tiramisu cannot determine if there exists some external advertisement that could lead to a policy violation; Tiramisu can only exhaustively explore link failures. Tiramisu must be provided concrete instantiations of external advertisements; in such a case, Tiramisu can analyze the network under the given advertisement(s) and determine if any policies can be violated. A related issue is that Tiramisu cannot verify control plane equivalence: two control planes are equivalent, if the behavior of the control planes (paths computed) is the same under all advertisements and all failure scenarios. In essence, while Tiramisu can replace Minesweeper for a vast number of policies, it is not a universal replacement. Minesweeper's SMT-encoding is useful to explore advertisements. Additionally, Tiramisu cannot check quantitative advertisement policies—e.g., does an ISP limit the number of prefixes accepted from a peer.

Tiramisu's modeling of packet filters in the TPG only considers IP-based filtering. Consequently, in networks with protocol- or port-based packet filters, Tiramisu may over- or under-estimate reachability for packets using particular ports or protocols. Additionally, Tiramisu does not account for packet filters that impact route advertisements—e.g., filtering packets destined for a particular BGP neighbor—or route filters where multiple tags affect the same destination—e.g., community groups, AS-path filters, etc. We plan to extend Tiramisu in the future to address these limitations.

Finally, Tiramisu cannot correctly model a control plane where (1) iBGP routes lead to route deflection, and (2) an iBGP process assigns preferences (lp) or tag-based filters to routes received from its iBGP neighbors (Appendix C).

10 Related Work

We surveyed various related works in detail in earlier sections. Here, we survey others that were not covered earlier.

ERA [12] is another control plane verification tool. It symbolically represents advertisements which it propagates through a network and transforms it based on how routers are configured. ERA can verify reachability against arbitrary external advertisements, but it does not have the full coverage of control plane constructs as Tiramisu to analyze a range of policies. Bagpipe [29] is similar in spirit to Minesweeper and Tiramisu, but it only applies to a network that only runs BGP. FSR [28] focuses on encoding BGP path preferences.

Batfish [13] and C-BGP [24] are control plane simulators. They analyze the control plane's path computation as a function of a given environment, e.g., a given failure or an incoming advertisement, by conducting low level message exchanges, emulating convergence, and creating a concrete data plane. Tiramisu also conducts simulations of the control plane; but, for certain policies, Tiramisu can explore multiple paths at once via graph traversal and avoid protocol simulation. For other policies, Tiramisu only simulates a path vector protocol. Although P-Rex [18] is modeled for fast verification under failures, it focuses solely on MPLS.

11 Conclusion

While existing graph-based control plane abstractions are fast, they are not as general. Symbolic and explicit-state model checkers are general, but not fast. In this paper, we showed that graphs can be used as the basis for general and fast network verification. Our insight is that, rich, multi-layered graphs, coupled with algorithmic choices that are customized per policy can achieve the best of both worlds. Our evaluation of a prototype [2] shows that we offer 2-600X better speed than state-of-the-art, scale gracefully with network size, and model key features found in network configurations in the wild.

Acknowledgments. We thank the anonymous reviewers for their insightful comments. We also thank the network engineers that provided us real university and datacenter configurations. This work is supported by the National Science Foundation grants CNS-1637516 and CNS-1763512.

References

[1] Route selection in cisco routers. https://bit.ly/2He9zYk.

[2] Tiramisu source code. https://github.com/anubhavnidhi/batfish/tree/tiramisu.

[3] Gurobi. http://www.gurobi.com/, 2017.

[4] Widespread impact caused by Level 3 BGP route leak. https://dyn.com/blog/widespread-impact-caused-by-level-3-bgp-route-leak/, 2017.

[5] Verizon BGP misconfiguration creates blackhole. https://www.theregister.co.uk/2019/06/24/verizon_-bgp_misconfiguration_cloudflare/, 2019.

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *ACM SIGCOMM*, 2017.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *ACM SIGCOMM*, 2018.

[8] Theophilus Benson, Aditya Akella, and David Maltz. Unraveling the complexity of network management. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[9] Theophilus Benson, Aditya Akella, and Aman Shaikh. Demystifying configuration challenges and trade-offs in network-based ISP services. In *ACM SIGCOMM*, 2011.

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

[11] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical network-wide configuration synthesis with autocompletion. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[12] Seyed Kaveh Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd D. Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[13] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[14] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM*, 2016.

[15] Timothy G Griffin, F Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Transactions on Networking (ToN)*, 10(2):232–243, 2002.

[16] Timothy G Griffin and Joäo Luís Sobrinho. Metarouting. In *ACM SIGCOMM computer communication review*, volume 35, pages 1–12. ACM, 2005.

[17] Alex Horn, Ali Kheradmand, and Mukul R Prasad. Delta-net: Real-time network verification using atoms. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[18] Jesper Stenbjerg Jensen, Troels Beck Krøgh, Jonas Sand Madsen, Stefan Schmid, Jiří Srba, and Marc Tom Thorgersen. P-rex: Fast verification of mpls networks with multiple link failures. In *Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.

[19] David Karger, Rajeev Motwani, and GDS Ramkumar. On approximating the longest path in a graph. In *Workshop on Algorithms and Data structures*, pages 421–432. Springer, 1993.

[20] Simon Knight, Hung X Nguyen, Nick Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[21] David A. Maltz, Geoffrey Xie, Jibin Zhan, Hui Zhang, Gísli Hjálmtýsson, and Albert Greenberg. Routing design in operational networks: A look from the inside. In *ACM SIGCOMM*, 2004.

[22] John Moy. RFC2328: OSPF version 2, 1998.

[23] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[24] Bruno Quoitin and Steve Uhlig. Modeling the routing of an autonomous system with C-BGP. *IEEE network*, 19(6):12–19, 2005.

[25] Joao L Sobrinho. An algebraic theory of dynamic network routing. *IEEE/ACM Transactions on Networking*, 13(5):1160–1173, 2005.

[26] Joao Luis Sobrinho. Network routing with path vector protocols: Theory and applications. In *ACM SIGCOMM*, 2003.

[27] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring isp topologies with rocketfuel. In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 133–145. ACM, 2002.

[28] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. FSR: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Networking (ToN)*, 20(6):1814–1827, 2012.

[29] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016.

[30] Song Yang, Stojan Trajanovski, and Fernando A Kuipers. Optimization problems in correlated networks. *Computational social networks*, 3(1):1, 2016.

[31] Jin Y Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

A Yen's and *TYEN* Algorithm

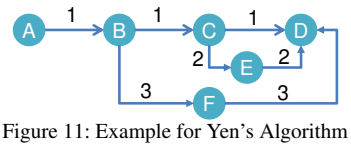


Figure 11: Example for Yen's Algorithm

Yen's algorithm. We use the graph in Figure 11 to explain this algorithm. Although line numbers below refer to *TYEN*, those lines of code also apply to the original Yen's algorithm.

Yen uses two lists: *listA* (to keep track of the shortest path seen so far) and *listB* (to keep track of candidates for the next shortest path). At the start, Yen finds the first shortest path (line 2) from *src* to *dst* using any shortest path algorithm (e.g. Dijkstra's). In Figure 11, it is $A \rightarrow B \rightarrow C \rightarrow D$. Let this path be P. Yen adds P to *listA*.

During each iteration of k, Yen takes every node n in path P (line 13, 14), and finds the *rootPath* and *spurPath* of that node. The *rootPath* of n is the subpath of P from *src* to node n (line 15). The *spurPath* is the shortest path from node n to *dst* (line 20) after making the following changes to the graph: i) to avoid loops, Yen removes all *rootPath* node except n from the graph, ii) to avoid recomputation, Yen removes all outgoing edges e from n, where e is part of any path P_A from *listA* having the same *rootPath* (line 17, 18). For example, if P_A is $A \rightarrow B \rightarrow C \rightarrow D$ and B is n, then $A \rightarrow B$ is the *rootPath* and e is $edge_{B \rightarrow C}$. By condition (i), Yen removes A and $edge_{A \rightarrow B}$. By condition (ii), it removes $edge_{B \rightarrow C}$. Then, it computes *spurPath* as $B \rightarrow F \rightarrow D$. Yen combines the *rootPath* and *spurPath* to form a new path P' (line 21) that is not blocked by tags. P' is added to *listB* if it doesn't already exist in *listA* or *listB* (line 23).

After traversing each node n in path P, Yen picks the shortest path from *listB* and reruns the previous steps with this path as the new P. This continues till Yen finds k paths.

Tiramisu Yen. To verify PREF, we make two simple modifications to Yen (*TYEN*). First, we replace Dijkstra with *TPVP*. Next, we add a condition to check that during each i^{th} iteration of k, the i^{th} path follows the preference order specified in PREF. To achieve this, *TYEN* associates each path with a variable, *eRemoved*. *eRemoved* keeps track (line 22) of edges that were removed from L3TPG (line 17-19) to compute that path. During each iteration of P, *TYEN* identifies the most preferred path specified in PREF that did not have an edge in *P.eRemoved* (line 8). If it varies from P, then preference is violated (line 9, 10).

Algorithm 3 Tiramisu Yen

Input:
G is the graph
src, *dst* are source and destination nodes
K is no. of path specified in path-preference policy
PREF, a map of preference level and path

1: **procedure** *TYEN* (G, src, dst, K)
2: $P \leftarrow$ path from src to dst returned by *TPVP*
3: $pathsSeen \leftarrow 0$
4: $P.eRemoved \leftarrow []$, as best path requires no edge removal
5: $listA \leftarrow []$, tracks paths already considered as P
6: $listB \leftarrow []$, tracks paths not yet considered
7: **while** $pathsSeen < K$ **do**
8: $mostPref \leftarrow$ most preferred path in PREF whose edges don't overlap with *P.eRemoved*
9: **if** $P \neq mostPref$ **then**
10: **return** false, since path preference is violated
11: $pathsSeen \leftarrow pathsSeen + 1$
12: add P to *listA*
13: **for** i $\leftarrow$ 0 to P.length - 1 **do**
14: $sNode \leftarrow i^{th}$ node of P
15: $rootPath \leftarrow$ subpath of P from *src* to *sNode*
16: **for each** $sp \in listA$ **do** ▷ paths in *listA*
17: **if** sp has same *rootPath* at *sNode* **then**
18: remove out edge of *sNode* in sp, so path sp is not considered
19: remove all nodes and edges of *rootPath* except *sNode* to avoid loops
20: $spurPath \leftarrow$ path from *sNode* to *dst* returned by *TPVP*
21: $P' \leftarrow rootPath + spurPath$
22: $eRemoved \leftarrow$ all edges removed in this iteration
23: add P' to end of *listB* if P' is valid and $P' \notin [listA, listB]$
24: append $P'.eRemoved$ to include *eRemoved*
25: add back all nodes and edges to the graph
26: P $\leftarrow$ remove first path from *listB*
27: **return** true, since loop didn't find preference violation

B Tiramisu ILPs

Table 2 lists the boolean indicator variables and functions used in the ILPs.

B.1 Tiramisu Min-cut

We now present the complete ILP for KFAIL (§6.1). We repeat the description of the constraints to ensure ease of reading.

The objective is to minimize the number of physical link

	Name	Description
Variable	F_e	set as 1 if edge e fails
	A_e	set as 1 if traffic propagates on edge e
	R_n	set as 1 if traffic reaches node n
	B_e	set as 1 if edge e carries blocked tag
	$T_{n,t}$	set as 1 if node n forwards tag t
Function	$nodes$	returns all nodes of graph
	$edges$	returns all edges of graph
	$dEdges$	returns all inter-device edges of graph
	$pEdges$	returns all physical edges of the network
	$oNodes$	returns all nodes except src and dst
	$iE(n)$	returns incoming edges of node n
	$oE(n)$	returns outgoing edges of node n
	$iN(n)$	returns start nodes of all incoming edges of node n
	$at(e)$	returns tags added by edge e
	$rt(e)$	returns tags removed by edge e
	$bt(e)$	returns tags blocked on edge e
	$ot(e)$	returns tags $\notin [at(e), rt(e)]$
	$start(e)$	returns start node of edge e
	$end(e)$	returns end node of edge e
	$phy(e)$	returns physical edge associated with edge e

Table 2: *Variables and Functions*

failures required to disconnect src from dst.

$$\textbf{Objective:} \quad \texttt{minimize} \sum_{i \in pEdges} F_i \tag{1}$$

Forwarding constraints. We first discuss the constraints added to represent *traffic forwarding*. The base constraint states that src originates the traffic. To disconnect the graph, the next constraint states that the traffic must not reach dst.

$$R_{src} = 1 \tag{2}$$

$$R_{dst} = 0 \tag{3}$$

For other nodes, the constraint is that traffic can reach a node n if it gets *propagated* (A_e) on any incoming edge e.

$$\forall n \in oNodes, R_n = \bigvee_{e \in iE(n)} A_e \tag{4}$$

(Logical AND/OR can be expressed as ILP constraints.)
Traffic can propagate through an edge e: if it reaches the start node of e ($start(e)$); if the traffic does not carry a tag that is blocked on e ($\neg B_e$); and if e represents an inter-device edge, the underlying physical link does not fail ($\neg F_{phy(e)}$). This is represented as

$$\forall n \in oNodes, \forall e \in iE(n): \quad A_e = R_{start(e)} \wedge \neg B_e \wedge \neg F_{phy(e)} \tag{5}$$

Tag Constraints. We now add constraints to model route tags. Route tags propagate in route advertisements from processes that add tags to processes that remove tags or block advertisements based on tags. However, the TPG models traffic propagation, which occurs in the opposite direction of route propagation. Hence, instead of modeling the flow of tags from the nodes that add them to the nodes that block/remove them, we model the flow of *blocked tags* from the nodes that block them to the nodes that remove/add them. The base constraints state that each edge that blocks on the tag "forwards" the blocked tag, and each edge that removes the blocked tag does not forward it.

$$\forall e \in edges, \forall t \in bt(e): \quad T_{e,t} = 1 \tag{6}$$

$$\forall e \in edges, \forall t \in rt(e): \quad T_{e,t} = 0 \tag{7}$$

For other edges, we add the constraint that edge e forwards a blocked tag t iff the start node of e ($start(e)$) receives traffic with t.

$$\forall e \in edges, \forall t \in ot(e): \quad T_{e,t} = \bigvee_{i \in iE(start(e))} T_{i,t} \tag{8}$$

Finally, we add the constraint that an edge e carries blocked traffic iff e receives a tag that is added by e.

$$\forall e \in edges: \quad B_e = \bigvee_{t \in at(e)} T_{e,t} \tag{9}$$

B.2 Tiramisu Longest Path

We now present the complete ILP for BOUND (§6.2). For ease of reading, we repeat our description of the constraints.

Recall that our objective is to maximize the number of inter device edges ($dEdges$) traversed by traffic (A_i).:

$$\textbf{Objective:} \quad \texttt{maximize} \sum_{i \in dEdges} A_i \tag{10}$$

Path constraints. To ensure traffic flows on only one path, src sends traffic, and dst receives traffic, we add the following constraints:

$$\sum_{out \in oE(src)} A_{out} = 1 \tag{11}$$

$$\sum_{in \in iE(dst)} A_{in} = 1 \tag{12}$$

For other nodes, we add the flow conservation property, i.e. the sum of incoming flows is equal to the outgoing flows.

$$\forall n \in oNodes: \sum_{in \in iE(n)} A_{in} = \sum_{out \in oE(n)} A_{out} \tag{13}$$

Traffic constraints. Next, we add constraints on traffic propagation. Traffic will be blocked on edge e if it satisfies the tag (B_e) constraints. This is similar to Eqn 9.

$$\forall e \in edges: A_e \leq \neg B_e \tag{14}$$

C Proofs

We first prove the correctness of *TPVP* (§C.1). This theorem is then used to prove the correctness of our ILPs(§C.2) and *TDFS* (§C.3).

C.1 *TPVP*

In this section, we prove that we correctly model and solve the stable paths problem. Griffin *et al* [15] showed that *an instance of the stable paths problems is a graph together with the permitted paths $\mathcal{P}$ at each node and the ranking functions for each node.* Hence, we first need to prove that we correctly model the permitted paths $\mathcal{P}$ and the ranking function. We will use the following lemmas to prove them. Finally, we will prove that as long as the network converges, *TPVP* finds the exact path taken in the network and hence the solution to the stable path problem.

Lemma 2. *All permitted paths $p \in \mathcal{P}$ that a routing process $v \in V$ may take in the actual network to reach the destination, exists in the TPG ($p \in TPG$).*

Proof: Consider the path $v_i \rightarrow v_{i+1} \ldots \rightarrow v_n$, where v represents a node in the graph. We will inductively prove that for any path $p \in \mathcal{P}$, a node v_i has a path to the *dst* node in the TPG, if there is an equivalent path that traverses the same routing processes (i.e. the traffic matches the RIB entries of these routing processes) and vlan-interfaces on the real network.

When $i = n$, then the node v_i represents the destination (*dst*) of traffic.

Assume there is a path from node v_{i+1} to v_n and this is modeled correctly. We will now prove that v_i is connected to v_{i+1} in the TPG if v_i can use v_{i+1} to reach the next-hop router (in its path towards the destination node). Node v_i can be one of the following types of nodes

Case (i): node v_i is an OSPF node According to §4.2, node v_{i+1} will be a VLAN-egress node. TPG connects v_i and v_{i+1} if *OSPF* is configured to operate on that *VLAN* interface. OSPF uses its configured interfaces [22] to (a) receive link-state advertisement from its neighbors, and (b) forward/send traffic towards its neighbors. Hence, OSPF will use the VLAN-egress node to forward traffic to its neighbors/next-hop router. Hence case (i) is modeled correctly.

Case (ii): node v_i is an BGP node According to §4.2, node v_{i+1} will be either an OSPF node or an VLAN-egress node. These instances represent BGP using either an OSPF-computed route or a connected subnet to reach the next hop router. Assume v_{i+1} is an *OSPF* node. BGP process is allowed to communicate with its neighboring adjacent process through an OSPF-computed path. Hence these nodes can be connected in the actual network. Assume v_{i+1} is a VLAN-egress node for VLAN interface V. TPG connects them if BGP's next-hop IP address falls within the subnet assigned to VLAN interface V. Similar to case (i), this interface is use to receive advertisements and send traffic to the next-hop router. Hence case (ii) is modeled correctly.

Case (iii): node v_i is a VLAN-ingress node According to §4.2, node v_{i+1} will be either an OSPF/BGP node representing the processes running on the same router, or an egress node for the same VLAN on that router. Assume v_{i+1} is an OSPF/BGP node. In the TPG, all traffic enters a router through a VLAN-ingress node. In the actual network, all incoming traffic looks up the global RIB of the router to move to the next-hop router. A router's global RIB contains entries from all the RIBs of all of its routing processes. The TPG connects the VLAN-ingress node to an OSPF/BGP node if that node is tainted in the RAG. Since only those processes that have a RIB entry for the *dst* are tainted, this is modeled correctly. Assume v_{i+1} is an VLAN-egress node. TPG connects them iff the underlying device is a switch. Since flooding occurs at Layer-2, this is modeled correctly.

Case (iv): node v_i is a VLAN-egress node According to §4.2, node v_{i+1} will be a VLAN-ingress node. TPG connects them if they belong to the same VLAN and their underlying routers are connected in the physical topology. This models basic physical connectivity in the real network. Hence, case (iv) is modeled correctly.

Case (v): node v_i is the *src* node According to §4.2, node v_{i+1} will be either an OSPF node or a BGP node. The *src* node is like any other VLAN-ingress node, with the difference being that the *src* originates the traffic. Hence, using similar arguments as case (iii), case (v) is modeled correctly.

Lemma 3. TPVP *will not choose a path $p \notin \mathcal{P}$ from a routing process $v \in V$ to the destination, that the routing process v cannot choose in the actual network.*

Proof: A TPG is partially based on the physical topology. In the TPG, the only inter-device edge is the edge between the VLAN-egress node of a device and the VLAN-ingress node of its neighboring device. Hence, the TPG will not have any path that does not exist in the actual physical network topology.

In the TPG, *OSPF* and *BGP* nodes are connected to a VLAN-egress node if they processes is configured to operate on that VLAN. Additionally, a VLAN-ingress node is connected to *OSPF* and *BGP* nodes if those processes are tainted in the RAG. Hence, the TPG will correctly connect the interfaces with its associated routing process, and won't have any path due to incorrect route adjacency.

Next, we prove that Tiramisu correctly models prefix/neighbor-based and tag-based filters. The TPG models prefix/neighbor-based filters by removing edges from the node associated with the process that uses those filters. Hence, the TPG will not have any path that is blocked due to prefix/neighbor-based filters in the actual network.

The only path that may exist in the TPG and not in the network is the path blocked by tags. *TPVP* uses routing algebra [16, 25] to model route computation and route selection operations. [16] showed that routing algebra can model and filter routes based on tags. Routing algebra uses "tag" attributes in their edge labels and path signatures. Their

$\oplus$ (operation) function models both addition of tags and blocking routes based on tags. Hence *TPVP* will not choose a path that is blocked by tags.
Hence *TPVP* will not choose a path p that cannot be chosen in the actual network.

Lemma 4. *L3TPG has the same set of layer 3 paths as the actual network and has no path that cannot exist in the actual network.*

Proof: Since advertisements can flow between layer 3 nodes as long as there is at least one layer 2 path that connects them, Tiramisu uses path contraction to replace layer 2 paths with a layer 3 edge.
Assume there is a layer 3 path P^* in the L3TPG that does not exist in the network. Since path contraction replaces subpaths with edges and does not add paths, P^* with (layer 2 nodes inbetween) must exist in the original TPG. I.e. If nodes are connected in the layer-3 graph, then they have to be connected even before path contraction. This contradicts Lemma 3.
Assume there is a layer 3 path $P^{\#}$ in the network that does not exist in the L3TPG. Since path contraction replaces subpaths with edges and does not eliminate paths, $P^{\#}$ with (layer 2 nodes inbetween) must not exist in the original TPG. This contradicts Lemma 2.

In the following lemmas, we operate on the contracted L3TPG.

Lemma 5. *For all routing process $v \in V$, given a set of paths $\mathcal{P}$, best$(\mathcal{P}, v)$ matches the path that v would choose when paths $\mathcal{P}$ are given as choices in the actual network.*

Proof: *TPVP* uses routing algebras to model the ranking behavior of actual protocols (*OSPF* and *BGP*). Routing algebras model route selection/path ranking using a $\preceq$ preference relation over path signatures. This correctly models the route selection algorithm in the underlying network. Hence the path chosen by *best*$(\mathcal{P}, v)$ for each routing process $v \in V$ matches the path chosen in the actual network.

Lemma 6. *For all routing process $v \in V$, the set of paths $\mathcal{P}$ considered by* TPVP *matches the paths that v can choose from in the actual network.*

Proof: Our networks may run either (i) only *OSPF* processes, (ii) only *BGP* processes, or (iii) both *OSPF* and *BGP* processes. We leverage Lemma 5 to prove the correctness for each scenario.

Case (i): OSPF-only network In the L3TPG, we have only *OSPF* nodes. From Lemma 5, we know that *OSPF* nodes will make the same choice as the actual OSPF processes. Hence case (i) is modeled correctly.

Case (ii): BGP-only network In the L3TPG, we have only *BGP* nodes. From Lemma 5, we know that *BGP* nodes will make the same choice as the actual *BGP* processes. Hence case (ii) is modeled correctly.

Case (iii): OSPF+BGP network In the layer 3 TPG, we have both *BGP* and *OSPF* nodes. Here choices made by *OSPF* or *BGP* nodes may depend on the opposite protocol's choices.

Case (iii-a): both node v_i and v_{i+1} are BGP nodes. This is similar to case (ii). Hence it is modeled correctly.
The next three cases represent instances where *BGP* uses an *OSPF* computed path to reach its next hop.

Case (iii-b): node v_i is a BGP node, node v_{i+1} is an OSPF node, and v_{i+1} is connected to a single BGP node. Since *OSPF* has only one choice, it will select the *BGP* process as its next hop and case (iii-b) is modeled correctly.

Case (iii-c): node v_i is a BGP node, node v_{i+1} is an OSPF node, and v_{i+1} is connected to multiple BGP nodes having equal preference (same cost). Since BGP *processes* are equally preferred, OSPF cost is used to select the best path/route [1]. Case (i) showed that given a set of OSPF-costs, the *OSPF* node will make the right choice. Hence, this case (iii-c) is modeled correctly.

Case (iii-d): node v_i is a BGP node, node v_{i+1} is an OSPF node and v_{i+1} is connected to multiple BGP nodes having different preferences. Since an *OSPF* node cannot make a decision using *BGP* preferences, Tiramisu cannot model this scenario. In the real network, this scenario arises when an iBGP process assigns preferences (lp) or tag-based filters to routes received from its iBGP neighbors.

Case (iii-e): node v_i is a BGP node, node v_{i+1} is an OSPF node and v_{i+1} is connected to one or multiple OSPF nodes. This scenario can only happen if *BGP* and *OSPF* have route redistribution. In this scenario, v_{i+1} makes the choice for intra-domain routing and v_i makes the choice for inter-domain routing. Using similar arguments as case (i) and case (ii), choices made by v_{i+1} and v_i will be correct.

Case (iii-f): node v_i is an OSPF node. Assume node v_{i+1} is an *OSPF* node. This is similar to case (i). Hence it is modeled correctly. Assume node v_{i+1} is a BGP node. This will lead to similar scenarios as case (iii-c) to case (iii-e). Hence, using similar arguments, case (iii-f) is modeled correctly as long as iBGP processes do not assign preferences to routes received from its iBGP neighbors.

Theorem 1. *If the network control plane converges,* TPVP *always finds the exact path taken in the network under any given failure scenario.*

Proof: Lemma 2 and 3 showed that we correctly model permitted paths $\mathcal{P}$. Lemma 5 and 6 showed that we correctly model path selection/ranking function. Now, we need to prove that under convergence, *TPVP* is equivalent to PVP.

The body of the loop code of *TPVP* (line 7 to 11) is equivalent to the *PVP* code that runs at each node [26]. The termination conditions of *PVP* and *TPVP* (line 12 to 13) are also similar. Both of them establish convergence when there are no new messages in the distributed and shared buffers respectively. And finally, both of them are executed after removing edges to represent failures.

The main difference between *PVP* [15, 26] and *TPVP* is the following: In PVP, there is no restriction on the order in which messages are processed by different routers. As long as the network converges and there exists a stable path, *PVP*

will find it irrespective of the order in which messages are sent and processed. In *TPVP*, we process and send messages in a fixed round-robin order (line 5). Since any ordering of messages in *PVP* leads to a valid solution, a fixed ordering of messages should also lead to a valid solution. Hence if the network converges, *TPVP* finds the exact path.[4]

C.2 ILP

We will first use two lemmas to prove that *TPVP* and the min-cut ILP consider the same set of paths. Recall that the min-cut ILP can be applied directly only on a TPG with no ACLs; in §6.1, we showed how it deals with ACLs.

Lemma 7. *The min-cut ILP only considers paths computed by TPVP.*

Proof: Assume the min-cut ILP considered a path that was not computed by *TPVP*. This implies the path was ignored by *TPVP* but not the ILP. This is not possible because (i) according to Lemma 2 and 3, *TPVP* correctly models permitted paths, and (ii) the ILP and *TPVP* exclude the same set of paths, i.e. both of them exclude paths based on tags. The ILP constraints on tags (Eqn 6 and 14) ensure that all paths blocked by tags are ignored by the ILP.

Lemma 8. *The min-cut ILP considers all paths computed by TPVP.*

Proof: Assume the min-cut ILP did not consider a path computed by *TPVP*. This implies the path was ignored by the ILP but not by *TPVP*. Similar to Lemma 7, this is not possible because (i) the ILP correctly models permitted paths (Lemma 2 and 3), and (ii) the ILP and *TPVP* exclude the same set of paths.

Theorem 9. *In the absence of ACLs, the min-cut ILP computes the minimal failures to disconnect the src and dst in the TPG.*

Proof: By Lemma 7 and 8, the ILP considers the same set of paths as *TPVP*. Next, we will show that this ILP computes a valid cut. Then, we will show that the cut is minimum.

Assume the ILP does not compute a valid cut. This means there is a path where all its edges A_e are equal to 1. However that also means that by Eqn 4, one of $R_{start(e)}$ is R_{dst} and R_{dst} is equal to 1. This contradicts Eqn 3.

Assume the ILP computes an invalid cut. This means there is still a path where all its edges A_e are equal to 1. Using the same argument, this will again contradict Eqn 4 with Eqn 3.

Assume the cut computed by ILP is not the minimum. This will contradict the objective function (Eqn 1) which minimizes the number of edge failures (removals) to disconnect the graph.

Next, we prove the correctness of the longest-path ILP.

Theorem 10. *The longest-path ILP computes the length of the longest inter-device path between src and dst in the TPG.*

Proof: Note that the longest-path ILP is modeled for the same TPG as the min-cut ILP. Although the ILPs are different, the constraints used to block based on tags are the same. Using similar arguments as min-cut ILP, we can prove the following two lemmas

Lemma 11. *The longest-path ILP only considers paths computed by TPVP.*

Proof: Similar to Lemma 7

Lemma 12. *The longest-path ILP considers all paths computed by TPVP.*

Proof: Similar to Lemma 8

Hence, by Lemma 11 and 12, this ILP also considers the same set of paths as *TPVP*. Next, we will show that this ILP computes a valid path. Then, we will show that this path is the longest.

Assume, the ILP finds an invalid path. This means there is some node which has an outgoing flow without any incoming flow. This contradicts the flow conservation constraint (Eqn 13). Assume, the ILP misses a valid path. This means there is some node which does not have an outgoing flow but has an incoming flow. This also contradicts Eqn 13.

Assume the path computed by ILP is not the longest. This will contradict the objective function (Eqn 10) which maximizes the number of edges traversed to reach the destination.

C.3 TDFS

Theorem 13. TDFS *identifies a src-dst pair as always unreachable iff there never exists a path from src to dst under all possible failures*

Proof: Assume there existed some path in the network that *TDFS* did not consider. This implies the path was ignored by TDFS and not *TPVP*. We will now prove by contradiction why this is not possible.

There are four types of paths that can exist in the network to establish reachability. We will show that *TDFS* will capture each of those paths.

Case (a): assume a path p_1 exists from *src* to *dst* which does not traverse any edge with tags. Assume this path is ignored by *TDFS*. Line 5 of *TDFS* runs *DFS* after removing edges that block on tags. In this case, no edges are removed. Since p_1 connects *src* to *dst*, *DFS* will return true and *TDFS* will say *dst* is reachable from *src*. This contradicts our assumption.

Case (b): assume a path p_2 exists from *src* to *dst* which does not traverse any edge that blocks on a tag. Assume this path is ignored by *TDFS*. Line 5 of *TDFS* runs *DFS* after removing edges that block on tags. Since p_2 connects *src* to *dst* without traversing any of these removed edges, *DFS* will return true and *TDFS* will say *dst* is reachable. This again contradicts our assumption.

Case (c): assume a path p_3 exists from *src* to *dst* that traverses an edge that blocks on tags but does not traverse any edge which adds tags to an advertisement. Assume this path is ignored by *TDFS*. Line 7 of *TDFS* runs *DFS* after (i) establishing that all paths go through edges that block on tags, and, (ii) removing edges that add tags. Because of (i), we know all

[4] Note that Tiramisu assumes network convergence. The routing algebra that models *TPVP* will have all the algebraic properties required for convergence.

p_3 will traverse a tag blocking edge. Since p_3 connects *src* to *dst* (and hence a tag-blocking edge to *dst*) without traversing the removed edges, *DFS* will return true and *TDFS* will say *dst* is reachable. This again contradicts our assumption.

Case (d): assume a path p_4 exists from *src* to *dst* which traverses an edge that blocks on tags, followed by an edge that removes that tag and an edge that adds that tag. Assume *TDFS* ignores this path. Line 9 of *TDFS* runs *DFS* after (i) establishing that all paths go through both edges that block on a tag and add that tag, and (ii) removing edges that remove that tag.

TDFS will return false if *DFS* states that the tag-blocking edge cannot reach the tag-adding edge. This can happen in two scenarios. In the first scenario, the tag-blocking edge could not reach the tag-adding edge even before node removal. In the second scenario, the tag-blocking edge could reach the tag-adding edge, but the removal of the tag-removing edge disconnected them. p_3 already captures the first scenario. And p_4 represents the second scenario. *TDFS* captures both these scenarios. Hence, this again contradicts our assumption.

Assume there exists some path $P^{\#}$ that is considered by *TDFS* as a valid path but does not exist in the network. As mentioned in Lemma 3, the only path that exists in the TPG and not in the network is the path blocked by tags. Hence, $P^{\#}$ must be blocked by tags. However, Line 11 of *TDFS* returns true if all paths that connect *src* to *dst* traverses a tag-blocking edge X, followed by a tag-adding edge Y, and no tag removing edge between X and Y. Hence, *TDFS* correctly identifies and ignores paths based on tags. This again contradicts our assumption.

D Other Policies

Some of the other policies that Tiramisu can verify are listed below:

Always Chain of Waypoints (CW). Similar to waypointing, we remove nodes associated with each waypoint, one at a time. Using *TDFS*, we check if nodes associated with one of the preceding waypoints in the chain can reach nodes associated with one of the following waypoints in the chain.

Equal Bound (EB). This policy checks that all paths from *src* to *dst* are of the same length. The objective of the ILP in §6.2 can be changed to find the shortest path length. If the longest and shortest path length varies, then this policy is violated.

Multipath Consistency (MC). Multipath consistency is violated when traffic is dropped along one path but blocked by an ACL on another. To support multipath in *TPVP*, we change the p_* variable to keep track of multiple most preferred path signatures to reach *dst*. Using *TPVP*, Tiramisu can identify the number of best paths to reach *dst*. We run *TPVP* on graphs with and without removing edges for ACLs. If the number of paths varies with and without ACLs, then the policy is violated.

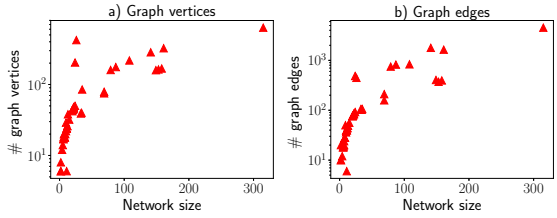


Figure 12: Size of multilayer graphs of all networks

Always no black holes (BH). Black holes occur when traffic gets forwarded to a router that does not have a valid forwarding entry. Blackholes are caused by i) ACLs: routers advertises routes for traffic that is blocked by their ACLs; ii) static routes: the next-hop router of a static route cannot reach *dst*. Tiramisu uses *TDFS* to check these conditions. For (i) Tiramisu first creates the graph without removing edges for ACLs. Let R be the router with a blocking ACL. If *src* can reach router R and R can reach *dst* (using *TDFS*), then traffic will reach router R under some failure, and then get dropped because of the ACL. For (ii) if *src* can reach the router with the static route and the next-hop router *cannot reach dst*, then the traffic gets dropped.

E Protocols/Modifiers in Network

We used university, datacenter, topology zoo, and Rocketfuel configurations in our evaluation §8. Table 3 shows what percentage of networks in these datasets support each network protocol/modifier.

	% of Networks			
Protocols/Modifiers	University	Datacenter	Topology Zoo	Rocketfuel
eBGP	100%	100%	100%	100%
iBGP	100%	0%	100%	100%
OSPF	100%	97%	100%	100%
Static routes	100%	100%	0%	0%
ACLs	100%	100%	0%	0%
Route Filters	100%	97%	100%	0%
Local Prefs	50%	0%	100%	0%
VRF	100%	0%	0%	0%
VLAN	100%	0%	0%	0%
Community	100%	100%	100%	0%

Table 3: Configuration constructs used in networks

Figure 12 characterizes the size of the TPGs generated by Tiramisu for these networks. It shows the number of nodes and edges used to represent the graph. We observe two outliers in both Figure 12a and Figure 12b. These occur for networks *Uni*2 (24 devices) and *Uni*3 (26 devices), from the university dataset. These networks have multiple VRFs and VLANs, and Tiramisu creates multiple nodes (and edges between these nodes) for different VRFs, routing processes and VLAN interfaces. Note also that for the other networks, the number of routing processes per device varies. Hence, the number of nodes and edges do not monotonically increase with network size.