2025 36th International Teletraffic Congress (ITC 36)

Scalable Verification of Routing Loops in Multi-Protocol Multi-Instance IP Networks

Youcef Magnouche, Sébastien Martin, Jérémie Leguay, Cong Mei
Huawei Technologies Ltd., Paris Research Center, France
{firstname.lastname}@huawei.com

***Abstract*—Permanent routing loops can easily arise due to misconfigurations of routing policies in IGP networks (e.g., OSPF, IS-IS), as defining loop-free policies in large-scale, multi-instance, and multi-protocol environments can be challenging for network administrators. In this paper, we present a scalable verification solution for analyzing routing configurations and policies. First, we provide examples of common routing loop scenarios and analyze their most prevalent root causes. Next, we introduce efficient algorithms to detect loops caused by routing preferences and route imports. These algorithms automatically generate explanations as output, enabling end users to address and resolve the issues. Finally, we present a performance evaluation conducted on a large IP RAN (Radio Access Network) and provide details about our implementation.**

***Index Terms*—Routing policies, routing loops, IGP, OSPF, IS-IS, Shortest-path, Dijkstra, LSA, LSP, RIB.**

I. Introduction

Autonomous Systems (AS) deployed by service operators or large enterprise networks are generally structured into areas and domains where various instances of routing protocols, such as OSPF (Open Shortest-path First) or IS-IS (Intermediate System to Intermediate System), operate. Border routers, which interconnect protocol instances, rely on routing policies to manage the advertisement and processing of routing information. Network operators meticulously configure these policies to ensure end-to-end connectivity while addressing critical considerations such as availability, performance, and security. As outlined in RFC 9067 [16], a routing policy defines how routes are imported, exported, and modified between protocol instances. However, configuring these policies is both complex and error-prone. Misconfigurations can result in severe issues, such as persistent routing loops. For example, if no traffic has yet been transmitted from a specific area of the network to a particular destination prefix, a loop may exist undetected, as it does not trigger Time-To-Live (TTL) expiration alarms for IP packets. Furthermore, even when TTL expiration alarms are received by the network management system, identifying the location of the loop and diagnosing its root cause remain a significant challenge for network operators.

To prevent misconfiguration errors, various solutions have been developed for control plane verification. Batfish [10] simulates the behavior of individual protocols to infer data plane information, allowing for comprehensive control plane analysis. Minesweeper [8] uses a formal method based on descriptive logic to examine the control plane configurations. Although these methods can verify numerous intended properties, such as node reachability, isolation, way-pointing, black holes, routing loops, bounded path length, and load balancing, they do not scale well in practical scenarios. To enhance scalability, more targeted solutions have been developed. For example, ARC [11] and Tiramisu [7] abstract the control plane with graph transformations to verify a limited set of routing properties, including security policies, reachability after failures, disjointness, and way-pointing. Despite these improvements, scalability remains a challenge, and there is no efficient method to identify the root cause of routing loops in complex multi-protocol and multi-instance scenarios, making it difficult to mitigate permanent loops quickly.

To verify routing configurations and policies at large scale, we introduce an efficient graph-based algorithmic solution specifically designed to detect permanent loops in multi-instance and multi-protocol networks. Implemented as a routing analysis module within a route reflector, this solution receives real-time updates on IP topology, routing changes, and policies. It can identify loops in networks with up to 10,000 nodes and 1 million prefixes within 5 minutes, while automatically generating explanations and actionable insights to address misconfigurations.

In the rest of this paper, we first present a set of use cases to illustrate how the misconfiguration of routing policies can result in permanent routing loops. Next, we provide a systematic overview of the most common root causes, deriving valuable insights that can help network operators quickly resolve misconfigurations. We then introduce our algorithmic solution for verifying routing policies at scale and automatically generating actionable insights to address misconfiguration issues. The algorithm framework leverages graph transformations and shortest-path computations. Finally, we present a performance evaluation conducted on a large IP RAN (Radio Access Network) instance and detail the practical tool we developed, including its implementation, graphical interface, and the insights it generates.

The paper is structured as follows. Sec. II reviews the state of the art, and Sec. III outlines key use cases for misconfigurations. Sec. IV examines common root causes and insights, while Sec. V introduces an algorithmic framework for detecting routing loops and generating explanations. Sec. VI presents our developed tool and numerical results. Finally, Sec. VII concludes the paper.

978-3-948377-03-8/19/$31.00 ©2025 ITC

II. Related work

As mentioned earlier, a number of control plane verification solutions have been proposed to verify properties such as reachability, isolation, waypoints, blackholes, disjointness, etc. They are basically 3 classes of solutions: simulation-based methods [10], formal methods [8], [15], [18], based on an SMT solver or model-checking, and graph-based methods [7], [11]. They aim to support a wide range of properties but fail to scale effectively in large-scale settings. Moreover, while most can detect loops, they are unable to explain them or provide actionable insights to mitigate them.

Batfish [10] is the closest related work to our contribution. It uses Datalog, a declarative logic programming language, to model the control plane, derive the data plane, and execute verification queries. When a counterexample is identified for an unverified property, Batfish generates a set of Datalog facts based on the counterexample packet. These facts highlight all forwarding rules the packet encountered along its path(s). Additionally, the facts include details such as the specific route used at each node, how the node learned the route (e.g., via OSPF), and how the route was derived from the configuration. However, the commercial company Intentionet, which develops Batfish, decided [9] to remove all uses of Datalog. This decision stemmed from challenges in expressing complex control plane behaviors and a lack of control over execution order, which is critical for achieving high performance and deterministic convergence.

Other methods have been proposed to avoid loops. In the context of SDN, an OpenFlow-based solution [19] was introduced to identify switches located within loops by statistically analyzing the TTL of packets, leveraging spectral analysis via Fourier transform. However, this approach is reactive and prone to false positives.

To address local transient forwarding loops, or micro-loops, in the event of link failures [13], extensions to IGP protocols have been proposed. Complementing these efforts, our work focuses on addressing permanent loops caused by misconfigured routing policies. Additional protocol extensions have also been proposed.

Furthermore, extensions involving tags or filtering methods for OSPF and IS-IS have been developed to improve the identification of Link State Advertisement (LSA) messages [3], [5] and mitigate conflicting propagation. While these methods can systematically eliminate loops, they are optional, lack standardization, and are often not implemented in devices.

In this context, our paper presents a dedicated, scalable solution for proactively analyzing routing policies in complex multi-protocol and multi-instance networks, enabling the systematic identification and mitigation of permanent loops.

III. Examples of misconfigurations

In this section, we present two use cases of popular root causes of permanent routing loops in multi-protocol multi-instance IGP networks. Before this, we review some of the basic protocol mechanisms involved. For more detailed information we refer the reader to the following RFCs [6], [14], [16] and product documentation [1], [2].

A. Related mechanisms in IGP protocols

At any border router, routes can be injected from the outside and exchanged between IGP instances. After a route has been learned by an instance, it is propagated using LSA messages in OSPF [14] and Link State Packet (LSP) messages in IS-IS [6].

External routes can be injected using a routing policy or a BGP peer. Border routers can also exchange routing information between IGP instances thanks to two mechanisms:

- **Route import** policies configured to advertise routes from one instance to another. Various options exist to control what prefixes to import and how route costs, or metrics, should be propagated or processed. In OSPF, an LSA Type 5, also called *External LSA*, is generated, after an import. Two metric types can be configured for this LSA: metric Type 1, where the IGP costs of the next links in the path are considered; and metric Type 2, where the IGP costs of the next links in the path are ignored. Indeed, Type 2 assumes that routing between AS'es is the major cost of routing a packet, and eliminates the need for conversion of external costs to internal link state metrics. In IS-IS, only Type 1 for LSPs (TLV 128 or 130) exists.
- **Peer** links that connect two IGP instances and make them somehow become one. In this case, any prefix in one instance can be reached from the other instance.

For import policies, among all the options available, the control of routing cost's propagation [1], [2] can induce loops. It is usually performed with the following two mutually exclusive parameters:

- **Inheritance**: when this option is enabled, the routing cost calculated by the upstream IGP instance is propagated.
- **Cost**: an arbitrary and user-defined import cost can be specified in the policy. In this case, it erases upstream IGP calculations. The previous link costs are ignored and replaced by an import cost (default value is 0). This option may be used for various network administration considerations (e.g. security, routing stability).

Disabling inheritance and configuring a specific cost is a popular root cause for loops [5].

Finally, a last important parameter in border routers' configuration that can yield permanent loops is the **preference** between IGP instances [4], [5]. Indeed, it defines the priority of routing information when updating the Forwarding Information Base (FIB), i.e., forwarding table, at border routers.

B. Loop cause: preferences

In the use case depicted in Fig. 1, a route for prefix 11.11.11.11 is injected on router A in instance IS-IS 31. An LSP is, then, propagated through instance IS-IS 31. In the RIB of B, route 11.11.11.11:A is inserted.

A peer connection is configured on routers A, B and D, between IS-IS instances 31 and 41. The same LSP is then propagated on IS-IS 41. In router B, a better preference for

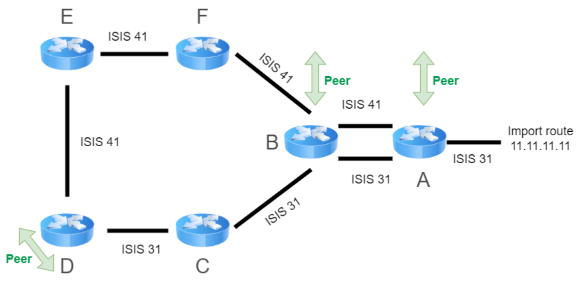


Fig. 1. Network with two IS-IS instances and peer connections configured in nodes A, B and D. At router B, IS-IS instance 41 is preferred against 31.

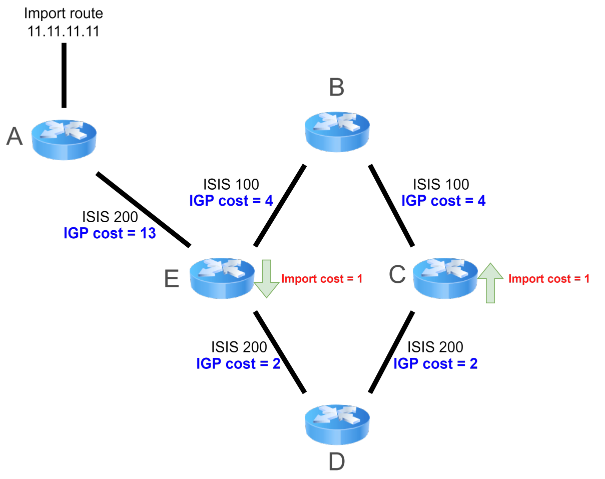


Fig. 2. Network with two IS-IS instances. Route imports are configured at nodes C and E for prefix 11.11.11.11.

instance IS-IS 41 than IS-IS 31 is configured. The RIB of B is updated by inserting a higher priority route 11.11.11.11:F. A loop is then created, B → F → E → D → C → B.

C. Loop cause: import costs

In the use case given in Fig. 2, route 11.11.11.11 is imported on router A in IS-IS 200 instance and an LSP is propagated. The RIB of router E is updated by inserting route 11.11.11.11:A. A route import to 11.11.11.11 is configured on router C from IS-IS 200 to IS-IS 100. The route import policy is configured with an import cost of 1. In this case, there is no inheritance. The same policy applies at router E. It follows that the cost of the path A→E→D→C is replaced by the import cost (from 17 to 1). A new LSP is propagated on IS-IS 100 with the new cost. At router E, the new route to 11.11.11.11 has a cost 9. E selects the new route and updates the RIB by inserting route 11.11.11.11:B. The loop is then created, E → B → C → D → E.

The two use cases are also both valid for OSPF scenarios.

IV. Root cause analysis

To analyze the root cause of routing loops more in details, let us recall that Dijkstra's algorithm [17], to compute shortest paths, relies on dynamic programming. It is optimal and solves the problem in polynomial time due to the following property:

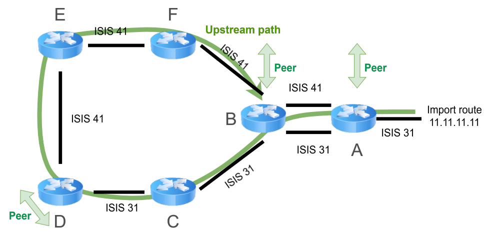


Fig. 3. Loop detection for Fig. 1's use case.

all sub-paths between two nodes, computed by the Dijkstra algorithm, are optimal paths.

In OSPF and IS-IS protocols, packets always follow the shortest path inside an instance, by definition. However, a routing loop can be induced by a routing policy when a border router prefers a path with a higher cost or if the cost of a path is suddenly set to an arbitrary value. As illustrated by use cases from Sec. III, two major root causes of permanent routing loops can happen:

a) Preferences among instances: in Fig. 1, the border router B prefers the route from IS-IS 41 rather than the route from IS-IS 31. Therefore, even if the route cost is higher, router B selects a non-optimal path. In this case, Dijkstra's property is not respected and a routing loop can appear. To solve the induced routing loop, there exist two ways. First, the preference can be removed. Second, we can ensure that the preferred route does not already cross border router B through another instance.

b) Route import without inheritance: in Fig. 2, the border router C imports the route from the IS-IS 200 to IS-IS 100, setting the cost to 1. In this case, the reduction of the cost is seen by Dijkstra as a negative cost which creates a loop. Technically, once the prefix is imported, the loop appears as follows. During LSA\LSP propagation, a border router r (node E in the example) updates the RIB with a route pr of cost x. Later, router r updates the RIB again with another route pr' of lower cost y. If pr' includes pr, a loop appears. Cost of pr' can be lower than the one of pr due to import costs between instances. To remove the loop, the following options can be proposed:

- increase the import cost by $x - y + 1$,
- increase the cost of links in $pr' \setminus pr$ such that the total of increasing cost is higher than or equal to $x - y + 1$, or
- decrease the cost of links in pr such that the sum of decreasing cost is higher than $x - y + 1$.

In the next section, we explain how to detect routing loops due to these root causes.

V. Algorithm design

We present efficient algorithms for systematically analyzing routing policies and detecting loops. The output they generate identifies the root cause and provides practical suggestions.

Let us build a new directed graph, extending the original topology, called *aggregated graph*, where each border router br is split into multiple nodes, br^i for each instance i it belongs to. Each prefix is handled by a subset of policies

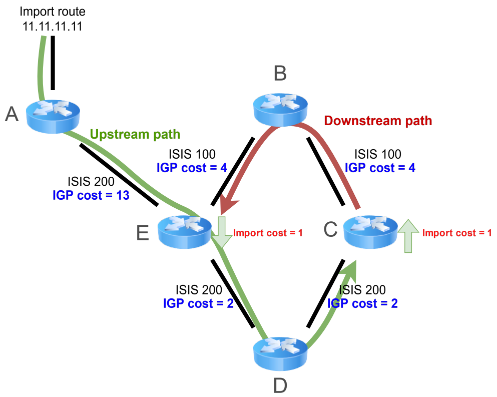


Fig. 4. Loop detection for Fig. 2's use case.

Algorithm 1 Loop detection for Preferences

```
Input:
  The aggregated network $G = (V, E)$
  $p$: prefix
Output: Set of loops $L$
  for instances $i_1, i_2$ such that $i_1$ is preferred to $i_2$ do
    for $br$ in $i_1$ and $i_2$ do
      Filter link $(br^{i_1}, br^{i_2})$ from the $G$
      $Path_1 \leftarrow$ Compute Shortest path from $p$ to $br^{i_1}$
      if $br^{i_2} \in Path_1$ then
        $L \leftarrow L \cup \{Path_1\}$
      end if
    end for
  end for
```

Algorithm 2 Loop detection for Import costs

```
Input:
  The aggregated network $G = (V, E)$
  $p$: prefix
Output: Set of loops $L$
  for $br^{i_1}$, $br^{i_2}$ with an import cost policy do
    $UpPath \leftarrow$ Compute shortest path from $p$ to $br^{i_1}$
    for $v \in UpPath$ do
      $DownPath \leftarrow$ Compute shortest path from $br^{i_2}$ to $v$
      if cost from $p$ to $v$ on $UpPath >$ cost from $br^{i_2}$ to $v$ on $DownPath$ then
        $L \leftarrow L \cup \{UpPath \cup DownPath\}$
      end if
    end for
  end for
```

at border routers and, therefore, we can consider each prefix independently. As mentioned before, shortest paths between border routers of the same instance are not impacted by border router policies. To capture routing policies and later compute routing paths, we apply the following steps to create the aggregated graph:

- compute all shortest paths between every pair of border routers belonging to the same instance,
- add an arc weighted by the shortest path cost, between every two border routers in the same instance and remove internal links,
- for each prefix, create an associated node p, and connect it to each border router copy of the instance where it is injected,
- for each border router br and between each pair of instances i_1 and i_2, links are added to capture routing policies and the way prefixes are advertised, as follows
  - for LSA/LSP type 5: an arc between br^{i_1} and br^{i_2} is added at a cost of 0,
  - for LSA type 5.2 (i.e., with a metric of Type 2): the cost of each arc from br^{i_2} to $\bar{b}r^{i_2}$ is set to 0 for every border router $\bar{b}r^{i_2} \neq br^{i_2}$ in instance i_2,
  - for a peer: add two arcs in opposite directions between br^{i_1} and br^{i_2} with cost 0,
  - for a user-defined import cost (no inheritance): an arc between the prefix p and br^{i_2} is added weighted by the import cost.

Once the aggregated graph is prepared, the following algorithms are executed for each prefix.

Algorithm 1 runs in $O(|Instances|^2 \times |BR| \times SP)$ and Algorithm 2 runs in $O(SPT + |BR| \times SPT)$ where BR is the set of border routers and SP (resp. SPT) is the shortest-path (resp. tree) problem's complexity.

A. Loop detection for Preferences

Algorithm 1 detects, locates, and explains loops caused by misconfiguration of preferences. When there are differing preferences between two instances at a border router, the algorithm computes the shortest path from the prefix to the copy of the border router of the preferred instance, call *upstream* path. If the path crosses another copy of the border router for an instance with a lower preference, a loop is detected, and recommendations for possible preference changes are issued.

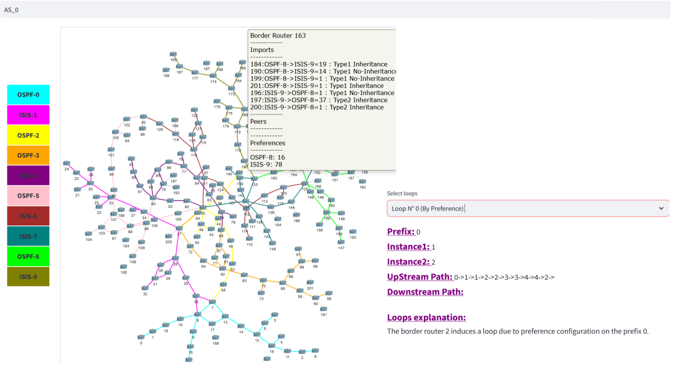
B. Loop detection for Import costs

Algorithm 2 performs the same task for import costs. The first step consists in computing an upstream path from the prefix to the border router with the policy. In the second step, *downstream* paths are computed to check if the Dijkstra property is respected, all paths from the border router with the policy to every node in the upstream path. By comparing the cost of the downstream path to node v to the cost of the upstream path to node v, we can detect if a loop will appear. Furthermore, when a loop is detected and located by the up and downstream paths, we can also deduce the root cause and get insights on how to remove the loop (see Sec. IV).

VI. Performance Evaluation

This section explains how the solution has been implemented and the outputs it delivers. We then provide numerical results, highlighting scalability.

Implementation. We developed a routing analytics module that receives topology and routing information from BGP-LS and retrieves configuration of routers using Netconf. The algorithms implemented in this module for the construction of the aggregated graph and the detection of loops extensively rely on shortest path computation. Therefore, in our implementation, algorithms have been improved using the shortest-path tree algorithm and the multi-dijkstra algorithm [12].



AS_0

```
Border Router 163
------------
Imports
------------
184:OSPF-8->ISIS-9=19 : Type1 Inheritance
190:OSPF-8->ISIS-9=14 : Type1 No-Inheritance
199:OSPF-8->ISIS-9=1 : Type1 No-Inheritance
201:OSPF-8->ISIS-9=1 : Type1 Inheritance
196:ISIS-9->OSPF-8=1 : Type1 No-Inheritance
197:ISIS-9->OSPF-8=37 : Type2 Inheritance
200:ISIS-9->OSPF-8=1 : Type2 Inheritance
------------
Peers
------------
------------
Preferences
------------
OSPF-8: 16
ISIS-9: 78
```

Select loops

Loop N° 0 (By Preference)

Prefix: 0

Instance1: 1

Instance2: 2

UpStream Path: 0->1->1->2->2->3->3->4->4->2->

Downstream Path:

Loops explanation:

The border router 2 induces a loop due to preference configuration on the prefix 0.

Fig. 5. Screenshot of the graphical interface.

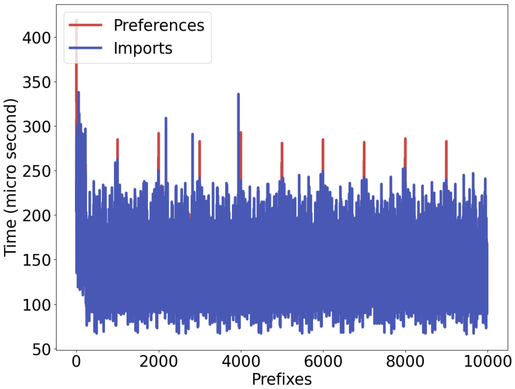


Fig. 6. Computation time for each prefix to check preferences and imports.

Fig. 5 presents the interface of the tool we developed showing the topology, routers' configuration as well as explanations about the loops found. For the two use cases considered in Sec. III, from our algorithms, we can deduce the following explanations to help the maintenance team to repair the loop.

a) For Fig 1: ”Border router B induces a loop due to the preference configuration on the prefix 11.11.11.11. Suggestion to disable the peer between instance IS-IS 31 and IS-IS 41”.

b) For Fig 2: ”Border router C induces a loop due to an Import cost for prefix 11.11.11.11. The cost of the downstream path is 9 which is smaller than the cost of the upstream path with a cost 13 to reach router E. Suggestion to enable cost inheritance at border router C”.

Numerical results. Our approach was tested in an IPRAN network to verify 10 AS, where each AS has a core network of 10 nodes, 5 aggregation networks of 50 nodes, and 15 access networks of 50 nodes. It provides a network with 10,100 nodes. Each sub-network (core, aggregation networks, access networks) is managed by a different protocol instance. The density of links in the topology is 0.4%. We considered 30 imports and 62 peers per prefix, distributed randomly at border routers. We randomly injected 10,000 prefixes at access routers. For each prefix, the two algorithms are executed to check preferences and imports. The running time of the two algorithms for each prefix is shown in Fig. 6. We need in average 0.283ms and in the worst case 0.623ms per prefix. It implies that we can detect, locate and explain loops for one million prefixes in less than 5 minutes. Furthermore, the two algorithms need less than 3Mb of RAM in total.

VII. Conclusion

We have proposed a scalable control plane verification solution that enhances reliability by preventing misconfiguration errors that lead to routing loops. Using two key use cases, we demonstrated how the solution works and the types of explanations it generates. Future work in this area can address more complex scenarios where information is incomplete, such as when only a subset of protocol instances are peered with the tool, or when static routes are injected at border routers but not reported. The solution can also be extended to support verification at the BGP/AS level.

References

[1] Huawei documentation: import-route (IS-IS). https://info.support.huawei.com/hedex/hdx.do?docid=EDOC1100277644&id=EN-US_CLIREF_0000001864283925.

[2] Huawei documentation: import-route (OSPF). https://support.huawei.com/enterprise/en/doc/EDOC1000128405/5b81628e/import-route-ospf.

[3] Redistribute OSPF Among Different OSPF Processes. https://www.cisco.com/c/en/us/support/docs/ip/open-shortest-path-first-ospf/4170-ospfprocesses.html.

[4] Redistribution between OSPF and ISIS leads to routing black hole. https://support.huawei.com/enterprise/en/knowledge/EKB1000038097.

[5] Understanding Routing Loop Detection for Routes Imported to OSPF. https://support.huawei.com/enterprise/en/doc/EDOC1100262536/41a27db4/understanding-routing-loop-detection-for-routes-imported-to-ospf.

[6] Use of OSI IS-IS for routing in TCP/IP and dual environments. RFC 1195, December 1990.

[7] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast and general network verification. *arXiv preprint arXiv:1906.02043*, 2019.

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

[9] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. Lessons from the evolution of the batfish configuration analysis tool. In *Proc. ACM SIGCOMM*, 2023.

[10] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *Proc. USENIX NSDI*, 2015.

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

[12] Roland Grappe, Mathieu Lacroix, and Sébastien Martin. The multiple pairs shortest path problem for sparse graphs: Exact algorithms. In *Proc IEEE CoDIT*, 2023.

[13] Stephane Litkowski, Bruno Decraene, Clarence Filsfils, and Pierre Francois. Micro-loop Prevention by Introducing a Local Convergence Delay. RFC 8333, March 2018.

[14] John Moy. OSPF Version 2. RFC 2328, April 1998.

[15] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *Proc. USENIX NSDI*, 2020.

[16] Y Qu, J Tantsura, A Lindem, and X Liu. RFC 9067 A YANG Data Model for Routing Policy. 2021.

[17] Alexander Schrijver. On the history of the shortest path problem. *Documenta Mathematica*, 17(1):155–167, 2012.

[18] Sooel Son, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Model checking invariant security properties in openflow. In *Proc. IEEE ICC*, 2013.

[19] Tao Yu, Longfei Yu, Diyue Chen, Hongyan Cui, and Jilong Wang. An SDN oriented loop detection mechanism based on TTL statistics. *China Communications*, 17(6):1–12, 2020.