Validating Interior Gateway Routing Protocols via Equivalent Topology Synthesis

Anonymous Author(s)

Abstract

Routers, relying on routing protocols to determine how data packets travel across the Internet, serve as the backbone of modern networks. Thus, vulnerable routing protocols can cause serious consequences, such as data leaks and network congestion. This work focuses on validating the implementation of Interior Gateway Protocols (IGPs), a key class of routing protocols. Unlike traditional protocols like TCP, which define structured data packets and state machines to facilitate communication, IGPs are designed to automatically manage the network topology. Consequently, conventional techniques, which primarily focus on communication correctness, cannot be directly applied to validate IGPs. Therefore, we propose ToDiff, a differential validation technique that uncovers IGP bugs in three steps: (1) it uses a network generation algorithm to create random yet valid networks that comply with the protocol, (2) it applies a semantics-guided and bounded program synthesis approach to generate equivalent topological programs, and (3) it simulates the network by inputting the topological programs into the IGP implementations and analyzing the routing behaviors, with any discrepancies suggesting the presence of a potential bug. We have evaluated ToDiff on the implementation of two common IGP protocols, OSPF and IS-IS. The results demonstrate that ToDiff outperforms existing approaches. To date, our tool has successfully identified 26 bugs, all confirmed or fixed by developers.

Keywords

Routing Protocols, Interior Gateway Protocols, Differential Validation, Topology Synthesis.

ACM Reference Format:
Anonymous Author(s). 2025. Validating Interior Gateway Routing Protocols via Equivalent Topology Synthesis. In . ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Unlike traditional communication protocols such as TCP/IP, routing protocols aim to automatically manage network topology, establish routing tables, adapt to network changes, and ensure robust network connectivity. Given the critical role that routing protocols play, their security becomes paramount. Vulnerabilities in these protocols can lead to severe disruptions and malicious activities. For instance, in 2018, attackers exploited vulnerabilities in a routing protocol to redirect traffic intended for Amazon's cloud service to a malicious IP address, causing users to lose at least $150,000 [41]. Thus, it is a critical task to validate the implementation correctness of routing protocols.

Existing Works. There have been a large number of works that can be used to validate routing protocol implementations; however, they are subject to various limitations. First, if we treat routing protocol implementations as common software, traditional methods, such as symbolic analysis [4, 6, 47, 52] and fuzzing [18, 20, 24], can be employed to detect general program defects like memory corruptions. However, network protocol bugs usually manifest as subtle deviations from specifications rather than obvious crashes, making their detection and analysis highly dependent on protocol-specific insights — an area where the aforementioned traditional methods often fall short.

In contrast, there are also many works such as network protocol fuzzing, e.g., [1, 29, 37, 44], differential analysis, e.g., [15, 16, 56], that integrate protocol-specific knowledge. The former generates and mutates network packets within protocol constraints to explore execution states, effectively identifying memory safety issues and packet-level errors. The latter heavily relies on multiple implementations of the same protocol and compares different implementations to detect bugs. However, both methods focus on detecting bugs related to communication among multiple parties but ignore problems in routing the communication messages. In other words, the approaches discussed above are ineffective in validating routing protocols, whose main functionality is not to exchange messages but to build a proper topology for routing.

Regarding the main functionality of routing protocols, a few techniques have been introduced, ranging from applying handcrafted topology test suites [10, 11, 13] to automated testing or model-checking techniques [25, 30, 31, 34, 35, 48, 51, 54, 55].[1] Manually crafted test suites use predefined topology cases to validate specific routing functionalities but often suffer from limited coverage and miss edge cases. Although capable of generating diverse topological structures for testing or model checking, existing automated techniques still suffer from the notorious testing oracle problem. For example, they have to rely on non-trivial manual efforts to build testing oracles, i.e., manually compute the correct routing tables and compare them to the ones derived from routing protocols.

Our Approach. To automate the validation procedure for routing protocols and address the testing oracle problem, in this work, we propose ToDiff, a differential testing technique to validate the implementation of Interior Gateway Protocols (IGPs), a critical category of routing protocols including OSPF [38] and IS-IS [5]. At the core of our approach, the key observation is that the same network topology can be established via different but equivalent commands, which we refer to as topological programs. As such, ToDiff systematically synthesizes multiple equivalent topological programs, which yield multiple network topologies. Since the synthesized topological programs are equivalent, any discrepancies in the outcome network topologies indicate potential bugs or even security vulnerabilities in IGP implementations. In a word, we reduce the testing problem to a program synthesis problem.

Conference'17, July 2017, Washington, DC, USA
2025. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

[1] We also notice that some works validate the design rather than the implementations of routing protocols [26, 33, 36, 50]. They are outside the scope of this paper.

More specifically, ToDiff works in three steps. First, a network with a random number of routers and a random topology is generated by a dedicated network generation algorithm. Despite being random, we have to follow certain constraints such that the outcome topology is valid. The generated network is referred to as the target network. Second, ToDiff synthesizes multiple and equivalent topological programs that are expected to yield the target network. To ensure equivalence and efficiency, we formally define the language semantics and apply a semantics-guided program synthesizer that is guaranteed to generate programs with bounded length. Third, by entering the topological programs into the IGP implementations, routers in the network will communicate with each other so that a correct network topology (e.g., router tables in each router) can be created at the end. Since it usually takes a long time for a topology to be built, ToDiff includes a series of heuristic yet effective methods to facilitate the convergence of the topology building procedure, thereby improving the whole testing efficiency. Finally, we compare the topologies derived from the topological programs to find potential bugs in IGP implementations.

Contributions. Putting the three steps above together yields ToDiff, a new differential analyzer capable of generating diverse networks and uncovering functionality errors in IGP implementations via automated testing oracle. Our design significantly reduces the testing complexity and makes ToDiff practical for different IGP protocols. In summary, we make the following contributions:

- We propose a novel differential testing framework to detect hidden bugs in IGP implementations.
  - It leverages a network generation algorithm that is capable of producing random yet valid routing topologies among a number of routers.
  - It features a topology program synthesizer that can generate equivalent topological programs with a bounded length, under the guidance of first-order-logic formulated semantics.
  - It designs a series of heuristic methods to accelerate network convergence and differentiate topologies, thus improving the efficiency of testing.
- We implement our approach as a tool, namely ToDiff, to validate the correctness of multiple IGP implementations. Our tool is efficient as it can produce tens of topologies in one minute and complete hundreds of differential analyses in one hour. Our tool is also effective as we have detected 26 zero-day bugs in the FRRouting project [12], a widely used and mature routing protocol suite for Linux and Unix platforms. In contrast, existing approaches detect none of the bugs. All the bugs detected have been confirmed or fixed by the developers. Notably, despite the extensive testing conducted on the protocols' implementation, we uncovered some deeply hidden bugs that had remained undiscovered for over 20 years. We will make ToDiff publicly available upon the paper is accepted.

Organization. The remainder of this paper is organized as follows. Section 2 introduces the background knowledge of IGP. Section 3 provides an overview of our approach via a motivating example. Section 4 details each step of our approach, and Section 5 discusses the evaluation results. Section 6 surveys closely related work. Section 7 concludes this paper.

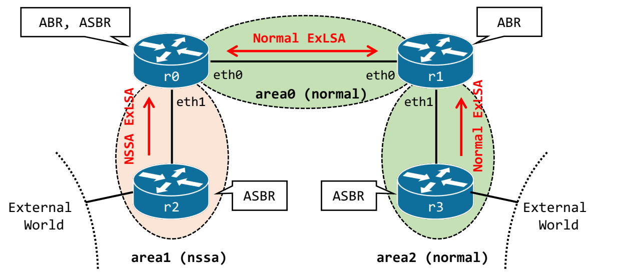


Figure 1: An OSPF network containing 4 routers and 3 areas.

```
interface r0-eth0 ip address 177.70.31.169/255.0.0.0
interface r0-eth0 ip ospf area 0

interface r0-eth1 ip address 207.235.166.37/255.255.255.0
interface r0-eth1 ip ospf area 1

router ospf area 1 nssa
```

Figure 2: An example of the topological program.

2 Preliminaries

IGP (Interior Gateway Protocol) is a class of essential routing protocols that enable efficient and secure routing within an autonomous system. By dynamically detecting and adapting to network topology changes, IGPs such as OSPF and ISIS help prevent routing loops and ensure reliable data transmission, enhancing overall network integrity. Without IGPs, networks would be more vulnerable to misconfigurations, unauthorized access, and traffic disruptions. Thus, it is a critical task to validate the correctness of IGP implementations. This section provides preliminary background on IGPs, using OSPF as an example. Other IGPs are similar to OSPF.

Routers. A network often contains many interconnected routers, denoted as r_i. Each router r_i has multiple interfaces eth_j, denoted as r_i-eth_j. Two routers are connected via interfaces, and each interface is assigned a unique IP address to enable communication among routers. For instance, Router r_0 and Router r_1 are connected via the interfaces r_0-eth_0 and r_1-eth_0 in Figure 1.

IP Address & Subnet Mask. For communication, each interface of a router should be assigned an IP address and subnet mask, both of which are 32-bit integers, but often written as $a.b.c.d$ where a, b, c, and d are the first, second, third, and fourth 8 bits of the integer, respectively. We say two interfaces are in the same subnet if the subnet masks are the same and the bitwise-and operation of the IP address and the subnet mask yields identical values. For instance, 192.168.10.11/255.255.0.0 and 192.168.13.12/255.255.0.0 are in the same subnet because the bitwise-and operation yields the same value, 192.168.0.0 and the subnet masks are the same.

OSPF Network and Topological Program. OSPF, as a routing protocol, helps routers build routing tables such that data packets can travel from their source to the destination through the most optimal or feasible routes. We refer to a network (often containing many interconnected routers) using OSPF as an <u>OSPF network</u>. OSPF partitions a network into multiple areas, say area_0, area_1, area_2, . . . , to simplify management and enhance efficiency. Each area contains a subset of interconnected routers. For example, Figure 1 shows an OSPF network with four routers and three areas: $area_0$ consists of two routers, r_0 and r_1; $area_1$ consists of two routers, r_0 and r_2; and $area_2$ consists of two routers, r_1 and r_3.

To enable area partitioning and the computation of routing tables, each router in the network installs an OSPF implementation, which then reads a topological program, often provided by the network administrators and reflecting how they want to configure the network topology. Figure 2 shows a simple topological program for Router r_0. Lines 1-2 and Lines 4-5 respectively specify the IP addresses of two interfaces, r_0-eth_0 and r_0-eth_1, as well as the areas, $area_0$ and $area_1$, they belong to. Line 7 specifies that $area_1$ is of a special type, namely NSSA, which is explained later.

According to the topological program, the network is automatically partitioned into areas by the OSPF implementation installed in each router. The routing information is also automatically computed at both the intra- and inter-area levels. Within an OSPF area, each router describes its known network topology in a data structure known as Link State Advertisement (LSA) and forwards LSAs to its neighbor routers. Upon receiving LSAs from neighbors, a router stores them in its link-state database and floods them to other neighbors. This process continues until all routers in the area have an identical view of the area's topology. In other words, each router finally knows how routers are connected in the area.

At the inter-area level, routers at the boundary of an area, known as Area Border Routers (ABRs), e.g., r_0 and r_1 in Figure 1, summarize and distribute information about one area's topology to other areas. As such, an area can have an overview of other area's topology to facilitate data packet transmission across different areas.

To sum up, we use the term OSPF network to mean a network of inter-connected routers with necessary topological information, including well-specified router types, areas with well-specified area types, and other parameters such as the IP address and the cost of each router interface. Important parameters will be detailed in Section 4. As illustrated before, an OSPF network can be described by a set of topological programs for each router in the network.

OSPF Areas. There are multiple special areas in an OSPF network. We use $area_0$ to denote the area that connects all other areas. An OSPF network ensures the existence of $area_0$. As demonstrated in Figure 1, $area_0$ connects $area_1$ and $area_2$.

An area could be a normal area, e.g., $area_0$ and $area_2$ in Figure 1, or of special types. For example, $area_1$ is of a special type known as NSSA, which does not forward complete topological information to other routers so as to reduce the workload.

OSPF LSAs. Routers maintain the topological information they have known in a data structure known as Link State Advertisement (LSA). As discussed before, routers exchange LSAs so as to have a complete view of the network topology. An LSA could be of special types. For instance, an external LSA (ExLSA) describes the topology of the network outside the OSPF network. Routers r_2 and r_3 forward ExLSA to other routers such that other routers have a view of the external world. Particularly, ExLSA traveling in a normal area is a Normal ExLSA, which is different from the one propagated in an NSSA area, i.e., NSSA ExLSA.

OSPF Routers. Routers in an OSPF network may be of different types. In addition to Area Border Routers (ABRs), which connect multiple areas, ASBR is the other important category of routers in an OSPF network. Usually, an ASBR, e.g., r_2 and r_3 in Figure 1, connects an OSPF network to the external world. However, some special routers are automatically designated as ASBRs to complete some special functionalities. For instance, in Figure 1, Router r_0 is set to ASBR such that it can translate the NSSA ExLSA into Normal ExLSA and forward the ExLSA to the normal area $area_0$.

```
1. int ospf_area_nssa_unset(...) {
2.     auto *area = ospf_area_lookup_by_area_id(ospf, area_id);
3.     ......
4.     area->NSSATranslatorRole = OSPF_NSSA_ROLE_CANDIDATE;
5.   + if (area->NSSATranslatorState == OSPF_NSSA_TRANSLATE_ENABLED)
6.   +     ospf_asbr_status_update(ospf, --ospf->redistribute);
7.     area->NSSATranslatorState = OSPF_NSSA_TRANSLATE_DISABLED;
8.     ......
```

Figure 3: A bug in an OSPF implementation.

3 Motivation and Overview

Before diving into the technical details, we first demonstrate a real-world bug detected by our tool (Section 3.1). This example shows the limitations of existing works (Section 3.2) and the merits of our methodology (Section 3.3).

3.1 Motivating Example

Let us study a zero-day bug found by our tool in an OSPF implementation from the FRRouting protocol suite, a widely used and mature routing protocol suite for Linux and Unix platforms [12]. Unfortunately, the bug has been hidden for 22 years due to the lack of effective validation techniques.

Consider the OSPF network in Figure 1. The bug happens when $area_2$ is set to NSSA and then switched back to a normal area. Like Router r_0 in $area_1$, when $area_2$ is set to NSSA, Router r_1 becomes an ASBR so that it can translate NSSA ExLSA into Normal ExLSA. When $area_2$ is switched back to a normal area, the translation is no longer needed. Thus, Router r_1 is no longer an ASBR, either. However, the OSPF implementation fails to cancel its role as an ASBR.

If we look into the code of the OSPF implementation (as shown in Figure 3), we can find that the task of canceling an NSSA area is in the function `ospf_area_nssa_unset`. The function misses Lines 5-6, which updates router to the correct role. This bug may lead to a series of network problems. For example, the bug forces ABR to generate unnecessary LSAs, which then spread widely throughout the network and increase the overhead of LSA propagation. More significantly, in some cases, external network traffic may be misdirected to this false ASBR, resulting in a traffic black hole that drops all traffic destined for a specific destination, or even more severe network security issues.

3.2 Why Existing Works Fail

General bug detection techniques such as symbolic analysis [4, 6, 47, 52] and fuzzing [18, 20] struggle to detect this bug because this bug relates to the OSPF specification rather than conventional vulnerabilities like memory corruptions. For example, missing Lines 5-6 in Figure 3 does not violate any memory safety property. Thus, the aforementioned techniques for general software bugs do not work.

Existing works such as network protocol fuzzers [1, 29, 37, 44], differential analyzers [15, 16, 56], and model checkers [2, 9, 39, 40]

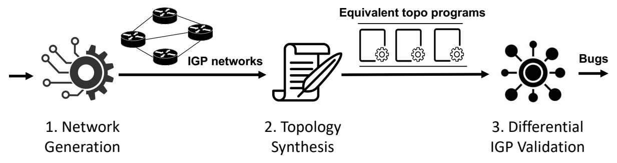


Figure 4: The overall workflow of ToDiff.

integrate protocol-specific knowledge to identify bugs. However, these methods primarily focus on issues related to network packet transmission or protocol state transition, treating the process of establishing network topology as a black box. As a result, they are incapable of identifying topology establishment errors, such as the one in the motivating example.

As discussed in Section 1, although there are a few approaches to testing routing protocols, they heavily depend on manual efforts to either construct different topologies [10, 11, 13] or build testing oracles [3, 21–23, 34, 51]. This labor-intensive procedure is error-prone and easy to miss scenarios that should be thoroughly tested. Consequently, this bug had been kept in the OSPF implementation for 22 years before we detected it.

In what follows, we provide an overview of our approach, demonstrating that ToDiff requires few manual efforts but can generate diverse network topologies and automated testing oracles to validate routing protocol implementations.

3.3 ToDiff in a Nutshell

Our approach, ToDiff, has three key advantages over existing methods. First, our approach is specially designed for routing protocols and aims to detect bugs during topology establishment. Second, our approach does not rely on predefined topologies but randomly generates diverse network topologies for testing. Third, our approach does not rely on manual efforts to build testing oracles but applies differential testing to address the oracle problem.

Despite these advantages, the practical implementation of ToDiff must still overcome several network-related challenges. Next, following the workflow shown in Figure 4, we walk through each step of our approach, demonstrating how it identifies the bug in the motivating example and discussing our strategies to address these challenges. Again, we use OSPF as an example. Other IGPs follow similar methodologies.

Step 1: Generating Random yet Valid OSPF Network. The first step is to generate a random yet valid OSPF network. While randomness ensures the diversity of scenarios where we test OSPF implementations, validity ensures the OSPF network can be used for validation. Recall that an OSPF network can be described by topological programs. Thus, the next step will synthesize multiple equivalent topological programs to specify this generated network.

While it may seem straightforward to generate a random graph where each node denotes a router and each edge represents the link between routers, such an approach could often produce invalid OSPF networks. For instance, recall that OSPF partitions a network into multiple areas, and all areas must be connected to area_0. If we randomly generate a graph like Figure 5(a) and randomly partition it into four areas, none of the four areas can play the role of area_0

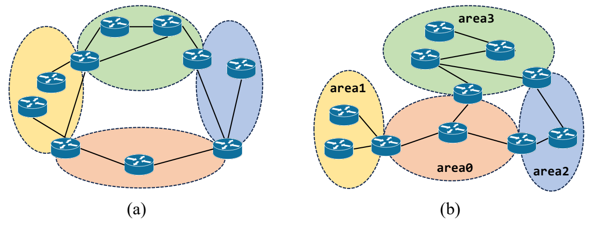


Figure 5: (a) A random yet invalid network. (b) A valid network generated by ToDiff.

because there must exist a disconnected area. In addition to the area constraint, there are also multiple other constraints a valid OSPF network must follow. Thus, the challenge below arises:

Challenge 1: How can we systematically generate random yet valid OSPF networks?

To satisfy the area constraint discussed above, we do not directly generate an OSPF network but randomly generate a list of OSPF areas, among which we randomly select one as area_0. All other areas are then connected to area_0 and are also randomly connected to each other. Figure 5(b) shows an example where the four areas are independently produced by a random graph generation algorithm, each with 4, 3, 3, and 5 routers. When connecting two areas, the area border routers are merged into one. Section 4 will detail how other constraints (e.g., IP address constraints) an OSPF network should follow are satisfied.

Step 2: Synthesizing Equivalent Topological Programs. Given the target network generated by Step 1, this step generates multiple equivalent topological programs describing the target network. For example, let us consider the OSPF network in our motivating example (see Figure 1). We can generate two topological programs for Router r_1 as shown in Figure 6. Figure 6(a) does not set the type of area_2. Thus, it is a normal area by default. In Figure 6(b), area_2 is designated as an NSSA area at Line 8, which is then canceled at Line 9. Thus, the semantics of the two topological programs are equivalent but may lead to different router behaviors, thereby helping us find bugs in the next step.

However, unlike the simple example, a real-world topological program can be highly complex, comprising a large variety of commands with different semantics. This complexity introduces a significant challenge:

Challenge 2: How can we generate equivalent topological programs consisting of a rich set of commands?

To address the challenge, we describe the target OSPF network as a first-order logic constraint, denoted as ψ, and formally define the semantics of each command in the topological program via first-order logic, too. With the semantics in hand, we can generate a sequence of random commands, say $c_1, c_2, \ldots, c_n$, and compute the aggregate semantics, denoted as a constraint ϕ. With careful guidance, ToDiff generates a topological program (i.e., a command sequence) with semantics ϕ such that n is less than a predefined constant and $\phi \equiv \psi$. In other words, the length of the synthesized

```
/* Program (a) */
interface r1-eth0 ip address 177.70.31.170/255.255.255.0
interface r1-eth0 ip ospf area 0

interface r1-eth1 ip address 207.215.156.37/255.255.255.0
interface r1-eth1 ip ospf area 2
```

```
/* Program (b) */
interface r1-eth0 ip address 177.70.31.170/255.255.255.0
interface r1-eth0 ip ospf area 0

interface r1-eth1 ip address 207.215.156.37/255.255.255.0
interface r1-eth1 ip ospf area 2

router ospf area 2 nssa
router ospf no area 2 nssa
```

Figure 6: An example of the equivalent topological programs.

topological program is bounded and it has equivalent semantics as the target OSPF network. More details can be found in Section 4.

Step 3: Determining Network Convergence and Differentiating Results. We enter the synthesized topological programs into the OSPF implementations (which are installed in the routers), routers will start to exchange topology information and build routing tables. Our goal is to compare the running results (e.g., the routers' status) of multiple equivalent topological programs. Any inconsistencies between the running results imply possible bugs in the OSPF implementations. Note that we often have to wait for a long time until the network converges, i.e., all routers have a complete view of the whole network and have successfully built the routing table. Comparing the intermediate results before network convergence is not meaningful. Thus, to improve the testing efficiency, we have to address the following challenge.

Challenge 3: How can we determine and accelerate the convergence of an OSPF network?

Since the OSPF specification does not provide a standard way to determine network convergence, we provide a few heuristic methods to determine and speed up network convergence by, for example, checking if all routers have successfully established connections to their neighbors. We provide more details in Section 4.

In the motivating example, after separately running the two equivalent topological programs generated in Step 2 and confirming the network converges according to the above rules, we observe that Router r_1 demonstrates inconsistent status — one indicates that Router r_1 is an ABR, while the other says it could be both ABR and ASBR. This inconsistency implies a bug in the OSPF implementation and helps us locate the buggy code in Figure 3.

4 Design: OSPF as an Example

This section formally details the three steps in ToDiff: generating random yet valid networks (Section 4.1), synthesizing equivalent topological programs (Section 4.2), and determining network convergence and differentiating results (Section 4.3). Note that while this section uses OSPF as an example, the methodology is general for other IGPs.

4.1 Step 1: Network Generation

The first step of our approach is to generate a random yet valid network. This includes a physical network and configurations that adhere to Ethernet protocols and routing protocol specifications. A valid network ensures that routing functionalities can be correctly activated. Below, we use a graph model to define a valid network and detail the random network generation algorithm.

4.1.1 *Valid Networks.*

Formally, we define a valid network running routing protocols as below.

Definition 4.1 (Network). A network is an undirected graph, $\mathbb{G} = (\mathbb{I}, \mathbb{L}, \mathbb{R})$, where $\mathbb{I} = \{\mathrm{eth}_0, \mathrm{eth}_1, \ldots\}$ is a set of Ethernet interfaces and $\mathbb{L} \subseteq \mathbb{I} \times \mathbb{I}$ is a set of links between the interfaces. $\mathbb{R} = \{\mathrm{r}_0, \mathrm{r}_1, \ldots\}$ denotes a set of routers in the network. A router $\mathrm{r}_i \in \mathbb{R}$ may contain multiple interfaces, denoted as $\nabla(\mathrm{r}_i) \subseteq \mathbb{I}$, and an interface can only belong to one router.

A valid network has to satisfy the following constraints, namely, unique IP address and subnet consistency, given that $\mathrm{ip}(\mathrm{eth}_i)$ and $\mathrm{mask}(\mathrm{eth}_i)$ represent the IP address and subnet mask, respectively.

Constraint 1: Unique IP Address. This constraint requires that each interface has an IP address different from all others in a network $\mathbb{G} = (\mathbb{I}, \mathbb{L}, \mathbb{R})$:

$$\forall\, \mathrm{eth}_i, \mathrm{eth}_j \in \mathbb{I} : \mathrm{ip}(\mathrm{eth}_i) \neq \mathrm{ip}(\mathrm{eth}_j).$$

Constraint 2: Subnet Consistency. This constraint requires that the interfaces at the two ends of a link are in the same subnetwork:

$$\forall (\mathrm{eth}_i, \mathrm{eth}_j) \in \mathbb{L} : \mathrm{ip}(\mathrm{eth}_i) \,\&\, \mathrm{mask}(\mathrm{eth}_i) = \mathrm{ip}(\mathrm{eth}_j) \,\&\, \mathrm{mask}(\mathrm{eth}_j) \\ \wedge\, \mathrm{mask}(\mathrm{eth}_i) = \mathrm{mask}(\mathrm{eth}_j),$$

where & denotes the bitwise-and operation.

When applying a specific routing protocol like OSPF, an interface will have more attributes, including but not limited to the following:

- $\mathrm{area}(\mathrm{eth}_i)$: the area the interface belongs to (recall that OSPF partitions a network into multiple areas).
- $\mathrm{area\_type}(\mathrm{eth}_i)$: the type of the area the interface belongs to (recall that an area could be a normal area, an NSSA area, etc.).
- $\mathrm{hello}(\mathrm{eth}_i)$: the time interval at which a router sends a Hello message to its neighbor via the interface. Routers in an OSPF network send Hello messages periodically to their neighbors to maintain connectivity.
- $\mathrm{dead}(\mathrm{eth}_i)$: the time interval at which a router considers the neighbor to be down. Routers in an OSPF network consider their neighbor is down after this time interval without receiving a hello message from their neighbor.
- $\mathrm{cost}(\mathrm{eth}_i)$: the cost measures the resource consumption to send data via this interface, which is used for traffic optimization.

At the same time, additional constraints (discussed below) should be satisfied to keep an OSPF network valid. Note that, to ease discussion, we present the most important constraints.

Constraint 3: Area Consistency. This constraint requires that interfaces at the two ends of a link are in the same area:

$$\forall (\mathrm{eth}_i, \mathrm{eth}_j) \in \mathbb{L} : \mathrm{area}(\mathrm{eth}_i) = \mathrm{area}(\mathrm{eth}_j).$$

Constraint 4: Area Connectivity. This constraint requires that interfaces in the same area should be interconnected:

$$\begin{aligned} \mathsf{area}(\mathsf{eth}_0) = \mathsf{area}(\mathsf{eth}_n) \Rightarrow \exists\ (\mathsf{eth}_0, \mathsf{eth}_1, \ldots, \mathsf{eth}_n), \mathsf{r} \in \mathbb{R} : \\ \mathsf{area}(\mathsf{eth}_i) = \mathsf{area}(\mathsf{eth}_0) \\ \wedge \quad (\mathsf{eth}_i, \mathsf{eth}_{i+1} \in \nabla(\mathsf{r}) \vee (\mathsf{eth}_i, \mathsf{eth}_{i+1}) \in \mathbb{L}). \end{aligned}$$

Constraint 5: Existence of Area 0. Recall that a valid OSPF network ensures the existence of a special area, namely area_0:

$$\exists\ \mathsf{eth}_i \in \mathbb{I} : \mathsf{area}(\mathsf{eth}_i) = \mathsf{area}_0 \wedge \mathsf{area\_type}(\mathsf{eth}_i) = \mathsf{normal}.$$

Constraint 6: Connecting to Area 0. Recall that a valid OSPF network requires all areas other than area_0 to connect to area_0 via at least one router:

$$\begin{aligned} \forall \mathsf{area}_k \neq \mathsf{area}_0, \exists\ \mathsf{r} \in \mathbb{R}, \mathsf{eth}_i, \mathsf{eth}_j \in \nabla(\mathsf{r}) : \\ \mathsf{area}(\mathsf{eth}_i) = \mathsf{area}_k \wedge \mathsf{area}(\mathsf{eth}_j) = \mathsf{area}_0. \end{aligned}$$

Constraint 7: Time Interval Consistency. Given two connected interfaces in an OSPF network, the hello and dead intervals of the two interfaces must be the same:

$$\begin{aligned} \forall (\mathsf{eth}_i, \mathsf{eth}_j) \in \mathbb{L} : \mathsf{hello}(\mathsf{eth}_i) = \mathsf{hello}(\mathsf{eth}_j) \\ \wedge\ \mathsf{dead}(\mathsf{eth}_i) = \mathsf{dead}(\mathsf{eth}_j). \end{aligned}$$

4.1.2 Random Network Generation. Algorithm 1 shows our approach to generating a random yet valid OSPF network satisfying the seven constraints discussed before. Basically, the algorithm can be split into three parts (Line 2, Lines 3-6, and Lines 7-8). The first part (Line 2) is to generate random OSPF areas. Each area is created by invoking the procedure in Lines 10-18. Lines 11-12 created random links, each connecting two interfaces, as illustrated in Figure 7(a), where each square is an interface. Lines 13-16 merge interfaces to form a router (recall that a router consists of multiple interfaces), as illustrated in Figure 7(b). In this procedure, we should ensure that all routers are interconnected, satisfying Constraints 3 and 4. Line 17 names the area as area_k and specifies the type of area, where k is an input integer ranging from 0 to a random positive integer. When $k = 0$, area_0, which must be a normal area, is created to satisfy Constraint 5.

The second part (Lines 3-6) connects the areas by invoking the procedure at Line 19, as exemplified in Figure 7(c). Particularly, Lines 3-4 ensure that all areas other than area_0 are connected to area_0, satisfying Constraint 6. Lines 5-6 randomly connect other areas. The third part (Lines 7-8) is straightforward, which is to assign random attributes to routers' interfaces, satisfying Constraints 1, 2, and 7. For instance, if we assign an IP address to an interface, the IP address cannot be assigned to others.

4.1.3 Extensions. We provide a simplified network model above to ease the explanations of approach. In practice, the model can be easily extended to include network devices such as switches and hosts. Unlike a link connecting two interfaces, a switch can be regarded as a special link connecting more than two interfaces, forming a subnet. As such, Constraint 2 should be rewritten as $\forall\ \mathsf{eth}_i, \mathsf{eth}_j \in \textbf{subnet} : \mathsf{ip}(\mathsf{eth}_i)\ \&\ \mathsf{mask}(\mathsf{eth}_i) = \mathsf{ip}(\mathsf{eth}_j)\ \&\ \mathsf{mask}(\mathsf{eth}_j) \wedge \mathsf{mask}(\mathsf{eth}_i) = \mathsf{mask}(\mathsf{eth}_j)$. In addition to switches, hosts can be viewed as special routers that do not run OSPF.

Algorithm 1: OSPF Network Generation.

```
1  procedure gen_ospf_network()
2      $\mathbb{G} \leftarrow \{\mathbb{G}_i = (\mathbb{I}_i, \mathbb{L}_i, \mathbb{R}_i) : \mathbb{G}_i \leftarrow$ gen_ospf_area($i$) for $i = 0, 1, 2, \ldots\}$;
3      foreach $\mathbb{G}_i \in \mathbb{G}$ and $i \neq 0$ do
4          con_ospf_area ($\mathbb{G}_i$, $\mathbb{G}_0$);
5      for random times do
6          con_ospf_area ($\mathbb{G}_i$, $\mathbb{G}_j$) where $i \neq j$;
7      foreach interface eth in $\mathbb{G}$ do
8          assign ip(eth), mask(eth), hello(eth), dead(eth) with respect to
             Constraints 1, 2, and 7;
9      return $\mathbb{G}$;
10 procedure gen_ospf_area($k$)
11     $\mathbb{I} \leftarrow$ a random set of interfaces;
12     $\mathbb{L} \leftarrow$ a random subset of $\mathbb{I} \times \mathbb{I}$ such that links do not share interfaces;
13     $\mathbb{R} \leftarrow \emptyset$;
14     for random times until there's a path between any pair of interfaces do
15         $\mathbb{I}' \leftarrow$ a random subset of $\mathbb{I}$ such that all interfaces in $\mathbb{I}'$ are from
             different links and have not been assigned to a router;
16         $\mathbb{R} \leftarrow \mathbb{R} \cup \{\mathsf{r}\}$; $\nabla(\mathsf{r}) \leftarrow \mathbb{I}'$;
17     $\forall$eth $\in \mathbb{I}$ : area(eth) $\leftarrow$ area$_k$; area_type(eth) $\leftarrow$ normal or nssa;
18     return $\mathbb{G} = (\mathbb{I}, \mathbb{L}, \mathbb{R})$;
19 procedure con_ospf_area($\mathbb{G}_i$, $\mathbb{G}_j$)
20     randomly pick two routers from $\mathbb{G}_i$ and $\mathbb{G}_j$, respectively;
21     merge the two routers into one;
```

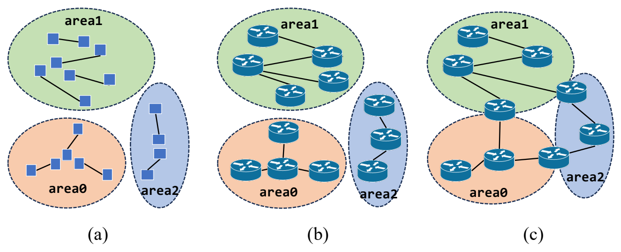


Figure 7: Example to show the network generation algorithm.

It is also easy to extend our network model to other IGPs, such as IS-IS. Section 5 reports our experimental results on both OSPF and IS-IS, the two most popular IGPs.

4.2 Step 2: Topological Program Synthesis

Given a random yet valid network generated in Step 1, Step 2 generates multiple equivalent topological programs to specify this network, via a novel semantics-guided program synthesis. This subsection consists of two parts, explaining the syntax and semantics of each command in a topological program (Section 4.2.1) and the program synthesis algorithm (Section 4.2.2), respectively.

4.2.1 Syntax and Semantics of a Topological Program. Each IGP implementation provides certain commands to control the network topology. While the commands could be different (just like different programming languages such as C/C++ and Java), they follow a similar syntax. Figure 8 illustrates an abstract syntax of topological programs for OSPF.

Basically, a topological program is a list of commands as demonstrated by the first rule in the grammar. To ease the explanation, the syntax is simplified to include a subset of important commands:

Program	:=	**Command**+
Command	:=	**Router** | **Interface**
Router	:=	router ospf no? **RouterSubCmd**
RouterSubCmd	:=	area <num> nssa
		| network <ip>/<mask> area <num>
Interface	:=	interface <r_i-eth$_j$> no? **InterfaceSubCmd**
InterfaceSubCmd	:=	ip address <ip>/<mask>
		| ip ospf area <num>
		| ip ospf cost <num>
		| ip ospf hello-interval <num>
		| ip ospf dead-interval <num>
<ip>, <mask>, <num>	∈	$\{0, 1, 2, \ldots\} \cup \{\top\}$

Figure 8: Syntax of a Topological Program.

either a router command or an interface command. A router command can control all interfaces and other information of a router while an interface command controls a single interface. In these commands, <ip>, <num>, and <mask> are all integers with a special value $\top$ meaning an undefined value.

A router command starts with the keywords "router ospf" or "router ospf no", followed by a sub-command. Assume <num> = k. A sub-command in the simplified syntax could update the type of an area, i.e., area$_k$, to a special type, namely NSSA, or specify that all interfaces, eth$_i$, satisfying ip(eth$_i$) & <mask> = <ip> & <mask>, belongs to area$_k$. If a router command contains the keyword "no", it performs an inverse operation of the sub-command. For instance, Line 9 in Figure 6 is a router command that cancels area$_2$'s role as an NSSA. The detailed semantics of a router command are listed in Table 1 (see ID 1-4).

An interface command configures the interface eth$_j$ of the router r$_i$. The corresponding sub-commands sets the IP address, area, cost, hello interval, and dead interval, respectively. A typical example is illustrated in Figure 6, where Lines 1-2 set the IP address of the interface r$_1$-eth$_0$. Similar to a router command, if an interface command contains the keyword "no", it performs an inverse operation of the sub-command. The detailed semantics of an interface command are listed in Table 1 (see ID 5-14).

In a topological program, the effects of a command c_i occurring before the other c_j could be covered by the effects of c_j. More complicated, the effects of a command could cover the partial effects of multiple commands. For instance, in Figure 9, Lines 2, 5, and 8 set the areas of the interfaces eth$_0$, eth$_1$, and eth$_2$ to area$_1$, area$_0$, and area$_0$, respectively. Line 10 affects all three interfaces — because all three IP addresses start with 177 — and changes the areas to area$_2$, area$_2$, and area$_2$, respectively. Line 11 only changes the areas of eth$_1$ and eth$_2$ to area$_1$ as their IP addresses start with 177.235.166 while eth$_0$ does not. Finally, the areas of the three interfaces are area$_2$, area$_1$, and area$_1$, respectively.

In practice, there are many complicated commands as above. Although the syntax used in this section is simplified to ease explanation, it covers sufficient and non-trivial cases we often encounter in practice. In the next part, we use the semantics of these commands to guide equivalent program synthesis.

4.2.2 Equivalent Program Synthesis. Given a valid network generated by Step 1, we now synthesize multiple equivalent topological

```
 1. interface r0-eth0 ip address 177.70.31.169/255.0.0.0
 2. interface r0-eth0 ip ospf area 1
 3.
 4. interface r0-eth1 ip address 177.235.166.37/255.255.0.0
 5. interface r0-eth1 ip ospf area 0
 6.
 7. interface r0-eth2 ip address 177.235.166.38/255.255.0.0
 8. interface r0-eth2 ip ospf area 0
 9.
10. router ospf network 177.0.0.0/255.0.0.0          area 2
11. router ospf network 177.235.166.0/255.255.255.0  area 1
```

Figure 9: Example to explain some complicated commands.

Algorithm 2: Topological Program Synthesis for a Router.

```
1 procedure synthesize_program($\psi \equiv \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$)
2     $\mathbb{U} \leftarrow \{\phi_1, \phi_2, \ldots, \phi_n\}$;
3     $\mathbb{S} \leftarrow \emptyset$;
4     $\mathbb{P} \leftarrow$ empty list;
5     while $\mathbb{U} \neq \emptyset$ do
6         $c \leftarrow$ select_command($\mathbb{U}, \mathbb{S}$) ;          /* Alg. 3 */
7         add command $c$ to the tail of $\mathbb{P}$;
8     return $\mathbb{P}$;
```

programs for each router in the network. Each synthesized program should drive a router to the expected status in the network. The expected status of a router can be formulated as a conjunctive first-order logic formula, $\psi = \wedge_i \phi_i$, where each ϕ_i specifies the value of router interfaces' attributes. For instance, Step 1 may generate the network in Figure 1, where r$_0$ is expected to satisfy $\psi \equiv$ ip(eth$_0$) = 177.70.31.169 $\wedge$ mask(eth$_0$) = 255.0.0.0 $\wedge$ area(eth$_0$) = area$_0$ $\wedge$ area_type(eth$_0$) = normal $\wedge$ ip(eth$_1$) = . . . $\wedge$ . . . .

As such, we send the target constraint ψ of a router to Algorithm 2 such that it synthesizes a random program, which can drive the router to the expected status denoted by ψ. Invoking Algorithm 2 multiple times yields multiple equivalent programs. Lines 2-3 of Algorithm 2 creates two sets, $\mathbb{U}$ representing the target constraints we expect to satisfy but have not been satisfied, and $\mathbb{S}$ is a set of constraints we expect to satisfy and have been satisfied. Clearly, the set $\mathbb{U}$ contains all ϕ_i in the target constraint ψ. Line 4 initializes an empty command list to represent the synthesized topological program. Each iteration in the follow-up loop (Lines 5-7) produces one command until $\mathbb{U} = \emptyset$, meaning that all expected constraints have been satisfied.

Algorithm 3 presents the procedure of command selection invoked at Line 6 of Algorithm 2. This is the key part of our program synthesis, which guarantees the synthesis procedure can terminate by generating a program whose length is bounded.

Termination Guarantee. Line 2 of Algorithm 3 selects a random command, and Line 3 checks if this command is good enough via Algorithm 4. The criteria of a good command are discussed later. If a command is good enough, we choose this command and update $\mathbb{U}$ and $\mathbb{S}$ at Line 4-8. In detail, Lines 4-5 move a constraint ϕ from $\mathbb{U}$ to $\mathbb{S}$ if ϕ is satisfied by the command, i.e., $[\![c]\!] \Rightarrow \phi$. Recall that $[\![c]\!]$ denotes the constraint a command implies and is listed in Table 1. Similarly, Lines 6-7 move a constraint ϕ from $\mathbb{S}$ to $\mathbb{U}$ if ϕ is dissatisfied by the command, i.e., $[\![c]\!] \wedge \phi \equiv$ *false*.

If a command is not good enough, Line 10 randomly selects a constraint from $\mathbb{U}$. Lines 11-12 generate a command to satisfy

Table 1: Command Semantics for Router r_i.

ID	Command c	Semantics $[\![c]\!]$
1	router ospf area <num> nssa	$\forall \text{eth}_i \in \nabla(r_i), \text{area}(\text{eth}_i) =$ <num> : $\text{area\_type}(\text{eth}_i) =$ nssa
2	router ospf no area <num> nssa	$\forall \text{eth}_i \in \nabla(r_i), \text{area}(\text{eth}_i) =$ <num> : $\text{area\_type}(\text{eth}_i) =$ normal (by default)
3	router ospf network <ip>/<mask> area <num>	$\forall \text{eth}_i \in \nabla(r_i), \text{ip}(\text{eth}_i)$ & $\text{mask}(\text{eth}_i) =$ <ip> & <mask> : $\text{area}(\text{eth}_i) =$ <num>
4	router ospf no network <ip>/<mask> area <num>	$\forall \text{eth}_i \in \nabla(r_i), \text{ip}(\text{eth}_i)$ & $\text{mask}(\text{eth}_i) =$ <ip> & <mask> : $\text{area}(\text{eth}_i) = \top$
5	interface <r_i-eth$_j$> ip address <ip>/<mask>	$\text{ip}(\text{eth}_j) =$ <ip> $\wedge\ \text{mask}(\text{eth}_j) =$ <mask>
6	interface <r_i-eth$_j$> no ip address <ip>/<mask>	$\text{ip}(\text{eth}_j) = \top \wedge \text{mask}(\text{eth}_j) = \top$
7	interface <r_i-eth$_j$> ip ospf area <num>	$\text{area}(\text{eth}_j) =$ <num>
8	interface <r_i-eth$_j$> no ip ospf area <num>	$\text{area}(\text{eth}_j) = \top$
9	interface <r_i-eth$_j$> ip ospf cost <num>	$\text{cost}(\text{eth}_j) =$ <num>
10	interface <r_i-eth$_j$> no ip ospf cost <num>	$\text{cost}(\text{eth}_j) = 10$ (by default)
11	interface <r_i-eth$_j$> ip ospf hello-interval <num>	$\text{hello}(\text{eth}_j) =$ <num>
12	interface <r_i-eth$_j$> no ip ospf hello-interval <num>	$\text{hello}(\text{eth}_j) = 10$ (by default)
13	interface <r_i-eth$_j$> ip ospf dead-interval <num>	$\text{dead}(\text{eth}_j) =$ <num>
14	interface <r_i-eth$_j$> no ip ospf dead-interval <num>	$\text{dead}(\text{eth}_j) = 40$ (by default)

Algorithm 3: Command Selection for Program Synthesis.

```
1  procedure select_command($\mathbb{U}, \mathbb{S}$)
2      $c \leftarrow$ randomly generate a command;
3      if check_command($\mathbb{S}, c$) = Good then
4          foreach $\phi \in \mathbb{U}$ do
5              if $[\![c]\!] \Rightarrow \phi$ then $\mathbb{U} \leftarrow \mathbb{U} \setminus \{\phi\}$; $\mathbb{S} \leftarrow \mathbb{S} \cup \{\phi\}$;
6          foreach $\phi \in \mathbb{S}$ do
7              if $[\![c]\!] \wedge \phi \equiv false$ then $\mathbb{U} \leftarrow \mathbb{U} \cup \{\phi\}$; $\mathbb{S} \leftarrow \mathbb{S} \setminus \{\phi\}$;
8          return c;
9      else
10         $\phi \leftarrow$ random constraint from $\mathbb{U}$;
11         $c \leftarrow$ generate one command to satisfy $\phi$;
12         $\mathbb{U} \leftarrow \mathbb{U} \setminus \{\phi\}$; $\mathbb{S} \leftarrow \mathbb{S} \cup \{\phi\}$;
13         return c;
```

Algorithm 4: Command Check for Program Synthesis.

```
1  procedure check_command($\mathbb{S}, c$)
2      foreach $\phi \in \mathbb{S}$ do
3          if $[\![c]\!] \wedge \phi \equiv false$ and $\#(\phi) > k$ then
4              return Bad;
5      return Good;
```

this constraint and move it from $\mathbb{U}$ to $\mathbb{S}$. Note that we can always generate a command to satisfy a given constraint. As an example, the second command in Figure 9 can always set the area of interface r_0-eth$_0$ to area$_1$.

As for the criterion of a good enough command, Algorithm 4 provides a solution. In the algorithm, we check if the command c will dissatisfy a constraint ϕ in $\mathbb{S}$ at Line 3. If so, the constraint ϕ will be moved from $\mathbb{S}$ to $\mathbb{U}$, which is not an expected operation because the goal of our algorithm is to satisfy constraints in $\mathbb{U}$ (i.e., reduce the size of $\mathbb{U}$). Thus, Line 3 also checks the other condition, i.e., $\#(\phi) > k$, where $\#(\phi)$ denotes the times of we move ϕ into $\mathbb{U}$ and k is a predefined upper bound of $\#(\phi)$.

As such, the algorithm keeps a trend that constraints are moved from $\mathbb{U}$ to $\mathbb{S}$. In other words, $\mathbb{U}$ will gradually decrease in size until it becomes an empty set, jumping out of the loop in Algorithm 2 and terminating the algorithm.

Example 4.2. Assume $\mathbb{U} = \{\phi_1, \phi_2\}$ and $\mathbb{S} = \{\}$ at the beginning of the program synthesis procedure, i.e., Algorithm 2. Our goal is to generate commands to satisfy ϕ_1 and ϕ_2, thereby moving them from $\mathbb{U}$ to $\mathbb{S}$. If we always generate random commands, e.g., $c_1, c_2, c_3, \ldots$, the constraint ϕ_1 could be satisfied by $c_1, c_3, c_5, \ldots$ and dissatisfied by $c_2, c_4, c_6, \ldots$. Consequently, ϕ_1 is constantly moved between the two sets, and there may be no chance for ϕ_2 to be moved out of $\mathbb{U}$.

In our algorithm, we limit the times, i.e., k, of moving ϕ from $\mathbb{S}$ to $\mathbb{U}$. For instance, if $k = 1$, after generating at most three commands (one moving ϕ_1 from $\mathbb{U}$ to $\mathbb{S}$, one moving from $\mathbb{S}$ to $\mathbb{U}$, and one from $\mathbb{U}$ to $\mathbb{S}$ again), we cannot dissatisfy ϕ_1 any longer and, thus, have a chance to generate commands for ϕ_2, thereby clearing the set $\mathbb{U}$ and terminating the algorithm.

Lemma 4.3. *Algorithm 2 can always terminate.*

Boundedness Guarantee. Recall the previous example and that k is a predefined constant limiting the times of moving a constraint ϕ from $\mathbb{S}$ to $\mathbb{U}$. Thus, we generate at most $2k + 1$ commands to move one constraint ϕ from $\mathbb{U}$ to $\mathbb{S}$. That is to say, we generate at most $(2k+1)|\mathbb{U}|$ commands to move all constraints from $\mathbb{U}$ to $\mathbb{S}$, synthesizing a program of length at most $(2k+1)|\mathbb{U}|$.

Lemma 4.4. *The length of a synthesized program is up to $(2k + 1)|\mathbb{U}|$, where k is a predefined constant.*

Putting the previous two lemmas together, we have the following theorem to conclude our program synthesis algorithm.

Theorem 4.5. *Algorithm 2 can always terminate by synthesizing a topological program whose length is up to $(2k + 1)|\mathbb{U}|$, where k is a predefined constant.*

4.3 Step 3: Differential Analysis

The third step is to input the synthesized topological programs into the routers, which then communicate with each other to exchange topological information until all routers compute complete routing tables. After the network converges, we compare the execution results of equivalent topological programs. Any inconsistency in the execution results indicates potential bugs.

4.3.1 *Network Simulation.*

To facilitate the testing procedure, we do not use physical routers and networks. Instead, the routers, as well as the networks, are simulated by the open-sourced Mininet framework [27]. Mininet also allows us to install different routing protocol implementations into the simulated routers.

We then input the synthesized topological programs into the routers such that the protocol implementations can run the topological programs. The entire program is randomly split into several subsequences. At random intervals, a subsequence is entered until the entire program has been input. This is because a command may take some time to take effect. If commands are entered too quickly, earlier commands may be overwritten by later ones before they have had a chance to take effect.

4.3.2 *Determining & Accelerating Network Convergence.*

After inputting the topological programs into the routers, it usually takes some time for the routers to exchange information and calculate routing data. We have to wait until this procedure is completed, i.e., the network converges — all commands in the topological programs have been executed, and all routers finish computing the routing table. Checking the intermediate status of the network is less meaningful for differential testing because a network may converge through different intermediate statuses.

Unfortunately, it is challenging to determine if a network converges because the specification of routing protocols, e.g., OSPF, does not explicitly define how to determine and speed up network convergence. To address this challenge, we use a few heuristic methods to estimate and accelerate convergence, as discussed below. The effectiveness of these methods is proven in our experiments, as discussed in Section 5.

Strategy 1: Neighbor Establishment. Like the TCP [14] handshake process, each router running OSPF maintains a state machine when the router connects to another. When the state machine reaches a final (a.k.a., terminated) state, the connection completes. Thus, to determine if a network converges, we check all routers' neighboring states to see if the states have been set to the final states.

Strategy 2: LSA Exchange. Recall that routers exchange LSAs to build routing tables. When a network converges, all LSA transmission queues in routers should be empty, meaning that a router no longer needs to exchange LSAs with others. Thus, we check if LSA transmission queues are empty to determine network convergence.

Strategy 3: Immediate Routing Calculation. By default, a router may calculate its routing table after some time to collect multiple LSAs from other routers. To speed up network convergence, we let all routers compute their routing table immediately after receiving an LSA from other routers.

Strategy 4: Time Interval Reduction. To establish a connection to neighboring routers and exchange LSAs, a router sends a packet, e.g., a Hello/LSA packet, to its neighbors periodically, e.g., every n seconds. Since $n > 5$ by default in most cases, which is too long and slows down network convergence, we set $n = 0.5$ to speed up convergence as well as the whole differential testing procedure.

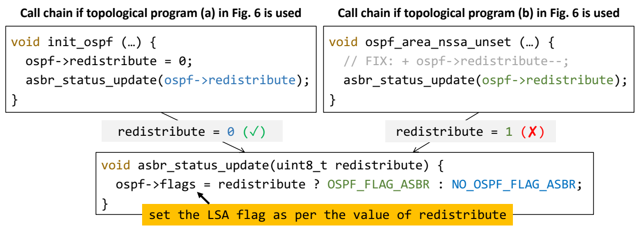
4.3.3 *Differential Analysis.*

After running multiple equivalent topological programs, we collect the execution results from each router and compare the execution results derived from equivalent topological programs. In practice, there are a lot of execution results but not all of them can be used in differential analysis. For example, each router maintains the routing information as a database of LSAs and the age of each LSA (i.e., when an LSA is created). The ages of an LSA across different executions may differ, but the core contents of LSAs should be the same. Thus, in the differential analysis, we do not compare the LSA ages but other information that should be the same across different executions, which we refer to as the differential oracles.

Table 2: Differential Oracle in Five Groups.

Group	Fields	Description
1. Commands	-	Effective commands in a topological program
2. Interfaces	IfAddress	IP address of the interface
	IfType	Interface type (e.g. broadcast, point-to-point, etc.)
	IfArea	Area the interface belongs to
	IfCost	Cost of the interface
	...	
3. Neighbors	NbState	State of a neighboring router
	NbPriority	Priority of the neighbor in the connection
	NbRole	Role of a neighbor, e.g., ABR, ASBR
	NbLXRe	Length of neighbor's retransmission packet list
	...	
4. LSAs	LsaOption	Various options of a router
	LsaFlags	Router's role, e.g., ABR, ASBR
	LsaType	Types of the LSA
	LsaID	ID of the router sending the LSA
	...	
5. Routes	RtDst	Destination address of the route
	RtCost	Cost to reach the destination address
	RtNxtHop	Next router in the route to the destination
	RtIntf	The interface via which a packet should be sent out
	...	

Table 2 lists important differential oracles used in our approach, which can be put into five groups. First, a router in an OSPF network often outputs commands (from a topological program) that really take effect, i.e., the commands whose effects are not canceled by other commands. For instance, the command at Line 8 in Figure 6 is not included in the output commands because it is canceled by the command at Line 9. Thus, given two equivalent topological programs, the output effective commands should be the same. If comparing the final effective commands yields any inconsistency, it indicates some bugs in the protocol's implementation. Second, interface Information that can play the role of differential oracles includes an interface's IP address, type, area, cost, to name a few. Third, a router maintains the information for each of its neighbors. Thus, in the differential analysis, we can compare a router's neighbor information, including but not limited to the neighboring router's state, priority, type, etc. Fourth, recall that routers exchange LSAs (see Section 2) to compute a complete topology of the whole network. Each router records the LSAs it receives from other routers in a database. Thus, in the differential analysis, we can compare the LSAs recorded by routers, including LSA's sources, flags, types, etc. Last but not least, we compare the routing tables computed across multiple executions of equivalent topological programs. An entry in a routing table includes the destination a route, the cost of sending a packet to the destination, the next hop to send a packet, and many others.


Call chain if topological program (a) in Fig. 6 is used
Call chain if topological program (b) in Fig. 6 is used
redistribute = 0 (✓)
redistribute = 1 (✗)
set the LSA flag as per the value of redistribute

```
void init_ospf (…) {
  ospf->redistribute = 0;
  asbr_status_update(ospf->redistribute);
}
```

```
void ospf_area_nssa_unset (…) {
  // FIX: + ospf->redistribute--;
  asbr_status_update(ospf->redistribute);
}
```

```
void asbr_status_update(uint8_t redistribute) {
  ospf->flags = redistribute ? OSPF_FLAG_ASBR : NO_OSPF_FLAG_ASBR;
}
```

Figure 10: Example to show the process of root cause analysis.

4.3.4 *Root Cause Analysis.*

Once ToDiff detects a discrepancy between router fields (see Table 2) across equivalent topological programs, we perform a lightweight root cause analysis to identify bugs in the IGP implementation under test. Specifically, given two equivalent topological programs that lead to inconsistent router fields, we run both in parallel and perform single-step debugging using GDB [17], carefully monitoring their execution step by step until the internal program states begin to diverge. To narrow down the scope of debugging, we apply program slicing [49] to the inconsistent router fields reported by ToDiff. This allows us to extract and track the relevant variables and control paths associated with the divergence, significantly reducing the amount of code that needs to be examined.

Example 4.6. Let us take the bug discussed in Section 3 as an example to illustrate root cause analysis. Recall that we executed two equivalent topological programs in Figure 6, and observed a discrepancy in the LSA flag: one indicates that a router is both an ABR and an ASBR, while the other shows that it is only an ABR.

Since the struct field `ospf->flags` in the code captures the LSA flag of routers, we apply a program slicer to identify all statements and variables related to `ospf->flags`, including the function parameter `redistribute` and another struct field `ospf->redistribute` shown in Figure 10. When debugging, we track the values of these variables until their values diverge.

As illustrated in Figure 10, the LSA flag is set in the function `asbr_status_update` based on the value of `ospf->redistribute`. If this value is not zero, the ASBR bit in `ospf->flags` is set; otherwise, it is unset. When debugging, we find that the function `asbr_status_update` is invoked in different call sites: one topological program lets the function be invoked by `init_ospf`, where `ospf->redistribute = 0`; the other equivalent topological program lets the function be invoked by `ospf_area_nssa_unset`, where `ospf->redistribute = 1`. This difference explains the divergence in the LSA flag and identifies the root cause of the bug.

In our experience, the root cause analysis illustrated above typically allows us to identify root causes in 30 minutes. We believe that protocol developers, being more familiar with the protocol implementation details, could identify such issues even faster, often within just a few minutes. In the future, we plan to integrate existing automatic root cause analyzers, e.g., [53], into ToDiff to fully automate the whole validation procedure.

4.4 Discussion

ToDiff is fully automated except for a few manual efforts before and after the differential analysis. First, before applying ToDiff, we must follow the official documents of IGPs to establish the constraints of an IGP network, as well as the semantics of IGP commands. Establishing these network constraints and command semantics allows us to build valid networks, thus avoiding the waste of resources exploring invalid networks and invalid topological programs in Steps 1 and 2. Note that the manual work above is a one-time effort and is common among modern automated techniques. For instance, automatic program analyzers, such as those for C/C++ programs, also require a one-time manual effort to model C/C++ semantics. Second, since root cause analysis is not the focus of this paper, we currently rely on a lightweight manual process to identify root causes after the differential analysis. In the future, we can replace it with automated root cause analyzers, e.g., [53], to fully automate our approach.

5 Evaluation

We implement our approach as a tool named ToDiff on top of the Mininet framework [27], an open-source network framework commonly used for simulating network execution. Specifically, Mininet allows us to build a virtual network with multiple routers on a single server and run different routing protocol implementations. We will make ToDiff publicly available after the paper is accepted. To demonstrate the efficacy of our approach, we conduct experiments with ToDiff to address the following three research questions:

- **RQ1.** How efficient are the three steps of ToDiff?
- **RQ2.** How effective is ToDiff in detecting bugs, compared to state of the art?
- **RQ3.** What are the root causes of the discovered bugs?

Subjects. In the experiments, we use ToDiff to test multiple implementations of OSPF and IS-IS from Frrouting [12]. We choose OSPF and IS-IS because they are the two most popular IGPs. We use implementations from Frrouting as it is an open-source routing protocol suite designed for Linux and Unix platforms and has been integrated into the software repositories of major Linux-based operating systems like Debian and CentOS. Frrouting is widely utilized in critical networking scenarios, including ISPs and SaaS infrastructure, and is trusted by technology giants such as NVIDIA and VMware. Frrouting has experienced robust growth in recent years, gradually becoming the standard implementation for popular routing protocols. To date, it has garnered 3.5K stars and 1.5K forks on GitHub, with a continuous stream of contributions from hundreds of developers worldwide.

Particularly, the details of the OSPF and IS-IS implementations are listed in Table 3. We choose the three most recent major versions of the FRRouting project for testing. The OSPF and IS-IS implementations consist of hundreds of thousands of lines of code, with OSPF ranging from 121KLoC to 145KLoC and IS-IS from 108KLoC to 135KLoC. The project has a relatively complex structure, comprising hundreds of source code files, with OSPF involving 157 to 181 files and IS-IS from 158 to 183 files. The implementations support a rich set of commands to write topological programs and provide precise control over the protocols, with 147 to 167 kinds of commands for OSPF and 74 to 100 commands for IS-IS, respectively.

It is worth noting that although our evaluation is conducted on the open-source IGP implementation, FRRouting [12], the results

Table 3: Details of OSPF and IS-IS Implementations.

IGP	Implementations	Size (KLoC)	# Files	# Commands
OSPF	frrouting-10.2	145	181	167
	frrouting-9.0	135	175	161
	frrouting-8.0	121	157	147
IS-IS	frrouting-10.2	135	183	100
	frrouting-9.0	126	177	100
	frrouting-8.0	108	158	74

are expected to generalize well to other routing protocol implementations. This is because, while we do not evaluate ToDiff on commercial implementations (e.g., Cisco, which are closed-source), these systems typically share similar architectural designs with FRRouting. For example, they use similar commands to configure and drive routers [7]. This architectural consistency ensures that ToDiff can be effectively applied to both open-source and commercial implementations, and seamlessly extended to support these implementations with minimal adaptation effort.

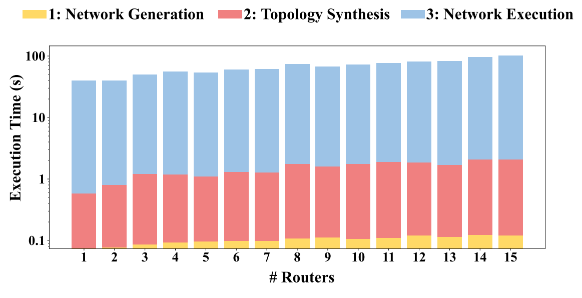
Baselines. To show the effectiveness of ToDiff, we compare ToDiff against existing validation techniques the FRRouting community is using. First, the FRRouting community actively handcrafts a rich test suite [10]. Currently, the test suite contains around 460 manually crafted test cases covering a wide range of network topologies. Second, we compare ToDiff to OSS-Fuzz [19], which is a popular fuzzing framework developed by Google, having successfully detected 36,000 bugs across 1,000 projects, including the OSPF and IS-IS implementations in FRRouting.

Environment. All experiments were conducted on a server with the following configuration: 32 cores, 64 threads, 3.4 GHz CPU, and 256 GB of memory, running Ubuntu 20.04 and Mininet 2.3.0.

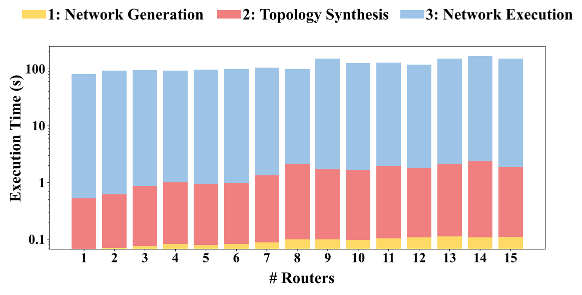
Rather than being a limitation, the use of a network simulated by Mininet brings two significant advantages. On the one hand, Mininet offers strong applicability as it creates realistic virtual networks by running real Linux kernel code, making it a widely adopted tool in both academia and industry for testing protocol implementations [8, 27]. As such, any approach that works in Mininet is also applicable to industry-grade routers and real-world deployment scenarios. Our method is therefore transferable to real environments without requiring fundamental modifications. On the other hand, the simulated network provides excellent scalability. Mininet supports the creation of large-scale virtual networks, which are often infeasible to construct or access in physical environments for testing purposes. This capability enables our method to be evaluated across a diverse range of topologies, demonstrating its scalability in testing complex protocol behaviors.

5.1 RQ1: Efficiency of ToDiff

To answer RQ1, we measure the execution time of each step in our method and conduct a detailed analysis. The experiment is conducted under different scenarios where 1 to 15 routers are generated in a network. A network of 15 routers is sufficiently large to trigger deeply hidden bugs through our approach. We record the time taken by ToDiff for its three main steps: network generation, topology synthesis, and differential IGP Validation. The experiment is repeated ten times in different implementations, and the average time cost and number of generated commands are reported.



(a)



(b)

Figure 11: The time cost of the three steps and the number of generated commands for (a) OSPF and (b) IS-IS.

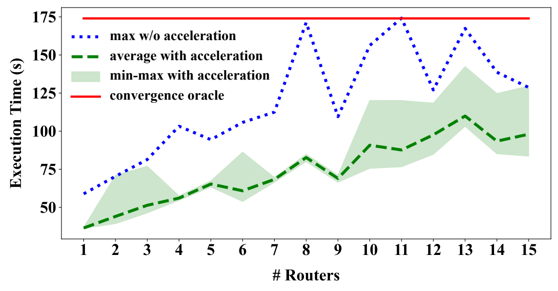


Figure 12: The time cost of Step 3 before and after applying our approach.

As plotted in Figure 11, (a) and (b) show the experiment results for OSPF and IS-IS, respectively. The X-axis denotes the number of routers in a generated network. The Y-axis, which is on a log scale, together with the bar chart, illustrates the time consumption of the three steps. The results demonstrate that our tool is highly efficient, with the total time cost of all three steps remaining under 3 minutes for both OSPF and IS-IS, with up to 15 routers. The testing time does not increase significantly with the network size, demonstrating the good scalability of our approach. For both OSPF and IS-IS, we complete Step 1, i.e., network generation, in less than one second, and Step 2, i.e., the synthesis of equivalent topological programs, within 2 seconds. The network execution was completed within 100 seconds for OSPF and 170 seconds for IS-IS. The total testing time for IS-IS is longer than that of OSPF, because IS-IS inherently has a slower convergence speed than OSPF.

Table 4: Bugs Discovered by ToDiff.

ID	IGP	Root Causes	ToDiff	FRRouting Test Suite	Google's OSS-Fuzz	Status
1	OSPF	Incorrect setting internal states about area structure	✓	✗	✗	PR merged
2	OSPF	Incorrect setting internal states about nssa status	✓	✗	✗	PR merged
3	OSPF	Incorrect setting internal states about area structure	✓	✗	✗	PR merged
4	OSPF	Incorrect setting internal hello timer	✓	✗	✗	PR merged
5	OSPF	Incorrect setting internal spf hold timer	✓	✗	✗	PR merged
6	OSPF	Incorrect setting wait timer	✓	✗	✗	Confirmed
7	OSPF	Incorrect setting dead multiplier timer	✓	✗	✗	PR merged
8	OSPF	Incorrect parsing the commands of no VLINK	✓	✗	✗	PR merged
9	OSPF	Incorrect parsing the commands of no AREANSSA	✓	✗	✗	PR merged
10	OSPF	Incorrect parsing the commands of no AREANSSARANGE	✓	✗	✗	PR merged
11	OSPF	Incorrect parsing the commands of no AREANSSACOST	✓	✗	✗	PR merged
12	OSPF	Incorrect parsing the commands of no AREADEFAULTCOST	✓	✗	✗	PR merged
13	OSPF	Incorrect parsing the commands of no OSPFWRITEMULTIPLIER	✓	✗	✗	PR merged
14	OSPF	Incorrect parsing the commands of no DEFUALTINFORMATION	✓	✗	✗	PR merged
15	OSPF	Incorrect parsing the commands of no DISTANCE	✓	✗	✗	PR merged
16	OSPF	Incorrect parsing the commands of no AREARANGECOST	✓	✗	✗	PR merged
17	OSPF	Incorrect parsing the commands of no AREASHORTCUT	✓	✗	✗	PR merged
18	OSPF	Incorrect parsing the commands of no DEADINTERVAL	✓	✗	✗	PR merged
19	OSPF	Incorrect parsing the commands of no ABRTYPE	✓	✗	✗	PR merged
20	OSPF	Incorrect parsing the commands of no WRITEMULTIPLIER	✓	✗	✗	PR merged
21	IS-IS	Incorrect setting interface's psnp-interval.	✓	✗	✗	PR merged
22	IS-IS	Incorrect setting interface's csnp-inveral.	✓	✗	✗	PR merged
23	IS-IS	Incorrect setting interface's priority.	✓	✗	✗	PR merged
24	IS-IS	Incorrect setting interface's hello-inveral	✓	✗	✗	PR merged
25	IS-IS	Incorrect setting interface's hello-multipllier	✓	✗	✗	PR approved
26	IS-IS	Incorrect setting interface's circuit-type	✓	✗	✗	PR approved

Since the time cost of Step 1 is negligible, we provide a detailed analysis for Step 2 and Step 3 below.

Detailed Analysis of Step 2. To show the scalability of our program synthesizer, we try to generate larger topological programs with up to 1,000 commands per router. For validation, the topological program comprising 1,000 commands is rich enough to cover a wide range of diverse network scenarios. The results show that this step takes less than 9 seconds to complete, attributable to our efficient program synthesis algorithm, which ensures good scalability for large-scale networks and complex topologies.

Detailed Analysis of Step 3. Step 3 is the most time-consuming step as we have to wait for a long time until the network converges before we can differentiate the results to find bugs. To reduce the waiting time, Section 4.3 presents a few heuristic methods for Step 3 to determine and speed up network convergence. In Figure 12, the dotted and blue line stands for the maximum time for a network to converge (the experiment uses our methods to determine network convergence but does not apply our method to accelerate convergence). We can observe that it takes up to about 175 seconds for a network to converge. It also means that if we do not provide any method discussed in Section 4.3 to determine and speed up network convergence, we have to always wait for at least 175 seconds for a network to converge, plotted by the solid and red line in the chart. With our strategies in Step 3, the average time waiting for a network to converge is significantly reduced by 1.6× to 4.7×, as plotted by the dashed and green line in Figure 12.

5.2 RQ2: Effectiveness of ToDiff

Code Coverage. We evaluate both the overall and topology-related code coverage for OSPF and IS-IS. The former refers to the code coverage across all source code files of OSPF and ISIS from frr/ospfd and frr/isisd directories of the Frrouting projects. Meanwhile, since our approach focuses on identifying topology-related bugs, we also evaluate the topology-related code coverage, which is computed by removing topology-irrelevant code.

Table 5: Overall / Topology-Related Code Coverage (%).

IGP	ToDiff	FRRouting Test Suite	Google's OSS-Fuzz
OSPF	62.3 / **85.1**	51.8 / **32.2**	15.7 / **1.1**
ISIS	68.9 / **78.2**	63.9 / **38.0**	8.9 / **5.3**

The coverage of ToDiff is collected during our experiments, which involve 200 randomly-generated valid networks and 100 equivalent topological programs for each network. We run each topological program only once. For the FRRouting test suite, we executed all test cases related to the core functionality of OSPF and IS-IS. In the case of OSS-Fuzz, the campaign was run continuously for 24 hours. To mitigate randomness, we repeated each experiment 10 times and reported the average results.

The code coverage results are summarized in Table 5. ToDiff consistently outperforms both the FRRouting test suite and OSS-Fuzz, achieving the highest overall and topology-related coverage. Unlike the FRRouting test suite, which depends on labor-intensive, manually crafted test cases, ToDiff automatically generates tests, enabling a more comprehensive and systematic exploration of network topologies. This automation results in a 2× increase in topology-related code coverage and is a key factor in ToDiff 's ability to uncover 26 bugs that the FRRouting test suite fails to detect. On the other side, OSS-Fuzz predominantly generates invalid packets during mutation, which are often limited to exercising exception-handling paths and rarely reach the core logic of the protocol, resulting in significantly lower coverage.

The code not covered by ToDiff is mainly related to user-defined protocol extensions (e.g., OSPF's opaque LSAs, which allow users to define custom LSA formats and contents). As ToDiff is specifically designed to verify standard-compliant protocol functionality, such optional or implementation-specific features are intentionally excluded from its testing scope.

Bug Detection Capability. To answer RQ2, we use ToDiff to detect bugs in OSPF and IS-IS implementations listed in Table 3. As discussed before, the implementations are from high-quality open-source projects and have been frequently checked by mature bug detection tools. Nonetheless, ToDiff can still detect 26 zero-day bugs listed in Table 4. Some of these bugs had even been hidden in the implementations for over 20 years, showing the high effectiveness of our approach. Meanwhile, many of the identified bugs are security-critical and exploitable. For example, the bug example discussed in Section 3 may redirect network traffic to the wrong routers, causing black holes, data leaks, and other severe consequences. As responsible researchers, we also upload patches to fix detected bugs. As shown in Table 3, "PR merged" means the patches have been merged into the code; "PR approved" means the patches have been approved by the developers and are pending merge; and "Confirmed" means developers confirm the validity of a bug report but are still working on fixing the bug. The links to these bug reports are hidden for double-blind review.

For bug detection, ToDiff achieves 100% precision and a 0% false positive rate. This is attributed to ToDiff 's design, which compares router attributes that must be consistent across equivalent topologies (see Section 4.3). Consequently, any observed discrepancy during differential testing signals the presence of a bug.

As shown in Table 4, the handcrafted test suite and OSS-Fuzz cannot detect any bugs we discover. On the one hand, the handcrafted test suite, albeit containing many test cases, cannot cover many cases that we can randomly generate. On the other hand, OSS-Fuzz fails to detect these bugs because it treats OSPF and IS-IS as common communication protocols and, thus, does not provide a special design to validate topology-related business logic in their implementations. These limitations highlight the advantages of our approach in effectively uncovering topology-related bugs.

As discussed in Section 1, although there are some automated approach to testing routing protocol implementations [25, 34, 35, 51], they are either too old and outdated or not publicly available, making them not directly comparable. As explained before, the key weaknesses of these existing methods are that they heavily depend on manual efforts to build testing oracles, whereas our approach applies differential testing to achieve fully automated validation.

5.3 RQ3: Root Cause of Discovered Bugs

We analyzed the root causes of all the bugs we found (listed in Table 4) and categorized them into two groups. We also provided case studies and discussed the potential impacts of these bugs, typically including network congestion and routing black holes.

Group 1: Incorrect Parsing of Commands (13/26). This type of error occurs when a protocol implementation incorrectly parses certain topological commands. These errors can cause the protocol implementation to reject valid or accept invalid commands, leading to misinterpretation of topological commands and, ultimately, incorrect network configurations. As illustrated below, such misconfigurations may lead to severe network problems like congestion.

```
1. DEFUN(ip_ospf_dead_interval, ip_ospf_dead_interval_cmd,
2.-     "ip ospf dead-interval minimal hello-multiplier (1-20)",
3.+     "ip ospf dead-interval minimal hello-multiplier (2-20)",
4.      ......
```

(a)

```
1. int ospf_vty_dead_interval_set(...){
2.     ......
3.+    re = seconds - event_timer_remain_second(oi->t_wait);
4.+    EVENT_OFF(oi->t_wait);
5.+    if (re > 0)  OSPF_ISM_TIMER_ON(oi->t_wait, ......);
6.     ......
```

(b)

Figure 13: Case studies of discovered bugs.

Figure 13(a) shows a bug discovered by our tool in the OSPF protocol, which has been hidden for over 20 years. The code defines the command: ip ospf dead interval minimal hello-multiplier <num>, which is used to set the hello and dead timers of an interface. The correct value of <num> should range from 2 to 20, but the buggy code allows 1 to 20. In consequence, if a network administrator uses the command to set the hello timer to 1 second (i.e., <num> = 1), the dead timer is automatically set to 1/<num> = 1 second, too. In this case, OSPF's hello timer and dead timer are set to the same value, causing OSPF to continuously restart the neighbor establishment process and send related packets. This behavior can result in network congestion along the affected path.

Group 2: Incorrect Action Logic for Routers (13/26). These errors occur when a protocol implementation contains wrong logic to set variables that control router states, timers, and so on. Such errors can lead to incorrect protocol behavior, such as executing unintended actions or sending malformed packets. These issues can result in severe network problems like routing black holes. The bug discussed in the motivating example belongs to this category.

Figure 13(b) shows the other bug we discovered in OSPF implementations. The code snippet in the figure handles the action logic for the command: ip ospf dead interval <num>. This command sets the wait timer, which is responsible for timing the transition from the waiting state to the connected state during the establishment of connection among routers. If a topological program uses a very large value for <num>, such as 1000, and then sets it back to a reasonably small value, the buggy code (without lines 3-5) fails to update the timer. As a result, the neighbor state remains stuck in the waiting state, thus preventing two routers from establishing neighbor relationships. This bug may result in an incomplete network topology and cause routing black holes or other potential network problems.

6 Related Work

Routing protocols differ from common (communication, management, or security) protocols as routing protocols concern network topologies. In contrast, the others focus on the communication among multiple parties and regard the network topology between two parties as black boxes. As discussed in Section 1, most existing works, e.g., network protocol fuzzing [1, 29, 32, 37, 42, 44, 45] or network differential analysis [15, 16, 56] are not designed for routing protocols and cannot detect errors happening at the time of topology establishment. In what follows, we discuss related works regarding the validation of routing protocols, but, in a word, to the best of our knowledge, ToDiff is the first work that utilizes program synthesis to enable differential testing for routing protocols.

Validating Routing Protocol Specifications. Unlike our work, which assumes the protocol specification is correct and validates if an implementation correctly follows the specification, many existing works do not validate the implementation of routing protocols but their specifications. They create abstract models as per the specification of a routing protocol and then perform model checking or verification to check if certain properties may be violated in the model [26, 33, 36, 50]. Wibling [50] leveraged linear temporal logic to verify the correctness of connection establishment and message broadcasting for the LUNAR protocol. Khayou and Sarakbi [26] used abstract algebraic to model the EIGRP protocol [43], verifying if its routing calculations converge and adhere to the shortest-path distance constraints. Maag et al. [33] modeled the MANET protocol [46] using an extended finite state machines [28], thereby verifying if the protocol correctly establishes network topology. Maxa et al. [36] designed a secure protocol, SUAP, and analyzed its security features via model checking. The techniques used in these works are fundamentally different from our approach and outside the scope of this paper because we aim to validate specific routing protocol implementations instead of the specifications.

Validating Routing Protocol Implementations. Similar to our work, there are many testing techniques checking if a routing protocol implementation follows the protocol's specification. Using handcrafted test suites [10, 11, 13] is the most straightforward way to test a protocol's implementation. However, these test suites face coverage limitations and cannot thoroughly test protocol correctness across diverse network topologies. Compared to handcrafted test suites, our work makes it possible to generate diverse topologies with automated testing oracles to automate the testing procedure.

Automated methods can generate diverse test cases and can be categorized as active or passive. Active methods [21–23, 25, 34, 35, 51] use models such as graphs or finite state machines to actively generate test packets, which are then sent to the protocols' implementations for evaluation. Hao et al. [21] propose a probabilistic algorithm that randomly inserts connections into a network and calculates the correct routing tables as an oracle. They then test implementations using the generated networks and compare the results with the oracle. This approach requires significant manual effort to compute the testing oracle — the correct routing table for each protocol. In contrast, our method applies differential testing, which not only provides automated testing oracles but also analyzes both intermediate router status and the final routing tables. Helmy et al. [22] and Kasemsuwan et al. [25] mutate a set of prepared network topologies to generate additional test cases. These methods generate relatively limited topologies and also suffer from the testing oracle problem. In contrast, our approach is not restricted by prepared networks and testing oracles. With a random network generation algorithm and differential testing, our approach can synthesize diverse topologies with automated testing oracle, enabling broader test coverage. Maag and Zaidi [34], Malik [35], Wu et al. [51] have to transform the implementation of routing protocols into an abstract model, which is often a complex and error-prone process. In contrast, such abstraction is not necessary for our approach, thereby reducing analysis complexity and enhancing applicability.

Passive methods [30, 31, 48, 54, 55] do not actively generate networks or routing protocol packets but record and analyze real-world network traffic to detect bugs in routing protocol implementations. These passive approaches differ from ToDiff, which is an active method, in the following aspects. First, we actively generate diverse networks and, thus, enable more comprehensive test coverage, whereas the passive approaches are restricted by observed network traffic. Second, we actively synthesize equivalent topological programs to address the testing oracle problem, while the passive approaches still rely on manual efforts to build testing oracles. This manual process is both labor-intensive and error-prone, restricting existing methods to a single protocol or a limited subset of protocol functionalities. Our approach addresses the oracle problem via differential analysis.

7 Conclusion

This work presents ToDiff, a differential testing technique to detect hidden bugs within IGP implementations. ToDiff first generates random yet valid networks, applies a semantics-guided and bounded program synthesizer to generate equivalent topological programs, and differentiates outcome topologies. To date, ToDiff has identified 26 bugs across common IGPs, all confirmed or fixed.

References

[1] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful greybox fuzzing. In *Proceedings of the 31st USENIX Security Symposium (SEC '22)*. USENIX Association, 3255–3272. https://www.usenix.org/conference/usenixsecurity22/presentation/ba

[2] Davide Benetti, Massimo Merro, and Luca Viganò. 2010. Model Checking Ad Hoc Network Routing Protocols: ARAN vs. endairA. In *2010 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM '10)*. IEEE, 191–202. https://doi.org/10.1109/SEFM.2010.24

[3] Sahana Bhosale and Ravindra Joshi. 2008. Conformance testing of OSPF protocol. In *2008 IET International Conference on Wireless, Mobile and Multimedia Networks (WoWMoM '08)*. IET, 42–47. https://ieeexplore.ieee.org/document/4470071

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*. USENIX Association, 209–224. https://dl.acm.org/doi/10.5555/1855741.1855756

[5] R. Callon. 1990. Use of OSI IS-IS for routing in TCP/IP and dual environments. https://datatracker.ietf.org/doc/html/rfc1195.

[6] Chia Yuan Cho, Vijay D'Silva, and Dawn Song. 2013. BLITZ: compositional bounded model checking for real-world programs. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE '13)*. IEEE, 136–146. https://doi.org/10.1109/ASE.2013.6693074

[7] Cisco. 2016. IP Routing: OSPF Configuration Guide. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_ospf/configuration/xe-16/iro-xe-16-book/iro-cfg.html.

[8] Rogério Leão Santos de Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. 2014. Using Mininet for emulation and prototyping Software-Defined Networks. In *2014 IEEE Colombian Conference on Communications and Computing (COLCOM '14)*. IEEE, 1–6. https://doi.org/10.1109/ColComCon.2014.6860404

[9] F. de Renesse and A.H. Aghvami. 2004. Formal verification of ad-hoc routing protocols using SPIN model checker. In *Proceedings of the 12th IEEE Mediterranean Electrotechnical Conference (MELECON '04)*. IEEE, 1177–1182. https://doi.org/10.1109/MELCON.2004.1348275

[10] FRR Developers. 2024. FRR RFC/Compliance Test. https://github.com/FRRouting/frr/wiki/RFC_Compliance_Results/OSPF_extended_results.pdf.

[11] FRR Developers. 2024. FRR Topotests. https://github.com/FRRouting/frr/tree/master/tests/topotests.

[12] FRR Developers. 2024. FRRouting Project. https://frrouting.org/.

[13] Munet Developers. 2024. Munet. https://github.com/LabNConsulting/munet.

[14] Wesley Eddy. 2022. Transmission Control Protocol (TCP). https://datatracker.ietf.org/doc/html/rfc9293.

[15] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. 2021. Prognosis: closed-box analysis of network protocol implementations. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. ACM, 762–774. https://doi.org/10.1145/3452296.3472938

[16] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining model learning and model checking to analyze TCP implementations. In *Computer Aided Verification: 28th International Conference (CAV '16)*. 454–471. https://doi.org/10.1007/978-3-319-41540-6_25

[17] GNU. 2025. GDB: The GNU Project Debugger. https://www.sourceware.org/gdb/.

[18] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Queue* 10, 1 (2012), 20–27. https://doi.org/10.1145/2090147.2094081

[19] Google. 2025. OSS-Fuzz. https://google.github.io/oss-fuzz/.

[20] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Conference on Security (SEC'13)*. USENIX Association, 49–64. https://dl.acm.org/doi/10.5555/2534766.2534772

[21] Ruibing Hao, David Lee, Rakesh K. Sinha, and Dario Vlah. 2000. Testing IP routing protocols — from probabilistic algorithms to a software tool. In *Formal Methods for Distributed System Development Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE/PSTV '00)*. Springer US, 249–264. https://doi.org/10.1007/978-0-387-35533-7_16

[22] A. Helmy and D. Estrin. 1998. Simulation-based 'STRESS' testing case study: a multicast routing protocol. In *Proceedings. Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '98)*. IEEE, 36–43. https://doi.org/10.1109/MASCOT.1998.693672

[23] Ahmed Helmy, Deborah Estrin, and Sandeep K. S. Gupta. 1998. Fault-oriented test generation for multicast routing protocol design. In *Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE XI / PSTV XVIII '98)*. Kluwer, B.V., 93–109. https://doi.org/10.1007/978-0-387-35394-4_6

[24] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP '20)*. IEEE, 1613–1627. https://doi.org/10.1109/SP40000.2020.00063

[25] Poonyavee Kasemsuwan and Vasaka Visoottiviseth. 2017. OSV: OSPF vulnerability checking tool. In *2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE '17)*. IEEE, 1–6. https://doi.org/10.1109/JCSSE.2017.8025919

[26] Hussein Khayou and Bakr Sarakbi. 2017. A validation model for non-lexical routing protocols. *J. Netw. Comput. Appl.* 98, C (2017), 58–64. https://doi.org/10.1016/j.jnca.2017.09.006

[27] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets '24)*. ACM, 1–6. https://doi.org/10.1145/1868447.1868466

[28] D. Lee and M. Yannakakis. 1996. Principles and methods of testing finite state machines - a survey. *Proc. IEEE* 84, 8 (1996), 1090–1123. https://doi.org/10.1109/5.533956

[29] Junqiang Li, Senyi Li, Gang Sun, Ting Chen, and Hongfang Yu. 2022. SNPSFuzzer: A fast greybox fuzzer for stateful network protocols using snapshots. *IEEE Transactions on Information Forensics and Security* 17 (2022), 2673–2687. https://doi.org/10.1109/TIFS.2022.3192991

[30] J. Liu, Yougu Yuan, D.M. Nicol, R.S. Gray, C.C. Newport, D. Kotz, and L.F. Perrone. 2004. Simulation validation using direct execution of wireless ad-hoc routing protocols. In *18th Workshop on Parallel and Distributed Simulation (PADS '04)*. IEEE, 7–16. https://doi.org/10.1109/PADS.2004.1301280

[31] Jason Liu, Yougu Yuan, David M Nicol, Robert S Gray, Calvin C Newport, David Kotz, and Luiz Felipe Perrone. 2005. Empirical validation of wireless models in simulations of ad hoc routing protocols. *Simulation* 81, 4 (2005), 307–323. https://doi.org/10.1177/0037549705055017

[32] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. 2023. Bleem: packet sequence oriented fuzzing for protocol implementations. In *32nd USENIX Security Symposium (SEC '23)*. USENIX Association, 4481–4498. https://dl.acm.org/doi/10.5555/3620237.3620488

[33] Stephane Maag, Cyril Grepet, and Ana Cavalli. 2008. A formal validation methodology for MANET routing protocols based on nodes' self similarity. *Comput. Commun.* 31, 4 (2008), 827–841. https://doi.org/10.1016/j.comcom.2007.10.031

[34] Stéphane Maag and Fatiha Zaidi. 2006. Testing methodology for an ad hoc routing protocol. In *Proceedings of the ACM International Workshop on Performance Monitoring, Measurement, and Evaluation of Heterogeneous Wireless and Wired Networks (PM2HW2N '06)*. ACM, 48–55. https://doi.org/10.1145/1163653.1163663

[35] Saif Ur Rehman Malik. 2014. *Using formal methods to validate the usage, protocols, and feasibility in large scale computing systems.* Ph. D. Dissertation. North Dakota State University. https://core.ac.uk/download/pdf/211302484.pdf#page=123

[36] Jean-Aimé Maxa, Mohamed Slim Ben Mahmoud, and Nicolas Larrieu. 2016. Extended verification of secure UAANET routing protocol. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC '16)*. IEEE, 1–16. https://doi.org/10.1109/DASC.2016.7777970

[37] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS '24)*. IEEE. https://doi.org/10.14722/ndss.2024.24556

[38] John Moy. 1998. OSPF Version 2. https://datatracker.ietf.org/doc/html/rfc2328.

[39] Madanlal Musuvathi and Dawson R. Engler. 2004. Model checking large network protocol implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1 (NSDI'04)*. USENIX Association, 12. https://dl.acm.org/doi/10.5555/1251175.1251187

[40] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. 2003. CMC: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.* 36, SI (2003), 75–88. https://doi.org/10.1145/844128.844136

[41] Ameet Naik. 2018. Anatomy of a BGP Hijack on Amazon's Route 53 DNS Service. https://www.thousandeyes.com/blog/amazon-route-53-dns-and-bgp-hijack.

[42] Roberto Natella. 2022. Stateafl: greybox fuzzing for stateful network servers. *Empirical Software Engineering* 27, 7 (2022), 191. https://doi.org/10.1007/s10664-022-10233-3

[43] Ivan Pepelnjak. 1999. *EIGRP network design solutions.* Cisco press. https://dl.acm.org/doi/10.5555/519477

[44] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST '20)*. IEEE, 460–465. https://doi.org/10.1109/ICST46399.2020.00062

[45] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. 2023. Nsfuzz: towards efficient and state-aware network service fuzzing. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–26. https://doi.org/10.1145/3580598

[46] E.M. Royer and Chai-Keong Toh. 1999. A review of current routing protocols for ad hoc mobile wireless networks. *IEEE Personal Communications* 6, 2 (1999), 46–55. https://doi.org/10.1109/98.760423

[47] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, 693–706. https://doi.org/10.1145/3192366.3192418

[48] Hasan Ural, Zhi Xu, and Fan Zhang. 2007. An improved approach to passive testing of FSM-based systems. In *Second International Workshop on Automation of Software Test (AST '07)*. IEEE, 6–6. https://doi.org/10.1109/AST.2007.1

[49] Mark Weiser. 1984. Program slicing. *IEEE Transactions on Software Engineering* 10, 4 (1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

[50] Oskar Wibling. 2005. *Ad hoc routing protocol validation.* Ph. D. Dissertation. Uppsala University. https://uu.diva-portal.org/smash/get/diva2:117170/FULLTEXT01.pdf

[51] Jianping Wu, Zhongjie Li, and Xia Yin. 2003. Towards modeling and testing of IP routing protocols. In *Proceedings of the 15th IFIP International Conference on Testing of Communicating Systems (TestCom'03)*. Springer-Verlag, 49–62. https://dl.acm.org/doi/10.5555/1764575.1764582

[52] Yichen Xie and Alex Aiken. 2005. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*. ACM, 351–363. https://doi.org/10.1145/1040305.1040334

[53] Carter Yagemann, Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee. 2021. Automated bug hunting with data-driven symbolic root cause analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. ACM, 320–336. https://doi.org/10.1145/3460120.3485363

[54] Yixin Zhao, Jianping Ju, and Yin Xia. 2002. From active to passive—progress in testing Internet routing protocols. *Journal of Computer Science and Technology* 17, 3 (2002), 264–283. https://doi.org/10.1007/0-306-47003-9_7

[55] Yixin Zhao, Xia Yin, and Jianping Wu. 2001. Online Test System, an application of passive testing in routing protocols test. In *Proceedings of 9th International Conference on Networks (ICON'01)*. IEEE, 190–195. https://doi.org/10.1109/ICON.2001.962339

[56] Mingwei Zheng, Qingkai Shi, Xuwei Liu, Xiangzhe Xu, Le Yu, Congyu Liu, Guannan Wei, and Xiangyu Zhang. 2024. ParDiff: practical static differential analysis of network protocol parsers. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 1208 – 1234. https://doi.org/10.1145/3649854