

A General Approach to Generate Test Packets With Network Configurations

Yahui Li , Han Zhang , *Member, IEEE*, Jilong Wang, *Senior Member, IEEE*, Zhiliang Wang , *Senior Member, IEEE*, Xia Yin, Xingang Shi , *Member, IEEE*, and Jianping Wu, *Fellow, IEEE*

Abstract—The correctness and reliability of modern networks are often the greatest concerns. A myriad network events like software update, device crash and resource exhaustion, inevitably lead to liveness errors on data plane. This paper focuses on fault detection of the network data plane using test packets. Existing test packet generation techniques are limited in two aspects: i) it is difficult to collect the input data plane snapshot through SNMP or terminals ii) it may rise false negatives due to inconsistent snapshot. In this paper, we propose a new framework, SWIFT, that automatically generates test packets with network configurations. SWIFT minimizes the number of test packets by allowing a packet to go through multiple links or interfaces. For network updates, SWIFT updates test packets in an incremental way to revalidate the network. We evaluate its performance using hundreds of benchmark network configurations. The results show that it takes few seconds to generate test packets to exercise all links and interfaces, and updates the test packets in few seconds for configuration changes. We also deployed a SWIFT prototype in a university network, and successfully detected many network outages.

Index Terms—Network Configurations, Abstract Representation, Test Packets, Network Reliability, Network Management.

I. INTRODUCTION

MODERN networks are growing increasingly complicated. As shown in Fig. 1, users implement high-level policies (A) by configuring device-specific configuration files (B). Then, devices generate the data plane (C), forwarding information bases (FIBs), using distributed routing protocols, which logically determines the forwarding behavior of each packet. Finally, the actual implementation of data plane, including the liveness of interfaces, links and devices, determines the actual

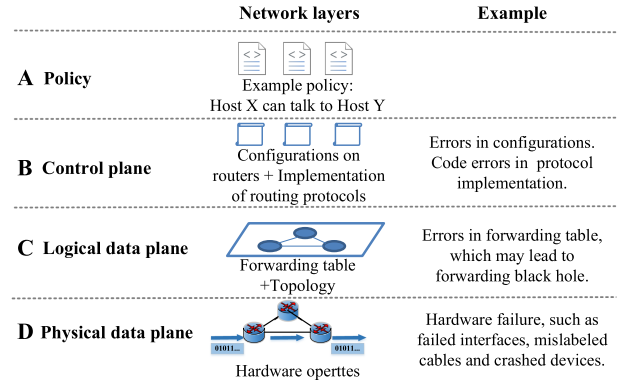


Fig. 1. Static checking versus dynamic checking. Static checking (e.g. [1] or [2]) confirms $A = B$ or $A = C$. Dynamic checking (e.g. SWIFT) confirms liveness properties of the topology and whether $B = D$.

TABLE I
RANKING OF CAUSES REPORTED BY NETWORK MANAGERS

Category	% of ≥ 4
Hardware failure	41.18%
Software bug on router/switch	39.71%
ACL or protocol misconfiguration	29.41%
QoS/TE misconfiguration	22.06%
Software upgrade on router/switch	19.12%
External	33.82%
Attack	23.53%
Unknown	16.18%

forwarding behavior of each packet (D). To ensure the actual data plane behaves as the policy defined by users, all above layers should remain consistent at all times. The network should also satisfy a large set of properties, in particular liveness properties, which require sufficient links and devices are working. However, a mass of network events, such as interfaces go down, device update and device crash, can make the forwarding behavior of actual data plane violate high-level policies.

According a survey conducted by North American Network Operators' Group (NANOG), software bugs and hardware failures are the main causes of packets forwarding failures [3]. Inspired by this survey, we further invite more network engineers, including the managers of the largest global service providers, to summarize the causes of the network failures. Table I shows the ranking results (5 = most often, 1 = least often) of causes reported by network managers including the 61 managers from NANOG survey and 7 managers from our survey. 41.18% of the

Manuscript received 14 September 2022; revised 26 December 2022; accepted 19 January 2023. Date of publication 2 February 2023; date of current version 7 March 2023. This work was supported in part by the National Science Foundation under Grant 62102020, in part by the National Key Research and Development Program under Grant 2020YFE0200500 and in part by the National Science Foundation under Grants 62002009. Recommended for acceptance by Y. Yang. (Corresponding author: Han Zhang.)

Yahui Li is with the Institute for Network Sciences, Cyberspace, Tsinghua University, Beijing 100190, China (e-mail: liyahui_ego@126.com).

Han Zhang, Jilong Wang, Zhiliang Wang, and Xingang Shi are with the Institute for Network Sciences, Cyberspace, Tsinghua University, Beijing 100190, China, and with the Zhongguancun Laboratory, Beijing 100081, China, and also with the Tsinghua National Laboratory for Information Science, and Technology (TNLIST), Beijing 100084, China (e-mail: zhhan@tsinghua.edu.cn; wjl@tsinghua.edu.cn; wzl@cernet.edu.cn; shixg@cernet.edu.cn).

Xia Yin and Jianping Wu are with the Department of Computer Science, Technology, Tsinghua University, Beijing 100190, China, and with the Zhongguancun Laboratory, Beijing 100081, China, and also with the Tsinghua National Laboratory for Information Science, and Technology (TNLIST), Beijing 100084, China (e-mail: yinxia00@tsinghua.edu.cn; jianping@cernet.edu.cn).

Digital Object Identifier 10.1109/TPDS.2023.3241433

managers report that the ranking of hardware failure ≥ 4 and 39.71% of the managers report that the ranking of software bug on router/switch ≥ 4 . In many cases, software bugs and hardware failures can lead to reachability failure, congestion, forwarding loop, performance faults or security vulnerabilities [4], which could cause a great loss of property. Therefore, it is necessary to proactively check the liveness properties of the actual data plane.

Towards reliable network, recently, researches have proposed many static checking or dynamic checking tools. For example, Batfish [5], era [6], Plankton [7], Minesweeper [1] and SRE [8] verify whether the configuration (B) satisfies the network policies (A). They are exclusively designed to detect configuration errors in the control plane. Other data plane static checking tools, like NOD [2], Delta-net [9], and Veriflow [10], verify whether the forwarding information bases (C) violates network reachability properties (e.g., no forwarding loop, no black hole). These tools can help to detect the errors in the forwarding information bases. However, the problems in actual implementation of data plane (D), such as failed interfaces, mislabeled cables and crashed devices, cannot be detected by these approaches.

It is extremely challenging for network managers to manually confirm that each interface in the data plane executes correctly. For example, a router with a faulty line card starts dropping packets silently. It will not be noticed until some users complain about connectivity. Then, managers use primordial tools (e.g., ping, traceroute) to check devices based on their knowledge. For each run of these tools, network managers can only analyze whether and how a single given packet reaches its destination. These tools suffer poor data plane coverage, which are far from efficient to ensure the correctness of all links and interfaces. Some recent works, like ATPG [3] and Pronto [11] propose to generate test packets to exercise every link and interface¹ by reading FIBs (C). However, they suffer two critical limitations. (i) They assume that users collect data plane snapshots through SNMP, or terminals maintained by devices. Until now, there has been *no effective way* to collect snapshots for the networks with an amount of distributed devices [12]. (ii) They assume that the snapshot of data plane is consistent. As devices provide a snapshot of their forwarding information bases at slightly different times [13], collecting stable and consistent data plane snapshots is very challenging, and *inconsistent snapshots* could lead to false-positive and false-negative test results.

Fortunately, we observe that control plane uses network configurations, topology and current environments (i.e., route advertisements) to generate data plane. In addition, network configurations on the routers are usually stable and easy to obtain. Considering the above factors, we propose to detect the liveness errors with network configurations. To achieve this goal, we face several aspects of challenges. First, the relationship between *configurations* and *forwarding behavior* are quite complex. There are various protocols and routing mechanisms (e.g., iBGP, eBGP, OSPF, route redistribution, VXLANs, VLANs, ACLs, static routing) which further interact with each other in many aspects, and configurations on different routers jointly

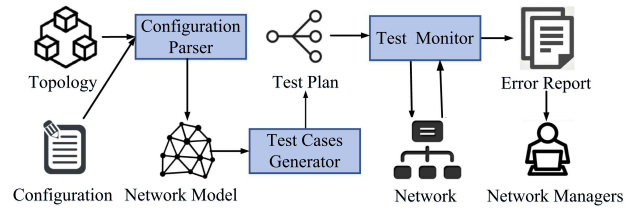


Fig. 2. SWIFT framework.

affect packet forwarding in a distributed way. Reasoning about the packet forwarding with configurations is quite challenging. Second, every interface on data plane should be exercised by at least one test packet (*completeness*), and the number of test packets should be limited such as the regular traffic is not effected (*bandwidth efficiency*), and it is always desirable for the generation time to be reasonably low (*fast generation*). Fulfilling the three goals at the same time is rather difficult. Third, in many situations, such as incident mitigation, new application requirements, security concern or business service concern, network managers perform *frequent network configuration updates* [1]. Test packet generation tool should update the probes in a short time to respond the frequent network changes. Reasoning about the relationship between the *updated configuration constructs* and *impacted probes* is quite challenging.

In this article, we propose a novel framework, SWIFT, to automatically and systematically test the packet forwarding on the actual data plane. To address the first challenge, we propose a *formal model* to succinctly capture the key behaviors of various routing protocols and routing mechanisms. To address the second challenge, we propose a novel probe generation algorithm which *minimizes* the number of test packets while *retaining the coverage* of all interfaces. With the network model, it constructs the reachability-to-ports table, and each row in the table provides an ordered list of interfaces the packet matched as it traversed the network. With the table, it then computes a minimum subset of the test packets which could cover all interfaces. To address the third challenge, we propose a novel probe update algorithm which *incrementally updates* the test packets in a short time. The key idea is to identify the updated entries in the reachability-to-ports table due to changed configurations, delete the invalid probes from the removed entries and generate minimum test packets to cover new entries.

Fig. 2 illustrates a high level image of SWIFT. First, it collects the network configurations from devices, and parses these configurations. Then, it generates test packets to exercise every link and interface based on the formal model. It then requests test terminals to periodically send test packets. If the destination receives the probes, it informs the test monitor that the historical interfaces work correctly.

In sum, the contributions in this paper are as follows:

- To the best of our knowledge, SWIFT is the first testing tool for liveness errors that generates test packets with network configurations.
- SWIFT provides a novel test packet generation approach which generates the minimum test packets to exercise all

¹In this article, we use the terms ‘ports’ and ‘interface’ interchangeably.

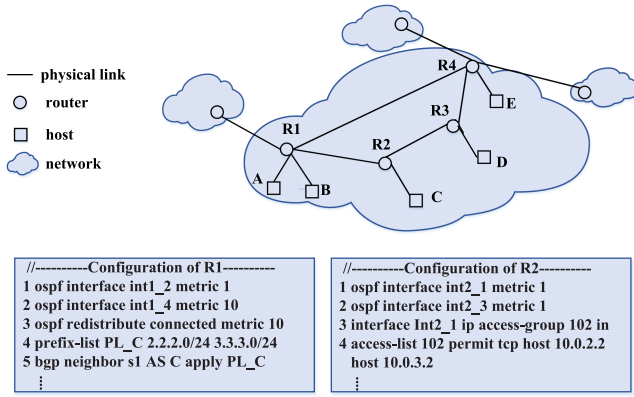


Fig. 3. A configuration snippet of the network. Routers $R1$ to $R4$ participate in an OSPF instance, where the cost is all assigned with weight 1 except the weight (10) between $R1$ and $R4$. Routers connect to five subnets, A to E . On router $R2$, an ACL blocking traffic from B destined for E is configured.

links and interfaces, and a test packet update approach that incrementally updates probes to revalidate the network.

- The accuracy and efficiency of SWIFT are evaluated with benchmark network configurations. The results show that SWIFT takes few seconds to generate or update test packets to exercise all links and interfaces.

The remainder of this paper is organized as follows. We present the network model in Section II. Then, we introduce the detailed methods of test packet generation and test packet update in Section III and Section IV. In Section V, we introduce the evaluation of our methodology. Finally, we discuss the related work in Section VI, and present concluding remarks in Section VII.

II. NETWORK MODEL

Generating test packets with network configurations requires knowing the network forwarding state. In this section, we introduce the network model, which is used to compute network forwarding state from the low-level network configurations.

A. Network Configurations

Routers in networks run a set of distributed routing protocols. These routing protocols exchange route advertisements with neighboring routers and compute path with different algorithms. Configurations on a router written by network managers or automatic configuration generator [14], [15] in a vendor-specific configuration language (e.g., Cisco IOS [16], Juniper IOS [17]) define which protocol it can use, what link cost should be set, which neighbor it sends route advertisements to, how it selects the best router multiple possible paths and so on. With these configurations, each router performs computations and produces forwarding rules which are used to forwarding packets along the appropriate path.

Fig. 3 provides the simplified configurations from router $R1$ and router $R2$, loosely written in Cisco's IOS language. In $R1$'s configuration, the first line specifies that it runs OSPF on interface that connects it to $R2$ with routing cost metric of

1, and the second line specifies that it runs OSPF on interface that connects it to $R4$ with routing cost metric of 10. Line 3 in configuration of $R1$ defines that routes from connected routes are redistributed into OSPF. Lines 4–5 in configuration of $R1$ refer to that it runs BGP with its neighbor AS C , and it will only accept only announcements for prefixes (i.e., 2.2.2.0/24, 3.3.3.0/24) defined in prefix list PL_C . Router $R2$ is similarly configured except that it has an access control list (ACL). Line 4 in configuration of $R2$ refers to that interface $int2_1$ denies the traffic which matches the access list 102.

B. Abstract Representation for Networks

With formal model based testing, users can determine which links and interfaces can be exercised by a given test packet. However, modeling network with configurations is challenging, as the forwarding behavior of packets is defined by multiple routing protocols, sophisticated features and their intersection. For constructing network model, one possible approach is to simulate all possible outcomes of individual routing protocol and their intersections. Unfortunately, it often leads to a lot of unnecessary work. For test packets generation task, we aim to reason what happens to packet forwarding with the given configurations. We observe that, for this task, it often only requires final forwarding paths of the packets regardless the details of how the forwarding path is computed in devices. Therefore, we propose *extended traffic propagation graph* to concisely capture the configurations' and protocols' impact on the network's forwarding behavior, which avoids modeling network with the actual protocol message exchanges of routing protocols. Our model extends the abstraction in ARC [18], which cannot account for commonly-used network constructs, including local preference of BGP, communities of BGP, and iBGP.

Each extended traffic propagation graph (ETPG) models the behavior of the network's routing protocols and their interactions among the specific traffic class. A *traffic class* refers to the traffic flowing from one endpoint group to another group (e.g., hosts, subnets). An ETPG shows how traffic from a subnet is propagated to another subnet.

We introduce the following definitions to facilitate subsequent presentation of ETPG. *Routing process* is the instance of a protocol, where router exchanges route advertisements with neighboring routers. *Routing adjacency* refers to a pair of routing processes on neighboring devices. *Routing instance* is a collection of processes running on different devices that exchange information using a specific protocol. If configuration defines route redistribution, the router exchanges the path information between these protocol processes.

Vertices. Each *routing process*, including static route, is abstracted as an incoming vertex (I) and an outgoing (O) vertex. These abstractions are suitable to express route redistribution and route selection between routing processes. The source endpoint and destination endpoint are abstracted as a special source vertex (SRC) and a special destination vertex (DST), respectively.

Edges. Edges between vertices are computed to model the traffic flow resulting from *intra-device* communication and

inter-device communication between *routing processes*. These edges show the possible traffic flow enabled by the information exchange between the routing processes. Each edge is annotated with a weight value and an attribute vector. The weight of the edge is used to model the route selection, and the attribute vector is used to model the configured filters to reflect the actual path of the traffic class taken in the network. The flow of routing information from source vertex to destination vertex results the flow of traffic from destination vertex to source vertex.

Inter-device edges are between vertices associated with the same routing process on different devices. The inter-device edge always goes from an out vertex to an in vertex. An inter-device edge from an out vertex of process p on a router R to an in vertex of process $\bar{p}$ on another router $\bar{R}$ shows that (i) there is a physical link or a sequence of links between router R and router $\bar{R}$, (ii) router R and router $\bar{R}$ participate in the same routing instance and exchange routing information, (iii) neither process p or process $\bar{p}$ filters the route advertisements about this traffic class and (iv) process p learns a route for this traffic class from process $\bar{p}$ by exchanging of the routing information. Note that, static routes are also viewed as a routing process. Different from dynamic routes computed by the processes of routing protocols, static routes are not obtained based on the exchanging advertisements between routing processes. Therefore, the out vertex of static route is connected to in vertices for all routing processes on the device whose interface's IP address matches the next hop IP in the static route.

Intra-device communication consists of the route redistribution between routing protocols and route selection between routing protocols on the same device. An intra-device edge always goes from a routing process's in vertex to its out vertex, or from the redistributtee process's in vertex to the redistributor process's out vertex. When multiple routing processes learn a route to a destination, the route with the lowest administrative distance (AD) will be selected and used on this device. The intra-device edge goes from the lower-AD process' out vertex to the higher-AD process' in vertex. The forwarding behavior of a particular packet using iBGP depends on the forwarding behavior (e.g., OSPF-provided routes, connected routes) of each user-defined iBGP-configured next-hop destination IP address between iBGP peers. If the IP address of the iBGP process' peer falls within the subnet assigned to a VLAN on the same device, the out vertex of the iBGP process is connected to the in vertex of the VLAN process. If no such VLAN process exists, the out vertex of the iBGP process is connected to the in vertex of the OSPF process associated with another ETPG graph corresponding to the traffic class with the iBGP-configured next-hop destination IP.

Filter mechanisms. Routing filters in configurations prevent a routing process from advertising particular prefixes to another process, and ACLs block or permit particular traffic classes. To model these mechanisms, we prune the corresponding inter-device edges or intra-device edges. For instance, if a routing filter of BGP process P may filter routes imported from a neighbor $\bar{P}$ based on the advertised prefix, it removes from the vertex associated with P , the outgoing edge that corresponds to $\bar{P}$.

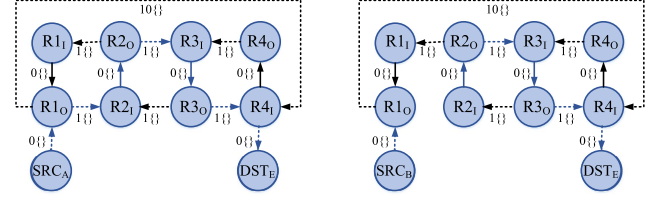


Fig. 4. ETPGs for traffic classes $A \rightsquigarrow E$ and $B \rightsquigarrow E$. Circles represent process vertices. Dark lines are intra-device edges, and light lines are inter-device edges. The blue lines are the min-cost paths through the ETPGs.

Edge attributes. Routing protocols use a different set of metrics (e.g., link costs, AS-path length) to express path costs or preferences. To reflect the actual forwarding behavior for the traffic class, we annotate the weight for each edge with the weight assignment algorithm [18], such that the shortest path between the source and destination is the forwarding path in the real network. The tag attribute of each edge is propagated and updated based on the configurations (e.g., BGP community-based filter, ASN-based filter).

Fig. 4 shows two example ETPGs of traffic classes $A \rightsquigarrow E$ and $B \rightsquigarrow E$ for the network in Fig. 3. $R1_I$ and $R1_O$ in left figure represent OSPF process on router $R1$. $R2_I$, $R2_O$, $R3_O$, $R4_O$ etc, are similarly named. $R1_I \rightarrow R1_O$ models intra-device communication on router $R1$. $R1_I \rightarrow R2_O$ models routing adjacencies between a pair of routing processes on $R1$ and $R2$. The intra-device edges are assigned a weight of 0, and the weights of inter-device edges match the configured OSPF link costs. The attribute vector for each edge is set as empty, as there is no tag-related configurations on these routers. In the right figure, as the ACL on $R2$ blocking traffic from B to E , the corresponding edge $R1_I \rightarrow R2_O$ is removed from the digraph. Details on constructing ETPGs are summarized in Algorithm 1.

III. TEST PACKETS GENERATION

To achieve the test packets generation goal, we propose the following three-step approach: *i*) converting the network configurations into ETPGs, *ii*) constructing all-pair reachability-to-ports table with ETPGs, *iii*) selecting a minimum set of probes to cover all links with the reachability-to-ports table.

A. Converting Configurations Into ETPGs

As mentioned above, the network's forwarding behavior can be modeled using a collection of extended traffic propagation graphs. Algorithm 1 lists the key operations (e.g., determining the appropriate vertices, weighted edges, and appropriate attribute vector) to construct each ETPG.

First, it initializes two vertices and one intra-device edge for each routing process (lines 1–2). Then, it accounts for route distribution. If no route filter is configured to deny the advertisement of redistributor's process, it adds one intra-device edge to model route exchange from one protocol process to another in a device (lines 3–7). The weight of each edge is set following the methods in [18]. Then, it models the routing exchange (e.g., route advertisements by BGP) within a routing instance between two

Algorithm 1: constructETG ($tc_s^d, \mathcal{N}_\Omega$).

Input: $\mathcal{N}_\Omega$: information parsing from network configurations.
Output: etg_s^d : ETPG for traffic class tc_s^d .

```

1 foreach  $proc \in \text{getRoutingProcess}(\mathcal{N}_\Omega)$  do
2    $etg_s^d \leftarrow \text{initVertexEdge}(tc_s^d, proc)$ ;
   // Add vertex  $proc_i$ , vertex  $proc_o$ , and
   // intra-dev edge  $proc_i \rightarrow proc_o$ .
3   if  $\text{routeFilter}(proc, \llbracket \text{getDst}(tc_s^d) \rrbracket) = \text{false}$  then
4     continue;
5   foreach  $\overline{proc} \in \text{getProcess}(\llbracket \text{getDevice}(proc) \rrbracket)$  do
6     if  $\text{redistribute}(proc, \overline{proc}) = \text{true} \wedge \text{routeFilter}(\overline{proc},$ 
7        $\llbracket \text{getDst}(tc_s^d) \rrbracket) = \text{false}$  then
8          $etg_s^d \leftarrow \text{addEdge}(\overline{proc}_i \rightarrow proc_o, etg_s^d)$ ;
         // Add intra-dev edge  $\overline{proc}_i \rightarrow proc_o$ .
9   foreach  $inf \in \text{getInterface}(proc)$  do
10    foreach  $inf \in \text{getPhyLink}(inf)$  do
11      foreach  $\overline{proc} \in \text{getUsedProcess}(\overline{inf})$  do
12        if  $\text{routeFilter}(\overline{proc}, \llbracket \text{getDst}(tc_s^d) \rrbracket) = \text{false}$ 
13          then
14             $etg_s^d \leftarrow \text{addEdge}(proc_o \rightarrow \overline{proc}_i, etg_s^d)$ ;
            // Add inter-dev edge.
15            if  $\text{aclFilter}(\llbracket \text{getACL}(inf) \rrbracket, etg_s^d) = \text{true}$ 
16               $\vee \text{aclFilter}(\llbracket \text{getACL}(\overline{inf}) \rrbracket, etg_s^d) =$ 
17                 $\text{true}$  then
18               $etg_s^d \leftarrow \text{removeEdge}(proc_o \rightarrow \overline{proc}_i,$ 
19                 $etg_s^d)$ ;
20  foreach  $dev \in \text{getDevices}(\mathcal{N}_\Omega)$  do
21    if  $\text{existPhyLink}(\llbracket \text{getSrc}(tc_s^d) \rrbracket, dev) = \text{true}$  then
22      foreach  $proc \in \text{getProcess}(dev)$  do
23         $etg_s^d \leftarrow \text{addEdge}(src \rightarrow proc_o, etg_s^d)$ ;
        // Add source edge.
24    if  $\text{existPhyLink}(dev, \llbracket \text{getDst}(tc_s^d) \rrbracket) = \text{true}$  then
25      foreach  $proc \in \text{getProcess}(dev)$  do
26         $etg_s^d \leftarrow \text{addEdge}(proc_i \rightarrow dst, etg_s^d)$ ;
        // Add destination edge.

```

devices (lines 8-12). If ACLs deny the traffic to the destination of the traffic class, it removes the corresponding edges (lines 13-14). Finally, it adds edges from the SRC vertex to a routing process's out vertex, which runs a device directly reached by the source endpoints. Similarly, if there is a device on which a routing process runs can directly reach the destination endpoints, it adds edges from that process's in vertex to the DST vertex.

There are at most dm nodes and mr edges in each ETPG, where d is the number of devices in the network, m is the maximum number of routing processes running on a single device, and r is the maximum number of devices participating in a single routing instance. Therefore, we can bound the worst case running time of constructing an ETPG by $\mathcal{O}(d^2m^2 + mr)$; and the overall computation cost of constructing ETPGs from device configurations yields $\mathcal{O}(nd^2m^2 + nmr)$, where n is the number of traffic classes.

B. Computing the Reachability-to-Ports Table

With the ETPG for each traffic class, we then try to answer the following questions: *i*) which traffic can be reachable from source port to destination port? *ii*) what is the sequence of the

TABLE II
REACHABILITY-TO-PORTS TABLE FOR THE NETWORK IN FIG. 3

Entry	ETPG	Ingress	Egress	Port history	Header space
1	$ETPG_{A \rightsquigarrow B}$	p_{10}	p_{11}	p_{10}, p_{11}	$src_ip = A.ip$ $dst_ip = B.ip$
2	$ETPG_{A \rightsquigarrow C}$	p_{10}	p_{20}	$p_{10}, p_{12} \rightarrow p_{21},$ p_{20}	$src_ip = A.ip$ $dst_ip = C.ip$
3	$ETPG_{A \rightsquigarrow D}$	p_{20}	p_{40}	$p_{10}, p_{12} \rightarrow p_{21},$ $p_{22} \rightarrow p_{31}, p_{30}$	$src_ip = A.ip$ $dst_ip = D.ip$
4	$ETPG_{C \rightsquigarrow D}$	p_{20}	p_{30}	$p_{20}, p_{22} \rightarrow p_{31},$ p_{30}	$src_ip = C.ip$ $dst_ip = D.ip$
5	$ETPG_{D \rightsquigarrow E}$	p_{30}	p_{40}	$p_{30}, p_{32}, p_{41}, p_{40}$	$src_ip = D.ip$ $dst_ip = E.ip$

links history² exercised by these traffic? To achieve this goal, we propose to construct the reachability-to-ports table with these ETPGs. Table II shows an example reachability-to-ports table for the network in Fig. 3. Each entry in the table has a packet space (i.e., the 6th column), indicating that any packet in this space can travel from the ingress port (i.e., the 3th column) to the egress port (i.e., the 4th column). The 5th column in the table refers to the sequence of the links exercised by these packets.

Algorithm 2 summarizes the key steps to construct the reachability-to-ports table. First, it loads any prefix that is mentioned in the network configurations (e.g., prefixes referenced in ACLs, prefixes advertised by routing processes). Some prefixes may overlap, we use rule optimization algorithms [19] compute a set of nonoverlapping prefixes. Then, it computes all possible combinations of these prefixes, each of which is corresponding to a traffic class (lines 1-2). In this step, we do not take the route advertisements sent by the ‘outside world’ into consideration, as it can only use the terminal whose port is permitted to send or receive probes. It is meaningless to generate test packets which cannot be produced by internal network. As the min-cost path from source vertex to destination vertex in the ETPG exactly is the forwarding path taken by the traffic class in the actual network, it then computes the min-cost path using depth-first or breadth-first polynomial-time graph algorithms. As the path is composed of the abstract edges in the ETPG, it then maps these abstract edges to the links in the real network (lines 3-6). Following the above steps, all the information for a traffic class can be obtained, and a new entry can be added to reachability-to-ports table (line 7). By doing this for all traffic classes, it can generate a reachability-to-ports table for the network.

The computation of the min-cost path in each ETPG yields $\mathcal{O}(d^2m^2)$ complexity, where d is the number of devices, m is the maximum number of routing processes running on a single device. Therefore, we can bound the worst case computation cost of constructing reachability-to-ports table by $\mathcal{O}(nd^2m^2)$, where n is the number of traffic classes.

C. Packet Generation With the Reachability-to-Ports Table

For each entry in reachability-to-ports table, one packet in the packet space (i.e., the 6th column in Table II) can cover all links (i.e., the 5th column) associated with that entry. In order

²At any point, each packet in one traffic class has a link history, which is an order lists of links that packets traverse so far in the network.

Algorithm 2: generateProbes ($\mathcal{N}_\Omega$).

Input: $\mathcal{N}_\Omega$: information parsing from network configurations;
 q : the pre-specified parameter.
Output: ETG_Ω : the ETGs; T : reachability-to-ports table; P : probes.

```

1  $T \leftarrow \text{initializeTable}()$ ,  $P \leftarrow \text{initializeProbe}()$ ,  $S \leftarrow \text{initializeCoveredSet}()$ ;
2  $C_\Omega \leftarrow \text{getTrafficClass}(\mathcal{N}_\Omega)$ ;
3 foreach  $tc_s^d \in C_\Omega$  do
4    $etg_s^d \leftarrow \text{constructETG}(tc_s^d, \mathcal{N}_\Omega)$ ;
5    $eh_s^d \leftarrow \text{getHistory}(\llbracket \text{getForwardingPath}(etg_s^d) \rrbracket, \mathcal{N}_\Omega)$ ;
6    $ETG_\Omega \leftarrow \text{appendETGs}(etg_s^d)$ ;
7    $T \leftarrow \text{appendEntry}(\llbracket \text{getIngress}(etg_s^d) \rrbracket, \llbracket \text{getEgress}(etg_s^d) \rrbracket, eh_s^d, \llbracket \text{getHeader}(etg_s^d) \rrbracket)$ ;
8  $\mathcal{L}_\Omega \leftarrow \text{computePorts}(\mathcal{N}_\Omega)$ ;
9  $H_\Omega \leftarrow \text{getHistory}(\llbracket \text{getEntry}(T) \rrbracket)$ ;
10 foreach  $H_i \in H_\Omega$  do
11    $H^{(k)} \leftarrow \text{initialize}(H_i, q)$ ;
12   //  $q^k \leq |H_i| \leq q^{k+1}$ 
13 while  $k \neq 0$  do
14   foreach  $H_i \in H^{(k)}$  do
15     if  $\text{cover}(\mathcal{L}_\Omega, S) = \text{true}$  then
16        $\text{break}$ ;
17     if  $|H_i \setminus S| \geq q^k$  then
18        $S \leftarrow \text{updateCoveredSet}(S \cup H_i)$ ;
19        $P \leftarrow \text{appendProbe}(P, \llbracket \text{getHeader}(H_i) \rrbracket)$ ;
20     else
21        $H_i \leftarrow H_i \setminus S$ ;
22        $H^{(k)} \leftarrow \text{updateSubCollection}(H_i, q)$ ;
23       //  $q^k \leq |H_i| \leq q^{k+1}$ 
24    $k \leftarrow \text{update}(k)$ ;
25 foreach  $H_i \in H^{(0)}$  do
26   if  $\text{cover}(\mathcal{L}_\Omega, S) = \text{true}$  then
27      $\text{break}$ ;
28   if  $|H_i \setminus S| = 1$  then
29      $S \leftarrow \text{updateCoveredSet}(S \cup H_i)$ ;
30      $P \leftarrow \text{appendProbe}(P, \llbracket \text{getHeader}(H_i) \rrbracket)$ ;

```

to exercise every link in the network, a naive approach is to pick up a packet from each traffic class in each entry. However, a link may appear in more than one entry in the reachability-to-ports table. For example, in Table II, links $(p_{20}, p_{22} \rightarrow p_{31}, p_{30})$ in entry 4 can also be covered by entry 2 and entry 3. Therefore, we can improve by removing the test packets which exercise the same port. Algorithm 2 summarizes how to select the minimum test packets in the reachability-to-ports table.

The selecting is actually a Min-Set-Cover problem – a classical NP-hard problem. Lines 8-28 in Algorithm 2 provide the key steps to compute the approximately minimum number of test packets based on the idea in [20]. It first computes all the ports $\mathcal{L}_\Omega$ to be covered with the network configurations, and a sequence of port sets $H_1, H_2, H_3, \dots, H_L$ with the reachability-to-ports table (lines 8-9). For each set H_i , it computes its level k with the formula $q^k \leq |H_i| \leq q^{k+1}$, where $q \geq 1$ is a pre-specified value, which governs the approximation factor (lines 10-11). For each round, it checks all sets $H^{(k)}$ at the highest level. For each set H_i in $H^{(k)}$, it computes the value of $|H_i \setminus S|$, where S contains the ports covered so far. Then, it selects the set H_i with the largest $|H_i \setminus S|$, adds the ports covered by H_i to S , and generates a probe with the corresponding traffic class of H_i and

TABLE III
CONFIGURATION CONSTRUCT CHANGES AND ITS IMPACTS

Construct changes	Traffic classes	Edges in ETPG
OSPF area	All traffic classes	Inter-device
ISIS level	All traffic classes	Inter-device
BGP peer	All traffic classes	Inter-device
ACL	A specific traffic class	Inter-device
Route-policy	A specific destination	Inter-device
Static route	A specific destination	Inter-device
Route redistribution	All traffic classes	Intra-device

adds it to the probes P (lines 12-17). For other set in the highest level k , if $|H_i \setminus S| \leq q^k$, it will be dropped down to a lower level $\bar{k}$ by computing $q^{\bar{k}} \leq |H_i| \leq q^{\bar{k}+1}$ (lines 19–21). It repeats above computation until S includes all ports (lines 23–28).

Whenever H_i is processed, each of its ports is checked for presence in S . It is then either added to the solution, or it is added to a sub-collection of smaller sets. In the worst case, H_i is moved to the next sub-collection in each round. It is guaranteed to shrink by at least factor of q every second time it is moved. Hence, the worst case total number of items in all manifestations of H_i is $|H_i| + \frac{|H_i|}{q} + \frac{|H_i|}{q^2} + \dots = \frac{|H_i|}{1-1/q} = (1 + 1/(q-1))|H_i|$. Generating probes with reachability-to-ports table yields $\mathcal{O}([1 + 1/(q-1)]\sum_i |H_i|)$ complexity.

SWIFT can also identify the location of failure links. For every passing test and every failing test, it places all links exercised into a set of passing links and a set of potentially failing links respectively. It then derives the suspect links with the difference set of the potentially failing links and passing links. It next selects and sends reserved packets whose link histories contain exactly one link from the suspect set. If the sending of the reserved packets fails, it infers the failure link. It repeats this process for each reserved packet chosen, and finally outputs the suspect failure links.

IV. TEST PACKET UPDATE

Due to business service and security concern, network managers frequently update their policies by modifying the network configurations, such as, adding static routes, removing ACLs, updating route filters [13]. Such modifications are often relatively small [21]: e.g., in 85% of the data center networks, operators only modify one or two devices. It is really time consuming to reconstruct network model or regenerate probes with the updated configurations. To address this problem, we propose a fast test packet update approach, focusing on test packets impacted due to network changes.

A. ETPG Update

As the reachability-to-ports table is constructed with the ETPGs, it first updates these ETPGs with changed network configurations, each of which models the forwarding behavior for a specific traffic class. Configuration constructs impact traffic classes at different granularities (i.e., a specific traffic class, a specific destination or all traffic classes). For instance, adding an ACL will block or permit a specific traffic class defined in the ACL. Removing a route filter or a static route will change

Algorithm 3: updateETGs ($\mathcal{N}_\Omega, \overline{\mathcal{N}}_\Omega$).

Input: ETG_Ω : the ETGs; $\mathcal{N}_\Omega$: information parsing from network configurations; $\overline{\mathcal{N}}_\Omega$: information parsing from the updated network configurations.

Output: $\overline{ETG}_\Omega$: the updated ETGs.

```

1  $\overline{ETG}_\Omega \leftarrow \text{initETGs}(ETG_\Omega)$ ;
2  $\mathcal{U}_\Omega \leftarrow \text{getUpdatedConfig}(\mathcal{N}_\Omega, \overline{\mathcal{N}}_\Omega)$ ;
3 if  $\text{getUpdatedAll}(\mathcal{U}_\Omega) \neq \text{empty}$  then
  // Process routing adjacency configurations
  for all ETGs.
4   foreach  $\text{proc} \in \text{getRoutingProcess}(\llbracket \text{getUpdatedConfig}(\mathcal{U}_\Omega) \rrbracket)$  do
5     foreach  $\overline{\text{proc}} \in \text{getProcess}(\llbracket \text{getDevice}(\text{proc}) \rrbracket)$  do
6       if  $\text{redistribute}(\text{proc}, \overline{\text{proc}}) = \text{true} \wedge \text{routeFilter}(\overline{\text{proc}}, \llbracket \text{getDst}(tc_s^d) \rrbracket) = \text{false}$  then
7          $\overline{ETG}_\Omega \leftarrow \text{updateEdge}(\overline{\text{proc}}_i \rightarrow \text{proc}_o)$ ;
8       foreach  $\text{inf} \in \text{getInterface}(\text{proc})$  do
9         foreach  $\overline{\text{proc}} \in \text{getUsedProcess}(\llbracket \text{getPhyLink}(\text{inf}) \rrbracket)$  do
10          if  $\text{routeFilter}(\overline{\text{proc}}, \llbracket \text{getDst}(tc_s^d) \rrbracket) = \text{false}$  then
11             $\overline{ETG}_\Omega \leftarrow \text{updateEdge}(\text{proc}_o \rightarrow \overline{\text{proc}}_i)$ ;
12   foreach  $\text{dev} \in \text{getDevices}(\mathcal{U}_\Omega)$  do
13     if  $\text{existPhyLink}(\llbracket \text{getSrcDst}(tc_s^d) \rrbracket, \text{dev}) = \text{true}$  then
14       foreach  $\text{proc} \in \text{getProcess}(\text{dev})$  do
15          $\overline{ETG}_\Omega \leftarrow \text{updateEdge}(\text{src} \rightarrow \text{proc}_o, \text{proc}_i \rightarrow \text{dst})$ ;
16 else if  $\text{getUpdatedDst}(\mathcal{U}_\Omega) \neq \text{empty}$  then
  // Process static routes / router filters
  for ETGs with specific destinations.
17   foreach  $\text{rf} \in \text{getFilter}(\mathcal{U}_\Omega)$  do
18      $dETG_\Omega \leftarrow \text{getOverlap}(\text{rf}, ETG_\Omega)$ ;
19      $\overline{\text{proc}} \leftarrow \text{getUsedProcess}(\text{rf})$ ;
20      $\overline{ETG}_\Omega \leftarrow \text{updateEdge}(\llbracket \text{getOutEdge}(\overline{\text{proc}}) \rrbracket, dETG_\Omega)$ ;
21 else if  $\text{getUpdatedAcl}(\mathcal{U}_\Omega) \neq \text{empty}$  then
  // Process ACLs for ETGs with specific
  source-destination pair.
22   foreach  $\text{acl} \in \text{getFilter}(\mathcal{U}_\Omega)$  do
23      $tcETG_\Omega \leftarrow \text{getOverlap}(\text{acl}, ETG_\Omega)$ ;
24      $\overline{\text{inf}} \leftarrow \text{getUsedInterface}(\text{acl})$ ;
25      $\overline{ETG}_\Omega \leftarrow \text{updateEdge}(\llbracket \text{getRelatedProcess}(\overline{\text{inf}}) \rrbracket, tcETG_\Omega)$ ;

```

the behavior of the traffic classes whose destinations are defined in the route filter or the static route. Updating a BGP routing adjacency or an OSPF routing adjacency may impact all traffic classes. Based on the observations, we update ETPGs at different granularities to track the impact of the changed configurations.

The key steps to update ETPGs are summarized in Algorithm 3. It first computes the updated configuration constructs (lines 1-2). If there exist changed constructs (e.g., the routing adjacencies of routing protocols) that impact all traffic classes, it updates all the ETPGs in the following way. It identifies the routing processes corresponding to the changed configuration constructs. For each routing process, if no route filter is configured to deny the process' advertisements, it updates the corresponding intra-device edge (lines 4-7). Then, it updates the intra-device edges which models the routing exchange within a routing instance between two devices (lines 8-11). Finally, it

Algorithm 4: updateProbes ($\mathcal{N}_\Omega$).

Input: T : reachability-to-ports table; ETG_Ω : the ETGs; $\overline{ETG}_\Omega$: the updated ETGs; P : probes; $\mathcal{L}_\Omega$.

Output: $\overline{P}$: the updated probes.

```

1  $dETG_\Omega \leftarrow \text{getDiffETGs}(ETG_\Omega, \overline{ETG}_\Omega)$ ;
2  $\overline{T} \leftarrow \text{initUpdatedTable}(T)$ ;
3 foreach  $etg_s^d \in dETG_\Omega$  do
  // Update reachability-to-ports table.
4    $eh_s^d \leftarrow \text{updateHistory}(\llbracket \text{updateForwardingPath}(etg_s^d) \rrbracket, \mathcal{N}_\Omega)$ ;
5    $\overline{T} \leftarrow \text{updateEntry}(\llbracket \text{getIngress}(etg_s^d) \rrbracket, \llbracket \text{getEgress}(etg_s^d) \rrbracket, eh_s^d, \llbracket \text{getHeader}(etg_s^d) \rrbracket)$ ;
6  $\overline{\mathcal{L}}_\Omega \leftarrow \text{computePorts}(\overline{\mathcal{N}}_\Omega)$ ;
7  $E_r \leftarrow \text{getRemovedEntry}(T, \overline{T})$ ;
8  $\mathcal{U}_\Omega \leftarrow \text{initUncoveredPorts}()$ ;
9 foreach  $e_r \in E_r$  do
10   if  $\text{isPickedProbe}(\llbracket \text{getHeader}(e_r) \rrbracket, P) = \text{true}$  then
11      $h \leftarrow \text{getHistory}(e_r)$ ;
12      $\mathcal{U}_\Omega \leftarrow \text{addUncoveredPorts}(\mathcal{U}_\Omega, h, \overline{\mathcal{L}}_\Omega, \mathcal{L}_\Omega)$ ;
13      $P \leftarrow \text{removeProbe}(\llbracket \text{getHeader}(e_r) \rrbracket, P)$ ;
14  $E_a \leftarrow \text{getAddedEntry}(T, \overline{T})$ ;
15 foreach  $e_a \in E_a$  do
16    $\mathcal{U}_\Omega \leftarrow \text{addUncoveredPorts}(\mathcal{U}_\Omega, \llbracket \text{getHeader}(e_a) \rrbracket, \overline{\mathcal{L}}_\Omega, \mathcal{L}_\Omega)$ ;
17  $T_U \leftarrow \text{getUnusedEntry}(T, \overline{T}, P)$ ;
18  $P_U \leftarrow \text{computeGreedyProbe}(\mathcal{U}_\Omega, T_U)$ ;
19  $\overline{P} \leftarrow \text{appendProbe}(P, P_U)$ ;

```

updates edges between the SRC (DST) vertex and the routing process's out (in) vertex (lines 12-15).

Then, it accounts for the changed constructs (e.g., route filter, static route) that impact multiple traffic classes with a specific destination (lines 16-20). For each updated route filter, it loads the prefix, and computes the overlapping traffic classes whose destination address is overlapped the prefix (lines 17-18). It then identifies the routing processes which uses the route filter, and updates the corresponding inter-device edges in the ETPGs of the traffic classes impacted (lines 19-20). The updated static routes can be processed in a similar way. Finally, it accounts for the changed configuration constructs (i.e., ACL) that impact a specific traffic class (lines 21-25). An ACL is usually assigned to specific interfaces, which permits or blocks traffic from specific source to specific destination. It loads the source and destination prefixes in the updated ACL, and computes the overlapping traffic classes whose source address and destination address are overlapped the prefixes (lines 22-23). Then, it updates the corresponding inter-device edges in the digraphs of the impacted traffic classes (lines 24-25). Following the above steps, we can obtain the updated ETPGs for the changed network configurations, and these ETPGs can be used to update the reachability-to-ports table.

For instance, an ACL is added on router $R3$ in the network in Fig. 3, which blocks the traffic from hosts C^1 (10.0.6.0/26) to subnet E (10.0.3.2/24). The prefix of C^1 is overlapping with the prefix of subnet C (10.0.6.0/24). After computing nonoverlapping prefix, the existing traffic class ($C \rightsquigarrow E$) is splitted into the traffic class $C^1 \rightsquigarrow E$ and the traffic class $C^2 \rightsquigarrow E$. The ETPGs for these traffic classes are illustrated in Fig. 5.

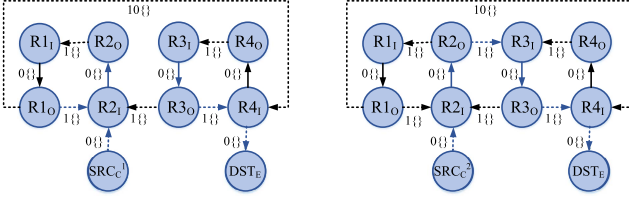


Fig. 5. ETPGs for the traffic class of $C^1 \rightsquigarrow E$ and the traffic class $C^2 \rightsquigarrow E$.

When the updated configuration constructs impact all traffic classes, updating ETPGs yields $\mathcal{O}(nd^2m^2 + nmr)$ complexity, where n is the number of traffic classes, d is the number of updated devices, m is number of the routing processes running on an updated device and r is the number of devices participating in a single updated routing instance. If the updated configuration constructs only contain route filters or static routes, it requires at most $\mathcal{O}(p)$ computations for each ETPG, where p is the number of updated route filters or static routes that impact that ETPG. Therefore, we can bound the running time of updating ETPGs by $\mathcal{O}(stp)$, where s is the number of endpoint groups that match with the updated route filters or static routes, and t is the number of total endpoint groups. If the updated constructs only contain ACLs, it requires at most $\mathcal{O}(e)$ computations for each ETPG, where e is the number of ACLs that impact this ETPG. Consequently, the computation cost of updating ETPGs is $\mathcal{O}(qe)$, where q is the number of endpoint groups that match with the ACLs. Consequently, we can bound the worst case running time of the ETPGs update by $\mathcal{O}(nd^2m^2 + nmr)$.

B. Packet Update With the Reachability-to-Ports Table

Lines 1–5 in Algorithm 4 summarize the key steps to update the reachability-to-ports table with the updated ETPGs. It starts by identifying the changed ETPGs, and initializing the updated reachability-to-ports table $\bar{T}$ (lines 1–2). For each changed ETPG, it computes the min-cost path between source vertex and destination vertex using a polynomial-time graph algorithm, and then updates the corresponding entry in the table $\bar{T}$ (lines 3–5). Following the above steps, we can obtain the updated reachability-to-ports table for the changed network configurations.

As the changed configuration constructs may only impact a few entries of the reachability-to-ports table, it is really time-consuming to regenerate probes with the updated reachability-to-ports table. We propose an approach to quickly update the probes such that the probes can still cover all ports in the network. The key steps are summarized in Algorithm 4 (lines 6–19). It computes the ports $\bar{\mathcal{L}}_\Omega$ to be covered with the updated network configuration, and identifies the changed entries in updated reachability-to-ports table. These changes can be classified into three categories: entry addition, entry removal and entry modification which can be considered as the removal of the old entry and addition of a new entry. For each entry e_r in the removed entries, if the packet in its header space is not selected as a probe, the removal of this entry cannot affect the probes. Otherwise, it removes the corresponding probe of this entry

from the existing probes. As each probe in the existing probes is selected to exercise some ports, the removal may result that some ports cannot be exercised now. In addition, not all of the history ports in the entry e_r are impacted, as some of ports can also be covered by the packets of other entries. Therefore, it updates the uncovered ports $\mathcal{U}_\Omega$ by computing the difference set of the history ports of the entry e_r and the ports $\mathcal{L}_\Omega$ (lines 9–13). For each entry in added entries, it checks whether the history ports of this entry can be exercised by the existing probes. If all of ports can be exercised by the existing probes, the addition of this entry do not impact the probe updates. Otherwise, it identifies the ports which cannot be covered by the existing probes (lines 14–16). Then, it identifies entries T_U whose header spaces are not used to generate the existing probes, and it selects the minimum test packets with the entries T_U to cover the ports $\mathcal{U}_\Omega$, following the probe generation approaches in Algorithm 2 (lines 17–19).

Updating the reachability-to-ports table requires at most $\mathcal{O}(rd^2m^2)$ computations, where r is the number of the updated ETPGs, d is the number of devices, and m is the maximum number of routing processes running on a single device. The trim on existing test packets yields $\mathcal{O}(p)$ complexity, where p is the number of test packets; and the identification of the table entries used to generate updated test packets requires at most $\mathcal{O}(t)$ computations, where t is the number of the entries in the updated reachability-to-ports table. The test packet selection with these table entries requires at most $\mathcal{O}([1 + 1/(q-1)]\sum_i |H_i|)$ computations, where q is the pre-specified value and H_i is the port set in each table entry. Therefore, we can bound the worst case running time of the test packet update by $\mathcal{O}(rd^2m^2 + [1 + 1/(q-1)]\sum_i |H_i|)$.

C. Discussion and Limitation

SWIFT has limitations in following three aspects: i) It can only use the terminals that are available, and test packets that terminals permit. All test packets generation frameworks share this limitation. ii) Similar to prior work [6], [18], in order to improve performance, we do not model the network control plane as a message-passing system. This choice means we ignore the middle transient states of the network. Therefore, SWIFT cannot uncover errors whose lifetime is in middle transient states. iii) Similar to configuration verification tools [5], [8], [22], SWIFT currently does not model additional protocols and features used in hybrid networks or virtual networks. Many networks have hybrid control planes that use both SDN and traditional distributed routing protocols; many virtual networks are designed in an overlay/underlay pattern that is implemented by encapsulating packets, e.g. using IP-in-IP, IP GRE, or VXLAN tunnels. We plan to extend SWIFT to support these features in our future research.

V. EVALUATION

We have implemented the above framework which consists of four major components: (i) a network configuration parser, (ii) a test packet generator, (iii) a test packet updater, and (iv) a monitor.

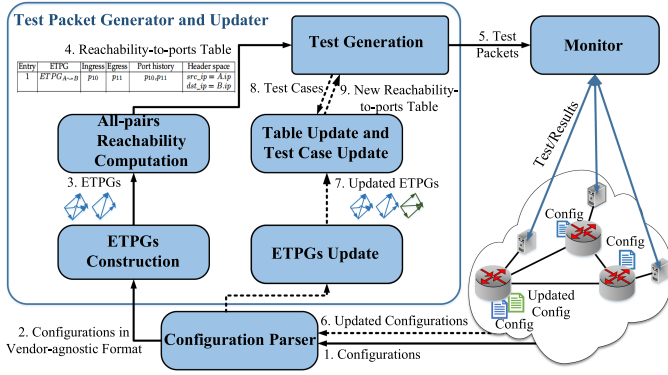


Fig. 6. The workflow of SWIFT.

Network configuration parser: As shown in Fig. 6, the parser can read router configurations collected by network managers. We construct the configuration parser based on the batfish parser [5] which widely used in control plane verification tools. Configurations from different routers could be written with different configuration languages (e.g., Cisco, Arista and JunOS). The parser can normalize these vendor-specific configurations to vendor-agnostic configuration.

Test packet generator: We have implemented the test packet generator in Java. The generator reads the vendor-agnostic configurations, converts these configurations to ETPGs, generates the reachability-to-ports table, and then executes the test packet generation algorithm. The generated test cases are sent to the monitor.

Test packet updater: We have implemented the test packet updater in Java. The updater loads the original vendor-agnostic configurations, the updated vendor-agnostic configurations, and the existing test packets. It then updates the reachability-to-ports tables with configurations, removes the invalid probes, computes entries which are not used to generate probes in the reachability-to-ports tables, and computes the ports to be tested. The packet generator is then informed to generate the new test cases with these entries and ports. Finally, the updated test cases are sent to the monitor.

Monitor. The monitor reads the stored generated test cases, and instructs the available test terminals (e.g., hosts) to send the appropriate probes, loads the feedback information of test terminals, and generate test reports. It then loads the updated test cases periodically, repeats the above process.

We have designed our experiments to answer two high-level questions: (i) How efficiently can SWIFT generate test packets? (ii) How efficient is SWIFT for the configuration update? All experiments are performed on a machine with 8 GB of RAM and Intel Core CPU running at 3.40 GHz.

A. Test Packet Generation

We evaluate SWIFT with benchmark network configurations including real network configurations and synthetic network configurations [23]. As shown in Fig. 7, these networks range in size from two to a few dozens of routers. The routers in

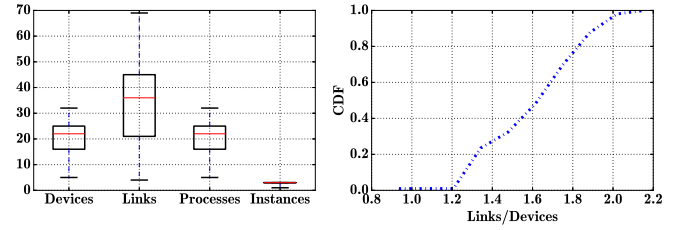


Fig. 7. Scale of the networks.

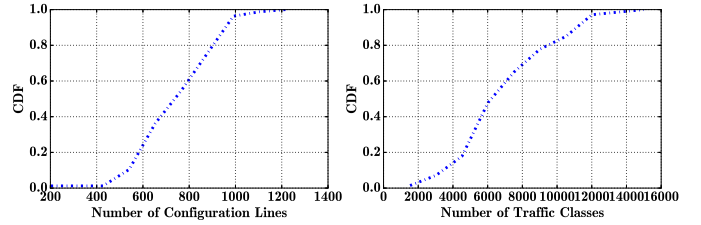


Fig. 8. Scale of the network configurations and scale of the traffic classes.

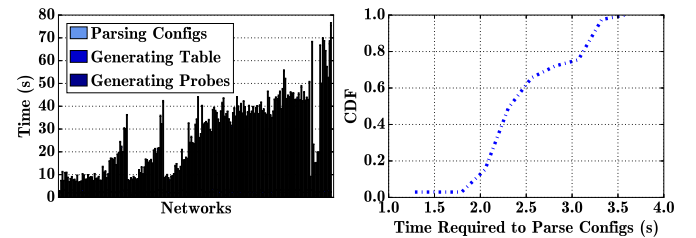


Fig. 9. Time for configuration parsing, reachability-to-ports table generation and test packet generation.

these networks are connected with one to several dozens of physical links. Fig. 7 provides the cumulative distribution of links/devices. The ratio of links to devices is less than 2 for 97% of the networks. The configurations of these networks include ISIS, OSPF, eBGP, iBGP, static routes, ACLs and route redistributions. Fig. 8 shows the scale of these networks configurations. The configurations range in size from ten to thousands of lines. Fig. 8 shows the scale of the traffic classes in these networks. We observe that these networks range in size from 200 to 14,000 of traffic classes. There are less than 14,000 traffic classes in 99% of these networks, and less than 10,000 traffic classes in 82% of networks.

1) Generation Efficiency: We evaluate the time of SWIFT required to generate test packets and the number of test packets generated. It is always desirable for the test packet generation time to be reasonably low, such that it is efficient to be used in practice. The number of test packets is expected to be reasonably small such as the regular traffic is not effected.

Time. Fig. 9 (left) shows the time for probe generation (including parsing network configurations, generating the reachability-to-ports table and generating the test packets with the reachability-to-ports table) for these networks. The time taken by these networks ranges from 4.8 s to 73.6 s, and the majority of the time is taken on generating the reachability-to-ports

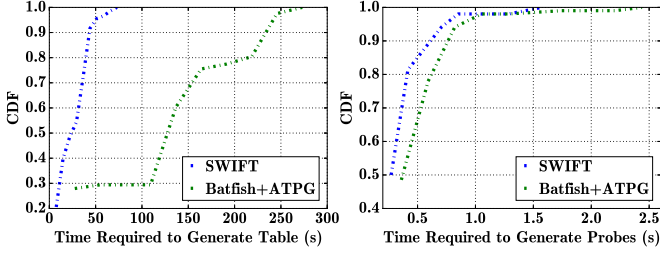


Fig. 10. CDF of the time cost for reachability-to-ports table generation and test packet generation.

TABLE IV
STATISTICS OF COMPARISON RESULT

Solution	Time to Generate Table			Time to Generate Probes		
	Highest	Lowest	Average	Highest	Lowest	Average
SWIFT	64.25	6.14	37.26	1.58	0.17	0.76
Batfish [5] + ATPG [3]	266.52	20.69	134.98	2.30	0.31	0.91

table. Fig. 9 (right) characterizes the cumulative distribution of the network configuration parsing time. The configuration parsing takes less than 3 s for 71% of the networks, and at most 3.5 s across all the networks. Fig. 10 characterizes the cumulative distribution of the reachability-to-ports table generation time. The time spent on generating the reachability-to-ports table for these networks ranges from 1.7 s to 71.3 s. The generation time is less than 30 s for 57% of the networks, and less than 40 s for 80% of the total networks. We evaluate the time required to generate ETPGs for each of the networks. We observe that this time for constructing the ETPGs is roughly correlated with the number of devices in the network (Pearson correlation co-efficient of 0.48), roughly correlated with the number of traffic classes in the network (Pearson correlation co-efficient of 0.74). The networks that take the longest are those with the most traffic classes. It takes less than 100 s to construct the ETPGs for that network which has 14 K traffic classes. The time per traffic class ranges from 3 ms to 10 ms (median 4.3 ms). As shown in Fig. 10 (right), the test packet generation time with reachability-to-ports table ranges from 0.28 s to 1.57 s. The generation time is less than 0.4 s for 80% of the networks, and less than 0.6 s for 89% of the networks.

We also evaluate storage space to save these ETPGs for each of the networks. We found that the storage space is roughly correlated with the number of devices in the network. Even that, 69% of networks' ETPGs take less than 100 MB of space when stored as serialized Java objects.

Someone may argue that running a simulator like Batfish [5] on configurations to get a data plane and then ATPG [3] on the data plane can generate test packets. On one hand, this approach does not support incremental test packet updates to react the network configuration updates. On the other hand, it computes test packets with simulation network model which makes the generation time not reasonably slow. Table IV shows statistics of comparison result. We observe that the time taken by Batfish [5]+ATPG [3] to generate test packets is much larger than the time taken by SWIFT.

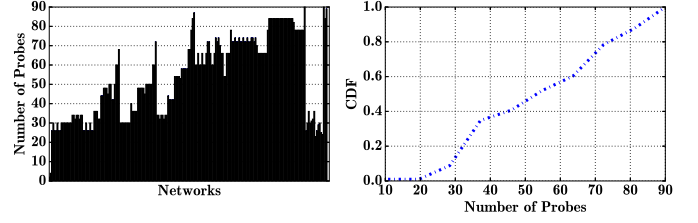


Fig. 11. Number of probes generated by SWIFT.

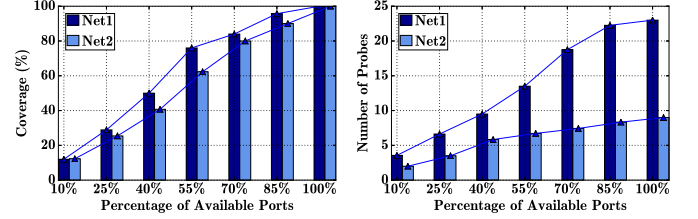


Fig. 12. Port coverage and number of probes versus percentage of available test terminals.

Number. Fig. 11 shows the number of test packets generated by SWIFT. The number of these test packets ranges from 22 to 90. We observe that 74% of the networks have fewer than 70 test packets, and 58% of the networks have fewer than 60 test packets. For each network, we also count the number of packets with the all-pair test packets generation strategy (i.e., each test agent sends ping test packets to other available test agents.). With coarse computation, the ratio of test packets generated by SWIFT to test packets generated by all-pair strategy is at most 5% across all the networks.

2) Coverage: Even network managers exclude external traffic, some internal ports of the network cannot be used to send test packets for security concerns [3]. This limitation is analogous to software testing in a program, which aims to test all code branches. Therefore, we evaluate the coverage of SWIFT with different percentage of *available ports* in networks.

To analyze how the coverage of ports relates to test terminals, we use two networks (i.e., Net1, Net2) [5] and randomly selects a certain percentage of ports as available ports. Net1 is a university network which consists of 3 border routers, 5 core routers and 13 distribution routers. These border routers run eBGP and iBGP for external connectivity and internal connectivity. The eBGP is used to connect the router with two provider ASes, and the iBGP is used to connect the router with core routers. Different from Net1, there are 17 routers in Net2 which run OSPF for connectivity.

Proportion and Number. Fig. 12 shows the port coverage (left) and the number of probes (right) for both Net1 and Net2 when the percentage of available test terminals is equal to 10%, 25%, 40%, 55%, 70%, 85% and 100%. For these networks, we run each test five times and take the mean. We observe that SWIFT has similar port coverage rates on Net1 and Net2. With 70% of the available test terminals, 84% of the ports of the Net1 can be exercised, and 80% of ports of the Net2 can be exercised in average. With 85% of the available test terminals, 94% of

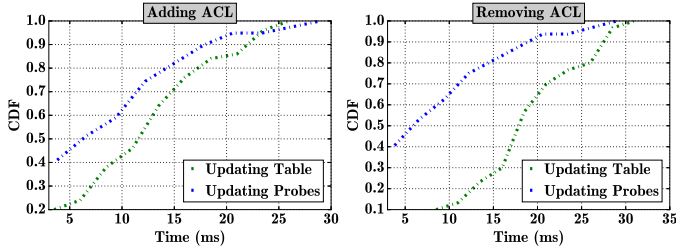


Fig. 13. CDF of the time cost of probe update for ACL update.

the ports of the Net1 can be exercised in average, while 90% of ports of the Net2 can be exercised. The result implies that the average value of port coverage of Net1 is higher than that of Net2. From the figure (right), we observe that, the number of probes increases as more test terminals are used in SWIFT. With 70% of the available test terminals, there are 18 test packets in Net1 and 7 test packets in Net2. With 85% of the available terminals, there are 23 test packets for Net1, and 9 test packets for Net2. The result implies that, compared with Net1, Net2 requires substantially fewer probes with the same percentage of available test terminals.

When all test terminals are available, each test for Net1 requires sending about 23 packets per port in the worst case. If these packets were sent over a single 1 Gb/s link, assuming each test packet is 100 bytes and not considering the propagation delay. Testing the entire network ten times every second would use less than 1% of the link bandwidth! Similarly, all the forwarding rules in Net2 can be tested using 8 test packets per port in the worst case. Testing the entire network ten times every second would also using less than 1% of the link bandwidth. One may argue that there are only dozens of links in Net1 and Net2. We also compute the cost of SWIFT for the network with the most traffic classes across all the networks we study [5]. Each test for this network requires sending about 85 packets per port in the worst case, and testing the entire network would use less than 1% of the link bandwidth (1 Gb/s).

B. Test Packet Update

1) *Update Efficiency*: As introduced early, network manager performs frequent network configuration updates due to security concern or business service concern. We now examine the update efficiency of SWIFT, by analyzing the time of test packet update with different types of configuration changes.

Time. Fig. 13 shows the cumulative distribution of the time when SWIFT reacts to an ACL addition and an ACL removal. We conduct 5 production runs for each of the ACL addition and ACL removal operation, and take the mean for the time cost. From the figure (left), we observe that, for reachability-to-ports table update, SWIFT takes less than 10 ms for 63% of the networks, and less than 15 ms for 83% of the networks. For test packet update with the reachability-to-ports table, it takes less than 13.4 ms for 60% of the networks, and less than 17.1 ms for 80% of the networks we study. As shown in the figure (right), to react to the removal of an ACL filter, SWIFT takes at most

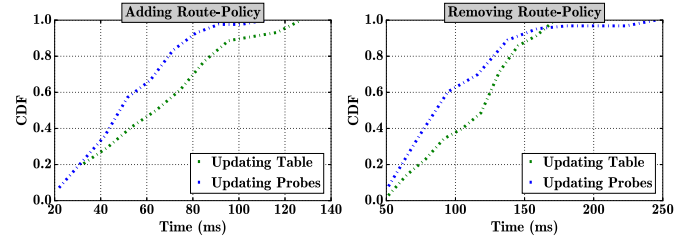


Fig. 14. CDF of the time cost of probe update for route-policy update.

62 ms across all the networks we study. For 80% of the networks, it takes fewer than 14.7 ms for the reachability-to-ports table update. For test packet update with the reachability-to-ports table, it takes fewer than 26.9 ms for 80% of the networks. The result also implies that the time used to react to the removal of an ACL is a bit more than the time used to react to the addition of an ACL.

Fig. 14 characterizes the time cost of SWIFT to react to the route-policy update. For the addition of route-policy, 60% of the networks take less than 56.2 ms to update the reachability-to-ports table, and 80% of the networks take less than 79.1 ms. For test packet update with the reachability-to-ports table, it takes less than 87.4 ms for 80% of the networks we study. As shown in the figure (right), for the removal of route-policy, it takes less than 145 ms for 78% of the networks to update the reachability-to-ports table. 80% of the networks take less than 142.5 ms to update test packets, and 95% of the networks take less than 154.2 ms. The result also implies that the time used to react to the removal of an route-policy is a bit more than that to react to the time used to react to the addition of an route-policy.

C. Testing in a Production Network

We deployed an experimental SWIFT system in the network of Beijing Jiaotong University. The network [24] consists of dozens of routers, hundreds of switches, and 46,000 network information points, which provides the network service for 36,500 users in the teaching area, student apartment area and family area. For test terminals, we utilized the hundreds of WiFi access points that were already installed throughout the university. This allowed us to achieve high coverage on links.

Fig. 15 provides the network topology of student apartment area. The network consists of the access, the convergence, and the core layers. There are more than 200 H3C-E026 switches in the access layer, which connects client devices, such as laptops, servers and peripherals on the internal network. If a host accesses a resource in some other network, it sends the traffic to the convergence layer. The convergence layer handles packet filtering and determines how packets can reach the core layer with a few dozens of H3C-S5516 Gigabit Ethernet switches. The core layer consists of several H3C-S8505 10 Gigabit Ethernet switches and NE40E routers, and connects with the networks of Cernet2, China Mobile, China Unicom, China Telecom and so on. The networks of teaching area, family area and data center are similarly connected to the backbone of the university network. As an aside, the data center network consists of dozens

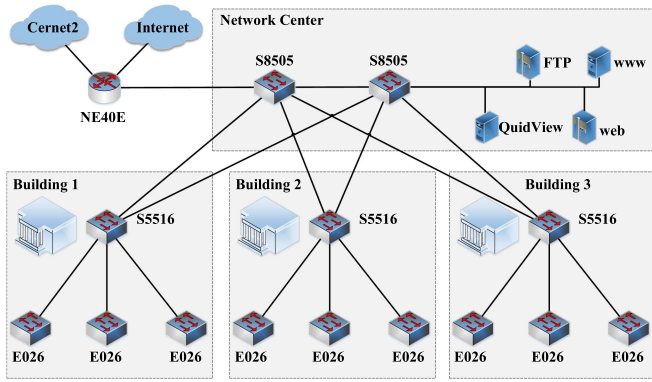


Fig. 15. The network topology of student apartment area.

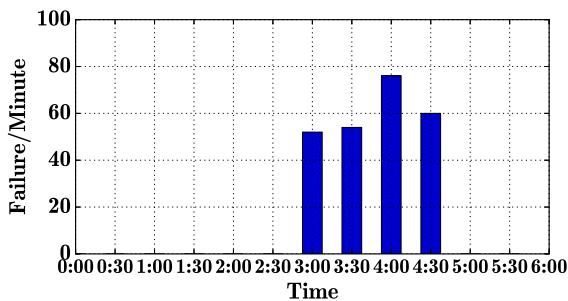


Fig. 16. The Dec 2, 2020 production network outages captured by SWIFT.

of H3C-S12500R routers, and its topology looks like a fat-tree topology.

From December 2020 to January 2021, the SWIFT system was used in the university network. Through the interview with the network managers, we knew that configuration updates of this university network are often carried out at night. Therefore, the central server loads the network configurations every 6 hours from 6:00 to 24:00, and every 30 minutes from 0:00 to 6:00. Each test agent downloads a list of ping targets from a central server every 30 minutes, and conducts ping tests every 30 seconds.

The test packets sent by SWIFT on each available port range from 3,557 to 4,085. Testing this production network uses less than 1% of the link bandwidth (10 Gb/s) in the worst case.

During the experiment, we found three major network outages. The first outages occurred on December 2 2020. Fig. 16 shows the number of failed test cases during that period. SWIFT only used 118 test packets in each run during the first outages. It used 144 test packets per round during the second outages. SWIFT found the third outages on January 24 2021. There are 38 failed test cases during this period, and SWIFT used 126 test packets per round. The network managers confirmed that the first outage was caused by the powered off of the border router in the Shaw Building. The powered off of the border router were compliant in Shaw Building's computer room construction and renovation project. The second outage on January 11 2021 was caused by the failed links in the weak electricity room of Building No. 7 of Changyuan in the family area. The network managers confirmed that the water pipe leaked, and the water

entered the weak electricity room and caused damage to the links. For the third outage, the managers were unable to find the root cause. Despite that, we emphasize that SWIFT correctly detected the outage in these cases and pinpointed the affected links.

VI. RELATED WORK

Researchers have recently proposed a lot of approaches to verify or test network control plane or data plane. These approaches can be divided into four classes: data plane testing, data plane verification, control plane verification and control plane testing.

Data Plane Testing: (i) The work closest to ours are ATPG [3] and Pronto [11], which collects FIBs and uses test packets to exercise every link. ATPG generates end-to-end test cases via header space analysis [25], and Pronto improves the speed of test case generation via the AP approach [26]. However, the stable and consistent data plane snapshots are not available for these tools. In addition, none of these tools support updating test packets incrementally. (ii) BUZZ [27] generates test cases to ensure that the stateful data plane elements meet the context-dependent policies. Its goal is to test the logical correctness of the data plane functions, not the correctness of the actual implementation of data plane in our work. (iii) Some recent works, like Monocle [28], RuleScope [29] and RuleChecker [30], propose to generate test packets to check every switch. However, they design their network models with flow tables, and only work in software defined networks.

Data Plane Verification: Various approaches (e.g., HSA [25], VMN [31]) have been proposed to verify the reachability behavior of the data plane [2], [9], [32], [33], [34]. These tools convert the data-plane information into boolean expressions or other formal models, and check whether a network invariant (e.g., reachability, forwarding loop) would be satisfied. The result from such verification is valid only for the specific data plane snapshot. However, they can detect the errors in FIBs or flow tables, but not the problems in actual implementation of data plane.

Control Plane Verification: Compared with data plane verification, control plane verification enables more powerful analysis. Control plane verification tools are designed to detect configuration errors proactively, before configurations are deployed. Batfish [5] simulates routing protocols under a given environment to obtain and analyze the resulting data plane. Hoyan [35] improves the the simulation model of Batfish by encoding the route announcement with the variables of link state. SRE [8] improves Hoyan with the symbolic router execution technology, which computes the reachability with the symbolic packet header and symbolic routing tables. ARC [18] abstracts the control plane as abstract weighted graphs, and verify the network invariants using graph algorithms. ERA [6] and Minesweeper [1] analyze control plane via the abstract routing records. DNA [22] improves the verification by taking control plane changes into account, and outputs consequent differences due to configuration changes. Our work is complementary to these efforts, which can capture the problems in the actual implementation of the network data plane.

Control Plane Testing: In software defined networks, many studies contribute to control plane testing. Some works (e.g., NICE [36], DeLorean [37], and Yakuwa et al. [38]) extract formal models from the source codes of SDN applications, and perform model-based testing. Others [39], [40] propose black-box testing methods for SDN applications. For example, STS [39] tests multiple controllers via a black-box fuzz testing method. They design the network models with the source codes of SDN applications, and these models cannot be extended to support configuration-based networks which are dominant currently. Unlike these studies, our work uses the configuration information in the distributed control plane to proactively check the liveness properties.

VII. CONCLUSION

In this paper, we present a systematic technique SWIFT that automatically generates probes to test network data plane. It is the first step to generate test packets with network configurations. SWIFT is able to convert the network configurations into extended traffic propagation graphs, and generate the minimum set of test packets to exercise all interfaces in the network. It can also react to the network configuration updates by incrementally updating probes in a short time. The evaluation suggests that SWIFT is efficient and useful for testing the actual implementation of the data plane. In future work, we will extend SWIFT to automatically isolate faulty devices or links; and automatically generate minimal configuration repairs to improve the fault tolerance of the network.

REFERENCES

- [1] R. Beckett, A. Gupta, R. Mahajan, and D. W. P. University, "A general approach to network configuration verification," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2017, pp. 155–168.
- [2] N. P. Lopes, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 499–512.
- [3] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," *IEEE/ACM Trans. Netw.*, vol. 22, no. 2, pp. 554–566, Apr. 2014.
- [4] G. C. Whittaker, "Network outages like nyse, united airlines, are the new natural disasters. understanding BGP path selection," Jul. 2015. [Online]. Available: <http://bit.ly/1HW9wgr>
- [5] A. Fogel et al., "A general approach to network configuration analysis," in *Proc. 12th USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 469–483.
- [6] S. K. Fayaz et al., "Efficient network reachability analysis using a succinct control plane representation," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 217–232.
- [7] S. Prabhu, A. Kheradmand, B. Godfrey, and M. Caesar, "Predicting network futures with plankton," in *Proc. Asia-Pacific Workshop Netw.*, 2017, pp. 92–98.
- [8] A. G.-J. Peng Zhang and D. Wang, "Symbolic router execution," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2022, pp. 214–227.
- [9] A. Horn, A. Kheradmand, and M. Prasad, "Delta-Net: Real-time network verification using atoms," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, USENIX Association, Boston, MA, 2017, pp. 735–749.
- [10] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 467–472, 2012.
- [11] Y. Zhao, H. Wang, X. Lin, T. Yu, and C. Qian, "Pronto: Efficient test packet generation for dynamic network data planes," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 13–22.
- [12] H. Zeng et al., "Libra: Divide and conquer to verify forwarding tables in huge networks," in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation* USENIX Association, Seattle, WA, 2014, pp. 87–99.
- [13] A. Gember-Jacobson, C. Raiciu, and L. Vanbever, "Integrating verification and repair into the control plane," in *Proc. ACM Workshop Hot Topics Netw.*, 2017, pp. 129–135.
- [14] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't mind the gap: Bridging network-wide objectives and device-level configurations," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 328–341.
- [15] Y.-W. E. Sung, X. Tie, S. H. Wong, and H. Zeng, "Robotron: Top-down network management at facebook scale," in *Proc. ACM SIGCOMM Conf.*, Association for Computing Machinery, New York, NY, USA, 2016, pp. 426–439.
- [16] "Cisco ios configuration fundamentals command reference," 2022. [Online]. Available: http://www.cisco.com/c/en/us/td/docs/ios/fundamentals/command/reference/cf_book.pdf
- [17] "Juniper ios configuration fundamentals command reference," 2022. [Online]. Available: <https://www.juniper.net/documentation/us/en/software/jatp/jatp-cli-reference/jatp-cli-reference.pdf>
- [18] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 300–313.
- [19] E. W. Fulp, "Optimization of network firewall policies using directed acyclic graphs," in *Proc. IEEE Internet Manage. Conf.*, 2005, pp. 62–67.
- [20] G. Cormode, H. Karloff, and A. Wirth, "Set cover algorithms for very large datasets," in *Proc. ACM Int. Conf. Inf. Knowl. Manage.*, 2010, Art. no. 479.
- [21] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan, "Management plane analytics," in *Proc. Internet Meas. Conf.*, Association for Computing Machinery, New York, NY, USA, 2015, pp. 395–408.
- [22] P. Zhang, A. Gember-Jacobson, Y. Zuo, Y. Huang, X. Liu, and H. Li, "Differential network analysis," in *Proc. 19th USENIX Symp. Netw. Syst. Des. Implementation*, USENIX Association, Renton, WA, USA, 2022, pp. 601–615.
- [23] "Network configurations," 2022. [Online]. Available: <https://github.com/network-network/network-configurations>
- [24] "Campus network of beijing jiaotong university," 2018. [Online]. Available: <http://ic.bjtu.edu.cn/zxjj/index.htm>
- [25] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 9–9.
- [26] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Trans. Netw.*, vol. 24, no. 2, pp. 887–900, Apr. 2016.
- [27] S. K. Fayaz et al., "Buzz: Testing context-dependent policies in stateful networks," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2016, pp. 275–289.
- [28] P. Pereini, M. Kuniar, and D. Kosti, "Monocle: Dynamic, fine-grained data plane monitoring," in *Proc. ACM Conf.*, 2015, pp. 1–13.
- [29] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, "Is every flow on the right track?: Inspect SDN forwarding with rulescope," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.
- [30] P. Zhang, C. Zhang, and C. Hu, "Fast testing network data plane with rulechecker," in *Proc. IEEE Int. Conf. Netw. Protoc.*, 2017, pp. 1–10.
- [31] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying Reachability in Networks With Mutable Datapaths," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, USENIX Association, Boston, MA, 2017, pp. 699–718.
- [32] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 290–301, 2011.
- [33] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "SymNet: Scalable symbolic execution for modern networks," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 314–327.
- [34] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Programm. Lang.*, 2016, pp. 69–83.
- [35] F. Ye et al., "Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global Wan," in *Proc. ACM SIGCOMM Conf.*, Association for Computing Machinery, New York, NY, USA, 2020, pp. 599–614.
- [36] M. Canini, D. Venzano, P. Perešini, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, USENIX, San Jose, CA, 2012, pp. 127–140.
- [37] J. Croft, R. Mahajan, M. Caesar, and M. Musuvathi, "Systematically exploring the behavior of control programs," in *Proc. USENIX Conf. USENIX Tech. Conf.*, 2015, pp. 165–176.
- [38] Y. Yakuwa, N. Tomizawa, and T. Tonouchi, "Efficient model checking of openflow networks using SDPDR-DS," in *Proc. 16th Asia-Pacific Netw. Operations Manage. Symp.*, 2014, pp. 1–6.

- [39] C. Scott et al., "Troubleshooting blackbox sdn control software with minimal causal sequences," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 395–406.
- [40] D. Lebrun, S. Vissicchio, and O. Bonaventure, "Towards test-driven software defined networking," *Proc. IEEE Netw. Operations Manage. Symp.*, 2014, pp. 1–9.



Zhiliang Wang (Senior Member, IEEE) received the BE, ME and PhD degrees in computer science from Tsinghua University, China, in 2001, 2003 and 2006, respectively. Currently, he is an associate professor with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include formal methods and protocol testing, next-generation Internet, and network measurement.



Yahui Li received the BS degree from the School of Software Engineering, Jilin University, Changchun, China, and the PhD degree from Tsinghua University, Beijing, China. She is currently working with the Institute for Network Sciences and Cyberspace, Tsinghua University. Her research concerns computer network systems and network security.



Xia Yin received the BE, ME and PhD degrees in computer science from Tsinghua University, in 1995, 1997 and 2000, respectively. She is a full professor with the Department of Computer Science and Technology, Tsinghua University. Her research interests include future Internet architectures, formal methods, protocol testing and large-scale Internet routing.



Han Zhang (Member, IEEE) received the BS degree in computer science and technology from Jilin University, Changchun, China, and the PhD degree from Tsinghua University, Beijing, China. He is currently working with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research concerns computer network systems and network security.



Xingang Shi (Member, IEEE) received the BS degree from Tsinghua University and the PhD degree from the Chinese University of Hong Kong. He is currently working with the Institute for Network Sciences and Cyberspace at Tsinghua University. His research interests include network measurement and routing protocols.



Jilong Wang (Senior Member, IEEE) received the PhD degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2000. He is currently a professor with Tsinghua University. His research focuses on network measurement, location-oriented network, SDN systems, and network security.



Jianping Wu (Fellow, IEEE) received the BS, MS, and PhD degrees from Tsinghua University. He is a full professor and the director of the Network Research Center and PhD supervisor with the Department of Computer Science, Tsinghua University. Since 1994, he has been in charge of the China Education and Research Network (CERNET). He is a member of the Information Advisory Committee, Office of National Information Infrastructure, Secretariat of State Council of China and is also a vice president of the Internet Society of China (ISC). His research interests include next-generation Internet, IPv6 deployment and technologies, and Internet protocol design and engineering.