A DEVOTED APPROACH TO TEST THE LIVENESS OF NETWORK

Dr.M. Ramesh Kumar[1], Dr. S.R.Boselin Prabhu[2], P.Ponni[3], D.Arthi[4], P.Preethi[5]
[1]Associate Professor, Department of Computer Science and Engineering,
VSB College of Engineering Technical Campus, Coimbatore, TamilNadu, India.
[2]Associate Professor, Department of Electronics and Communication Engineering,
VSB College of Engineering Technical Campus, Coimbatore, TamilNadu, India.
[3,4,5]Asst Professor, Department of Computer Science and Engineering,
VSB College of Engineering Technical Campus Coimbatore, TamilNadu, India.

Abstract

Automatic test packet generation is a framework model that test the liveness, congestion and performance of an network. The goal of the system is to automatically generate test packets to test the network, and pinpoint faults. This methodology reduces the data loss and the packet truncation from the network also it decreases the unwanted data congestion in the network.

Networks are getting larger and more complex, yet administrators rely on rudimentary tools such as and to debug problems. An automated and systematic approach for testing and debugging networks is called "Automatic Test Packet Generation" (ATPG). ATPG reads router configurations and generates a device-independent model. The model is used to generate a minimum set of test packets to (minimally) exercise every link in the network or (maximally) exercise every rule in the network. Test packets are sent periodically, and detected failures trigger a separate mechanism to localize the fault. ATPG can detect both functional (e.g., incorrect firewall rule) and performance problems (e.g., congested queue). ATPG complements but goes beyond earlier work in static checking (which cannot detect liveness or performance faults) or fault localization.

Keywords: Test packet generation, Test Packet Selection, Fault localization, Network Troubleshooting.

1. INTRODUCTION

ATPG (acronym for both **A**utomatic **T**est **P**attern **G**eneration and **A**utomatic **T**est **P**attern **G**enerator) is an electronic design automation method/technology used to find an input (or test) sequence that, when applied to a digital circuit, enables automatic test equipment to distinguish between the correct circuit behavior and the faulty circuit behavior caused by defects. The generated patterns are used to test semiconductor devices after manufacture, and in some cases to assist with determining the cause of failure (failure analysis.) The effectiveness of ATPG is measured by the amount of modeled defects, or fault models, that are detected and the number of generated patterns. These metrics generally indicate test quality (higher with more fault detections) and test application time (higher with more patterns). ATPG efficiency is another important consideration. It is influenced by the fault model under consideration, the type of circuit under test (full scan, synchronous sequential, or asynchronous sequential), the level of abstraction used to represent the circuit under test (gate, register-transfer, switch), and the required test quality. room without any wired connection to the presentation computer.

A defect is an error caused in a device during the manufacturing process. A fault model is a mathematical description of how a defect alters design behavior. The logic values observed at the device's primary outputs, while applying a test pattern to some device under test (DUT), are called the output of that test pattern. The output of a test pattern, when testing a fault-free device

that works exactly as designed, is called the expected output of that test pattern. A fault is said to be detected by a test pattern if the output of that test pattern, when testing a device that has only that one fault, is different than the expected output. The ATPG process for a targeted fault consists of two phases: fault activation and fault propagation. Fault activation establishes a signal value at the fault model site that is opposite of the value produced by the fault model. Fault propagation moves the resulting signal value, or fault effect, forward by sensitizing a path from the fault site to a primary output.

Testing liveness of a network is a fundamental problem for ISPs and large data center operators. Sending probes between every pair of edge ports is neither exhaustive nor scalable . It suffices to find a minimal set of end-to-end packets that traverse each link. However, doing this requires a way of abstracting across device specific configuration files (e.g., header space), generating headers and the links they reach (e.g., all-pairs reach¬ability), and finally determining a minimum set of test packets (Min-Set-Cover). Even though the networks have become complex in nature,network engineers still rely on basic tools and methods for debugging which requires an network engineer's utmost concentration and dedication.

Network engineers past experience and wisdom is put into test when a network is troubleshooted and this is difficult and time consuming.So,our goal is to automatically detect the reasons for troubleshooting and localizing the faults. Consider two examples.

Example 1: Suppose a router with a faulty line card starts dropping packets silently. Alice, who administers 100 routers, receives a ticket from several unhappy users complaining about connectivity. First, Alice examines each router to see if the configuration was changed recently and concludes that the configuration was untouched. Next, Alice uses her knowledge of the topology to triangulate the faulty device with `Ping` and `traceroute` . Finally, she calls a colleague to replace the line card.

Example 2: Suppose that video traffic is mapped to a specific queue in a router, but packets are dropped because the token bucket rate is too low. It is not at all clear how Alice can track down such a performance fault using `Ping` and `tracercute` .

Troubleshooting a network is difficult for three reasons. First, the forwarding state is distributed across multiple routers and firewalls and is defined by their forwarding tables, filter rules, and other confi guration parameters. Second, the forwarding state is hard to observe because it typically requires manu-ally logging into every box in the network. Third, there are many different programs, protocols, and humans updating the forwarding state simultaneously. When Alice uses `Ping` and `traceroute` , she is using a crude lens to examine the current forwarding state for clues to track down the failure.

It is a simplified view of network state. At the bottom of the figure is the forwarding state used to forward each packet, consisting of the L2 and L3 forwarding information base (FIB), access control lists, etc. The forwarding state is written by the control plane (that can be local or remote as in the SDN.

The model and should correctly implement the network ad-ministrator's policy. Examples of the policy include: "Security group X is isolated from security Group Y," "Use OSPF for routing," and "Video traffic should receive at least 1 Mb/s."

We can think of the controller compiling the policy (A) into device-specific configuration files (B), which in turn determine the forwarding behavior of each packet (C). To ensure the net-work behaves as designed, all three steps should remain consis-tent at all times, i.e., $A=B=C$. In addition, the topology, shown to the bottom right in the figure, should also satisfy a set of liveness properties L. Minimally, L requires that sufficient links and nodes are working; if the control plane specifies that a laptop can access a server, the desired outcome can fail if links fail. L can also specify performance guarantees that detect flaky links.

Recently, researchers have proposed tools to check that $A=B$, enforcing consistency between policy and the config-uration. While these approaches can find (or prevent) software logic errors in the control plane, they are not designed to identify liveness failures caused by failed links and routers, bugs caused by faulty router hardware or software, or performance problems caused by network congestion. Such failures

require checking for L and whether B–C. Alice's first problem was with L (link not working), and her second problem was with B–C (low level token bucket state not reflecting policy for video bandwidth).

In fact, we learned from a survey of 61 network operators that the two most common causes of net-work failure are hardware failures and software bugs, and that problems manifest themselves both as reachability failures and throughput/latency degradation. Our goal is to automatically de-tect these types of failures.

The main contribution of this paper is what we call an Auto-matic Test Packet Generation (ATPG) framework that automat-ically generates a minimal set of packets to test the liveness of the underlying topology and the congruence between data plane state and configuration specifications. The tool can also auto-matically generate packets to test performance assertions such as packet latency. In Example 1, instead of Alice manually de-ciding which Ping packets to send, the tool does so periodically on her behalf. In the Example the tool determines that it must send packets with certain headers to "exercise" the video queue, and then determines that these packets are being dropped.

ATPG detects and diagnoses errors by independently and ex-haustively testing all forwarding entries, firewall rules, and any packet processing rules in the network. In ATPG, test packets are generated algorithmically from the device confi guration files and FIBs, with the minimum number of packets required for complete coverage. Test packets are fed into the network so that every rule is exercised directly from the data plane.
Since ATPG treats links just like normal forwarding rules, its full cov-erage guarantees testing of every link in the network. It can also be specialized to generate a minimal set of packets that merely test every link for network liveness. At least in this basic form, we feel that ATPG or some similar technique is fundamental to networks: Instead of reacting to failures, many network operators such as Internet2 proactively check the health of their network using pings between all pairs of sources. How-ever, all-pairs Ping does not guarantee testing of all links and has been found to be unscalable for large networks such as PlanetLab .

Organizations can customize ATPG to meet their needs; for example, they can choose to merely check for network liveness (link cover) or check every rule (rule cover) to ensure security policy. ATPG can be customized to check only for reachability or for performance as well.

ATPG can adapt to constraints such as requiring test packets from only a few places in the network or using special routers to generate test packets from every port. ATPG can also be tuned to allocate more test packets to exer-cise more critical rules. For example, a healthcare network may dedicate more test packets to Firewall rules to ensure HIPPA compliance.

Put another way, we can check every rule in every router on the Stanford backbone 10 times every second by sending test packets that consume less than 1% of network bandwidth. The link cover for Stanford is even smaller, around 50 packets, which allows proactive liveness testing every millisecond using 1% of net-work bandwidth.

2. A SURVEY OF NETWORK OPERATORS

There are six most common symptoms, four cannot be detected by static checks of the type $A - B$ (throughput/ latency, intermittent connectivity, router CPU utilization, con-gestion) and require ATPG-like dynamic testing. Even the re-maining two failures (reachability failure and security Policy Violation) may require dynamic testing to detect forwarding plane failures.

Category	Avg	% of ≥ 4
Reachability Failure	3.67	56.90%
Throughput/Latency	3.39	52.54%
Intermittent Connectivity	3.38	53.45%
Router CPU High Utilization	2.87	31.67%
Congestion	2.65	28.07%
Security Policy Violation	2.33	17.54%
Forwarding Loop	1.89	10.71%
Broadcast/Multicast Storm	1.83	9.62%

(a)

Category	Avg	% of ≥ 4
Switch/Router Software Bug	3.12	40.35%
Hardware Failure	3.07	41.07%
External	3.06	42.37%
Attack	2.67	29.82%
ACL Misconfig.	2.44	20.00%
Software Upgrade	2.35	18.52%
Protocol Misconfiguration	2.29	23.64%
Unknown	2.25	17.65%
Host Network Stack Bug	1.98	16.00%
QoS/TE Misconfig.	1.70	7.41%

(b)

a) Symptoms of Network Failure
b) Causes of Network Failure

Causes: The two most common symptoms (switch and router software bugs and hardware failure) are best found by dynamic testing.

Cost of troubleshooting: Two metrics capture the cost of network debugging—the number of network-related tickets per month and the average time consumed to resolve a ticket (Fig. 2). There are 35% of networks that generate more than 100 tickets per month. Of the respondents, 40.4% estimate it takes under 30 min to resolve a ticket. However, 24.6% report that it takes over an hour on average.

Tools: Table II shows that ping, traceroute and SNMP are by far the most popular tools. When asked what the ideal tool for network debugging would be, 70.7% reported a desire for automatic test generation to check performance and correctness. Some added a desire for "long running tests to detect jitter or intermittent issues," "real-time link capacity monitoring," and "monitoring tools for network state."

In summary, while our survey is small, it supports the hypoth-esis that network administrators face complicated symptoms and causes.

The cost of debugging is nontrivial due to the frequency of problems and the time to solve these problems. Classical tools such as `ping` and `traceroute` are still heavily used, but administrators desire more sophisticated tools.

3. TEST PACKET GENERATION ALGORITHM:

1) Algorithm: We assume a set of test terminals in the network can send and receive test packets. Our goal is to generate a set of test packets to exercise every rule in every switch function, so that any fault will be observed by at least one test packet.

This is analogous to software test suites that try to test every possible branch in a program. The broader goal can be limited to testing every link or every queue. When generating test packets,

ATPG must respect two key constraints:

1) Port: ATPG must only use test terminals that are available;

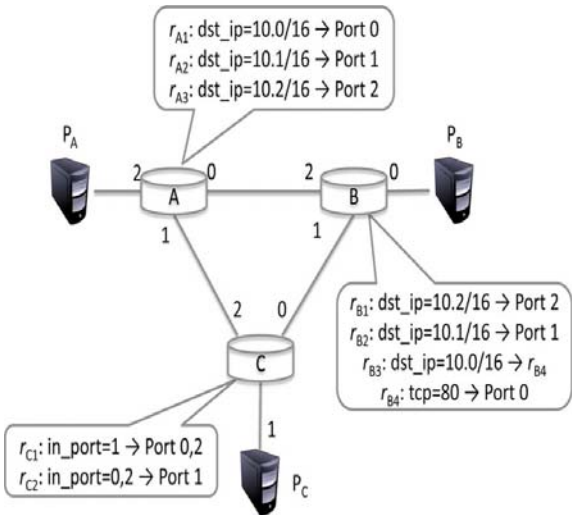
2)Header: ATPGmust only use headers that each test terminal is permitted to send. For example, the network administrator may only allow using a specific set of VLANs.

3.1 TEST PACKET SELECTION:

For a network with the switch functions, { $T_1,..,T_n$} and topology function Γ, determine the minimum set of test packets to exercise all reachable rules, subject to the port and header constraints.

ATPG chooses test packets using an algorithm we call Test Packet Selection (TPS). TPS first finds all equivalent classes between each pair of available ports.

An equivalent class is a set of packets that exercises the same combination of rules. It then samples each class to choose test packets, and finally compresses the resulting set of test packets to find the minimum covering set.



Example topology with three switches

ALL-PAIRS REACHABILITY TABLE:

Header	Ingress Port	Egress Port	Rule History
h_1	p_{11}	p_{12}	$[r_{11}, r_{12}, \ldots]$
h_2	p_{21}	p_{22}	$[r_{21}, r_{22}, \ldots]$
...	...	...	...
h_n	p_{n1}	p_{n2}	$[r_{n1}, r_{n2}, \ldots]$

3.2 PROPERTIES:

The TPS algorithm has the following useful properties.

Property 1 (Coverage): The set of test packets exercise all reachable rules and respect all port and header constraints.

Proof Sketch: Define a rule to be reachable if it can be exercised by at least one packet

satisfying the header constraint, and can be received by at least one test terminal. A reachable rule must be in the all-pairs reachability table; thus, set cover will pick at least one packet that exercises this rule. Some rules are not reachable: For example, an IP prefix may be made unreachable by a set of more specific prefixes either deliberately
(to provide backup) or accidentally (due to misconfiguration).

Property 2 (Near-Optimality): The set of test packets selected by TPS is optimal within logarithmic factors among all tests giving complete coverage.

Proof Sketch: This follows from the logarithmic (in the size of the set) approximation factor inherent in Greedy Set
Cover.

Property 3 (Polynomial Runtime): The complexity of finding test packets is $O(TDR^2)$ where is the number of test terminals, D
is the network diameter, and R is the average number of rules in each switch.

Proof Sketch: The complexity of computing reachability from one input port is $O(DR^{2)}$ [16], and this computation is repeated for each test terminal.

4. FAULT LOCALIZATION:

ATPG periodically sends a set of test packets. If test packets fail, ATPG pinpoints the fault(s) that caused the problem.

1) Fault Model: A rule fails if its observed behavior differs
from its expected behavior. ATPG keeps track of where rules fail using a result function . For a rule , the result function is defined as

R(r,pk)= 0, if pk fails at rule r
R(r,pk)= 1, if pk succeeds at rule r.

"Success" and "failure" depend on the nature of the rule: A forwarding rule fails if a test packet is not delivered to the intended
output port, whereas a drop rule behaves correctly when packets are dropped. Similarly, a link failure is a failure of a forwarding
rule in the topology function. On the other hand, if an output link is congested, failure is captured by the latency of a test packet going above a threshold.

4.1 Algorithm: Our algorithm for pinpointing faulty rules assumes that a test packet will succeed only if it succeeds at every hop. For intuition, a ping succeeds only when all the forwarding rules along the path behave correctly. Similarly, if a queue is congested, any packets that travel through it will incur higher latency and may fail an end-to-end test. Formally, we have the following.

Assumption 1 (Fault Propagation) R(pk)=1 if and only if for all r € pk,history, R(r,pk)=1:, ATPG pinpoints a faulty rule by first computing the minimal set of potentially faulty rules. Formally, we have Problem 2.

Problem 2 (Fault Localization): Given a list of $(pk_0, R(pk_0)),(pk_1, R(pk_1)),\ldots$ tuples, find all that satisfies, . $pk_i, R(pk_i,r)=0$.

We solve this problem opportunistically and in steps.

Step 1: Consider the results from sending the regular test packets. For every passing test, place all rules they exercise into a set of passing rules, P.

Step 2: ATPG next trims the set of suspect rules by weeding out correctly working rules. ATPG does this using the reserved packets (the packets eliminated by Min-Set-Cover). ATPG selects reserved packets whose rule histories contain exactly one rule from the suspect set and sends these packets. Suppose a
reserved packet exercises only rule in the suspect set. If the sending of fails, ATPG infers that rule is in error; if passes,is removed from the suspect set. ATPG repeats this process for each reserved packet chosen in Step 2.

Step 3: In most cases, the suspect set is small enough after
Step 2, that ATPG can terminate and report the suspect set. If needed, ATPG can narrow down the suspect set further by sending test packets that exercise two or more of the rules in the
suspect set using the same technique underlying Step 2. If these test packets pass, ATPG infers that none of the exercised rules
are in error and removes these rules from the suspect set. If our Fault Propagation assumption holds, the method will not miss
any faults, and therefore will have no false negatives.

False Positives: Note that the localization method may introduce false positives, rules left in the suspect set at the end of

Step 3. Specifically, one or more rules in the suspect set may in fact behave correctly.

5. USE CASES

We can use ATPG for both functional and performance testing, as the following use cases demonstrate.

Functional Testing

We can test the functional correctness of a network by testing that every reachable forwarding and drop rule in the network is behaving correctly.

Forwarding Rule: A forwarding rule is behaving correctly if a test packet exercises the rule and leaves on the correct port with the correct header.

Upon being tested by making sure a test packet passes correctly over the link without header modifications.

Drop Link Rule: A link rule is a special case of a forwarding rule. It can Rule: Testing drop rules is harder because we must verify the absence of received test packets. We need to know which test packets might reach an egress test terminal if a drop rule was to fail. To find these packets, in the all-pairs reacha-bility analysis, we conceptually "flip" each drop rule to a broad-cast rule in the transfer functions. We do not actually change rules in the switches—we simply emulate the drop rule failure in order to identify all the ways a packet could reach the egress test terminals.

6. IMPLEMENTATION

We implemented a prototype system to automatically parse router configurations and generate a set of test packets for the network.

A. Test Packet Generator

The test packet generator, written in Python, contains a Cisco IOS configuration parser and a Juniper Junos parser. The data-plane information, including router configurations, FIBs, MAC learning tables, and network topologies, is collected and parsed through the command line interface (Cisco IOS) or XML files (Junos). The generator then uses the Hassel header space analysis library to construct switch and topology functions.

All-pairs reachability is computed using the `multiProcess` parallel-processing module shipped with Python. Each process considers a subset of the test ports and finds all the reachable ports from each one.

After reachability tests are complete, re-sults are collected, and the master process executes the Min-Set-Cover algorithm. Test packets and the set of tested rules are stored in a SQLite database.

B. Network Monitor

The network monitor assumes there are special test agents in the network that are able to send/receive test packets. The net-work monitor reads the database and constructs test packets and instructs each agent to send the appropriate packets. Currently, test agents separate test packets by IP Proto field and TCP/UDP port number, but other fields, such as IP option, can also be used. If some of the tests fail, the monitor selects additional test packets from reserved packets to pinpoint the problem. The process repeats until the fault has been identified. The mon-itor uses JSON to communicate with the test agents, and uses SQLite's string matching to lookup test packets efficiently.

C. Alternate Implementations

Our prototype was designed to be minimally invasive, re-quiring no changes to the network except to add terminals at the edge. In networks requiring faster diagnosis, the following extensions are possible.

Cooperative Routers: A new feature could be added to switches/routers, so that a central ATPG system can instruct a router to send/receive test packets. In fact, for manufacturing testing purposes, it is likely that almost every commercial switch/router can already do this; we just need an open inter-face to control them.

SDN -Based Testing: In a software defined network (SDN) such as OpenFlow [27], the controller could directly instruct the switch to send test packets and to detect and forward received test packets to the control plane. For performance testing, test packets need to be time-stamped at the routers.

8. RELATED WORK AND DISCUSSION

A. Overhead and Performance

The principal sources of overhead for ATPG are polling the network periodically for forwarding state and performing all-pairs reachability. While one can reduce overhead by running the offline ATPG calculation less frequently, this runs the risk of using out-of-date forwarding information. Instead, we reduce overhead in two ways. First, we have recently sped up the all-pairs reachability calculation using a fast multithreaded/multimachine header space library.

Second, instead of extracting the complete network state every time ATPG is triggered, an incre-mental state updater can signi ficantly reduce both the retrieval time and the time to calculate reachability. We are working on real-time version of ATPG that incorporates both techniques. Test agents within terminals incur negligible overhead be- cause they merely demultiplex test packets addressed to their IP address at a modest rate (e.g., 1 per millisecond) compared to the link speeds (>1 Gb/s) most modern CPUs are capable of receiving.

B. Limitations

ATPG has the following limitations.

1) Dynamic boxes: ATPG cannot model boxes whose internal state can be changed by test packets. For example, an NAT that dynamically assigns TCP ports to outgoing packets can confuse the online monitor as the same test packet can give different results.

2) Nondeterministic boxes: Boxes can load-balance packets based on a hash function of packet fields, usually combined with a random seed; this is common in multipath routing such as ECMP. When the hash algorithm and parameters are unknown, ATPG cannot properly model such rules. However, if there are known packet patterns that can iterate through all possible outputs, ATPG can generate packets to traverse every output.

3) Invisible rules: A failed rule can make a backup rule active, and as a result, no changes may be observed by the test packets. This can happen when, despite a failure, a test packet is routed to the expected destination by other rules. In addition, an error in a backup rule cannot be detected in normal operation. Another example is when two drop rules appear in a row: The failure of one rule is undetectable since the effect will be masked by the other rule.

4) Transient network states: ATPG cannot uncover errors whose lifetime is shorter than the time between each round of tests. For example, congestion may disappear before an available bandwidth probing test concludes. Finer-grained test agents are needed to capture abnormalities of short duration.

5) Sampling: ATPG uses sampling when generating test packets. As a result, ATPG can miss match faults since the error is not uniform all matching headers. In worst case exhaustive testing is needed.

9. CONCLUSION:

Testing liveness of a network is a fundamental problem for ISPs and large data center operators. Sending probes between every pair of edge ports is neither exhaustive nor scalable ATPG, however, goes much further than liveness testing with the same framework. ATPG can test for reachability policy (by testing all rules including drop rules) and performance health (by associating performance measures such as latency and loss with test packets). Our implementation also augments testing with a simple fault localization scheme also constructed using the header space framework. As in software testing, the formal model helps maximize test coverage while minimizing test packets (Min- Set-Cover). Even the fundamental problem of automatically generating test packets for efficient liveness testing re-quires techniques akin to ATPG.

ATPG, however, goes much further than liveness testing with the same framework. ATPG can test for reachability policy (by testing all rules including drop rules) and performance health (by associating performance measures such as latency and loss with test packets). Our implementation also augments testing with a simple fault localization scheme also constructed using the header space framework. As in software testing, the formal

model helps maximize test coverage while minimizing test packets. Our results show that all forwarding rules in Stanford backbone or Internet2 can be exercised by a surprisingly small number of test packets (<4000 for Stanford, and <40000 for Internet2).

Our survey results indicate that they are eager for more sophisticated tools. Other fields of engineering indi-cate that these desires are not unreasonable: For example, both the ASIC and software design industries are buttressed by billion-dollar tool businesses that supply techniques for both static (e.g., design rule) and dynamic (e.g., timing) verification. In fact, many months after we built and named our system, we dis-covered to our surprise that ATPG was a well-known acronym in hardware chip testing, where it stands for Automatic Test Pat-tern Generation [2]. We hope network ATPG will be equally useful for automated dynamic testing of production networks.

10.REFERENCES:

[1] "ATPG code repository," [Online]. Available: http://eastzone.github. com/atpg/

[2] "Automatic Test Pattern Generation," 2013 [Online]. Available: http://en.wikipedia.org/wiki/Automatic_test_pattern_generation

[3] P. Barford, N. Duffield, A. Ron, and J. Sommers, "Network performance anomaly detection and localization," in Proc. IEEE INFOCOM, Apr. , pp. 1377–1385.

[4] "Beacon," [Online]. Available: http://www.beaconcontroller.net/

[5] Y. Bejerano and R. Rastogi, "Robust monitoring of link delays and faults in IP networks," IEEE/ACM Trans. Netw., vol. 14, no. 5, pp. 1092–1103, Oct. 2006.

[6] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in Proc. OSDI, Berkeley, CA, USA, 2008, pp. 209–224.

[7] M. Canini,D.Venzano, P. Peresini,D.Kostic, and J. Rexford, "A NICE way to test OpenFlow applications," in Proc. NSDI, 2012, pp. 10–10.

[8] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot, "Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data," in Proc. ACM CoNEXT, 2007, pp. 18:1–18:12..

[9] N. Duffield, "Network tomography of binary network performance characteristics," IEEE Trans. Inf. Theory, vol. 52, no. 12, pp. 5373–5388, Dec. 2006.

[10] N. Duffield, F. L. Presti, V. Paxson, and D. Towsley, "Inferring link loss using striped unicast probes," in Proc. IEEE INFOCOM, 2001, vol. 2, pp. 915–923.

[11] N. G. Duffield and M. Grossglauser, "Trajectory sampling for direct traffic observation," IEEE/ACM Trans. Netw., vol. 9, no. 3, pp. 280–292, Jun. 2001.

[12] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in Proc. ACM SIGCOMM, 2011, pp. 350–361.

[13] "Hassel, the Header Space Library," [Online]. Available: https://bitbucket. org/peymank/hassel-public/

[14] Internet2, Ann Arbor, MI, USA, "The Internet2 observatory data collections," [Online]. Available: http://www.internet2.edu/observatory/archive/data-collections.html

[15] M. Jain and C. Dovrolis, "End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput," IEEE/ACM Trans. Netw., vol. 11, no. 4, pp. 537–549, Aug. 2003.

[16] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis:Static checking for networks," in Proc. NSDI, 2012, pp. 9–9.

[17] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren, "IP fault localization via risk modeling," in Proc. NSDI, Berkeley, CA, USA, 2005, vol. 2, pp. 57–70.

[18] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A SOFT way for OpenFlow switch interoperability testing," in Proc. ACM CoNEXT, 2012, pp. 265–276.

[19] K. Lai and M. Baker, "Nettimer: A tool for measuring bottleneck link, bandwidth," in Proc. USITS, Berkeley, CA, USA, 2001, vol. 3, pp. 11–11.

[20] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in Proc. Hotnets, 2010, pp. 19:1–19:6.

[21] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb, "Detecting network-wide and router-specific misconfigurations through data

mining," IEEE/ACM Trans. Netw., vol. 17, no. 1, pp. 66–79, Feb. 2009.
[22] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "iplane: An information plane for distributed services," in Proc. OSDI, Berkeley, CA, USA, 2006, pp. 367–380.
[23] A. Mahimkar, Z. Ge, J. Wang, J. Yates, Y. Zhang, J. Emmons, B. Huntley, and M. Stockert, "Rapid detection of maintenance induced changes in service performance," in Proc. ACM CoNEXT, 2011, pp.
13:1–13:12.
[24] A. Mahimkar, J. Yates, Y. Zhang, A. Shaikh, J.Wang, Z. Ge, and C. T. Ee, "Troubleshooting chronic conditions in large IP networks," in Proc. ACM CoNEXT, 2008, pp. 2:1–2:12.
[25] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with Anteater," Comput. Commun. Rev., vol. 41, no. 4, pp. 290–301, Aug. 2011.
[26] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot, "Characterization of failures in an operational ip backbone network," IEEE/ACM Trans. Netw., vol. 16, no. 4, pp. 749–762, Aug. 2008.
[27] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," Comput. Commun. Rev., vol. 38, pp. 69–74, Mar. 2008.
[28] "OnTimeMeasure," [Online]. Available: http://ontime.oar.net/
[29] "Open vSwitch," [Online]. Available: http://openvswitch.org/
[30] H. Weatherspoon, "All-pairs ping service for PlanetLab ceased," 2005 [Online]. Available: http://lists.planet-lab.org/pipermail/users/2005- July/001518.html
[31] M.Reitblatt,N.Foster, J. Rexford, C. Schlesinger, andD.Walker, "Abstractions for network update," in Proc. ACM SIGCOMM, 2012, pp. 323–334.
[32] S. Shenker, "The future of networking, and the past of protocols," 2011 [Online].Available: http://opennetsummit.org/archives/oct11/shenkertue.pdf
[33] "Troubleshooting the network survey," 2012 [Online]. Available: http://eastzone.github.com/atpg/docs/NetDebugSurvey.pdf
[34] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage, "California fault lines: Understanding the causes and impact of network failures," Comput. Commun. Rev., vol. 41, no. 4, pp. 315–326, Aug. 2010.
[35] P. Yalagandula, P. Sharma, S. Banerjee, S. Basu, and S.-J. Lee, "S3: A scalable sensing service for monitoring large networked systems," in Proc. INM, 2006, pp. 71–76