Automatic Test Packet Generation

Hongyi Zeng, *Member, IEEE*, Peyman Kazemian, *Member, IEEE*, George Varghese, *Member, IEEE, Fellow, ACM*, and Nick McKeown, *Fellow, IEEE, ACM*

***Abstract*—Networks are getting larger and more complex, yet administrators rely on rudimentary tools such as `ping` and `traceroute` to debug problems. We propose an automated and systematic approach for testing and debugging networks 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 (which only localize faults given liveness results). We describe our prototype ATPG implementation and results on two real-world data sets: Stanford University's backbone network and Internet2. We find that a small number of test packets suffices to test all rules in these networks: For example, 4000 packets can cover all rules in Stanford backbone network, while 54 are enough to cover all links. Sending 4000 test packets 10 times per second consumes less than 1% of link capacity. ATPG code and the data sets are publicly available.**

***Index Terms*—Data plane analysis, network troubleshooting, test packet generation.**

I. Introduction

"Only strong trees stand the test of a storm."

—*Chinese idiom*

IT IS notoriously hard to debug networks. Every day, network engineers wrestle with router misconfigurations, fiber cuts, faulty interfaces, mislabeled cables, software bugs, intermittent links, and a myriad other reasons that cause networks to misbehave or fail completely. Network engineers hunt down bugs using the most rudimentary tools (e.g., `ping`, `traceroute`, SNMP, and `tcpdump`) and track down root causes using a combination of accrued wisdom and intuition. Debugging networks is only becoming harder as networks are getting *bigger* (modern data centers may contain 10 000 switches, a campus network may serve 50 000 users, a 100-Gb/s long-haul link may carry 100 000 flows) and are getting *more complicated* (with over 6000 RFCs, router software is based on millions of lines of source code, and network chips often contain billions of gates). It is a mall wonder that network engineers have been labeled "masters of complexity" [32]. 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 `traceroute`.

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 configuration parameters. Second, the forwarding state is hard to observe because it typically requires manually 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.

Fig. 1 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

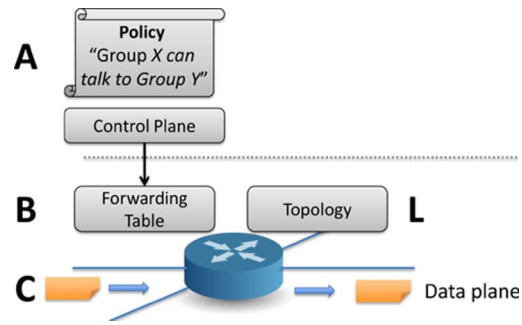


Fig. 1. Static versus dynamic checking: A policy is compiled to forwarding state, which is then executed by the forwarding plane. Static checking (e.g., [16]) confirms that $A = B$. Dynamic checking (e.g., ATPG in this paper) confirms that the topology is meeting liveness properties (L) *and* that $B = C$.

Manuscript received December 26, 2012; accepted March 08, 2013; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor C.-N. Chuah. Date of publication April 11, 2013; date of current version April 14, 2014. This work was supported by the NSF under Grants CNS-0832820, CNS-0855268, and CNS-1040593 and the Stanford Graduate Fellowship.

H. Zeng, P. Kazemian, and N. McKeown are with Stanford University, Stanford, CA 94305 USA (e-mail: hyzeng@stanford.edu; kazemian@stanford.edu; nickm@stanford.edu).

G. Varghese is with the University of California, San Diego, La Jolla, CA 92093 USA, and also with Microsoft Research, Mountain View, CA 94043 USA (e-mail: varghese@cs.ucsd.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2013.2253121

model [32]) and should correctly implement the network administrator'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 network behaves as designed, all three steps should remain consistent 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 *configuration* [7], [16], [25], [31]. 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 (see Table I in Section II) that the two most common causes of network 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 detect these types of failures.

The main contribution of this paper is what we call an Automatic Test Packet Generation (ATPG) framework that *automatically* 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 automatically generate packets to test *performance* assertions such as packet latency. In Example 1, instead of Alice manually deciding which `ping` packets to send, the tool does so periodically on her behalf. In Example 2, 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 exhaustively *testing* all forwarding entries, firewall rules, and any packet processing rules in the network. In ATPG, test packets are generated algorithmically from the device configuration 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 coverage 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 [14] *proactively* check the health of their network using pings between all pairs of sources. However, all-pairs `ping` does not guarantee testing of all links and has been found to be unscalable for large networks such as PlanetLab [30].

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 exercise more critical rules. For example, a healthcare network may dedicate more test packets to Firewall rules to ensure HIPPA compliance.

We tested our method on two real-world data sets—the backbone networks of Stanford University, Stanford, CA, USA, and Internet2, representing an enterprise network and a nationwide ISP. The results are encouraging: Thanks to the structure of real world rulesets, the number of test packets needed is surprisingly small. For the Stanford network with over 757 000 rules and more than 100 VLANs, we only need 4000 packets to exercise all forwarding rules and ACLs. On Internet2, 35 000 packets suffice to exercise all IPv4 forwarding rules. 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 network bandwidth.

The contributions of this paper are as follows:

1) a survey of network operators revealing common failures and root causes (Section II);
2) a test packet generation algorithm (Section IV-A);
3) a fault localization algorithm to isolate faulty devices and rules (Section IV-B);
4) ATPG use cases for functional and performance testing (Section V);
5) evaluation of a prototype ATPG system using rulesets collected from the Stanford and Internet2 backbones (Sections VI and VII).

II. Current Practice

To understand the problems network engineers encounter, and how they currently troubleshoot them, we invited subscribers to the NANOG[1] mailing list to complete a survey in May–June 2012. Of the 61 who responded, 12 administer small networks (< 1 k hosts), 23 medium networks (1 k–10 k hosts), 11 large networks (10 k–100 k hosts), and 12 very large networks (> 100 k hosts). All responses (anonymized) are reported in [33] and are summarized in Table I. The most relevant findings are as follows.

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

[1]North American Network Operators' Group

TABLE I
RANKING OF SYMPTOMS AND CAUSES REPORTED BY ADMINISTRATORS (5 = MOST OFTEN; 1 = LEAST OFTEN). THE RIGHT COLUMN SHOWS THE PERCENTAGE WHO REPORTED $\geq$4. (a) SYMPTOMS OF NETWORK FAILURE. (b) CAUSES OF NETWORK FAILURE

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)

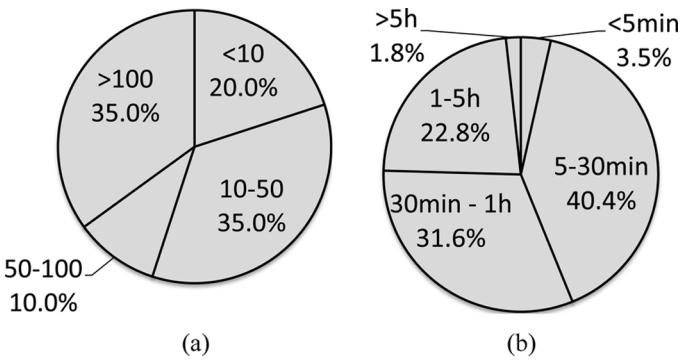


Fig. 2. Reported number of (a) network-related tickets generated per month and (b) time to resolve a ticket.

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 hypothesis 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.

TABLE II
TOOLS USED BY NETWORK ADMINISTRATORS (5 = MOST OFTEN; 1 = LEAST OFTEN)

Category	Avg	% of ≥ 4
`ping`	4.50	86.67%
`traceroute`	4.18	80.00%
SNMP	3.83	60.10%
Configuration Version Control	2.96	37.50%
`netperf/iperf`	2.35	17.31%
sFlow/NetFlow	2.60	26.92%

III. NETWORK MODEL

ATPG uses the *header space* framework—a geometric model of how packets are processed we described in [16] (and used in [31]). In header space, protocol-specific meanings associated with headers are ignored: A header is viewed as a flat sequence of ones and zeros. A header is a point (and a flow is a region) in the $\{0,1\}^L$ space, where L is an upper bound on header length. By using the header space framework, we obtain a unified, vendor-independent, and protocol-agnostic model of the network[2] that simplifies the packet generation process significantly.

A. Definitions

Fig. 3 summarizes the definitions in our model.

Packets: A packet is defined by a $(port, header)$ tuple, where the $port$ denotes a packet's position in the network at any time instant; each physical port in the network is assigned a unique number.

Switches: A *switch transfer function*, T, models a network device, such as a switch or router. Each network device contains a set of forwarding rules (e.g., the forwarding table) that determine how packets are processed. An arriving packet is associated with exactly one rule by matching it against each rule in descending order of priority, and is dropped if no rule matches.

Rules: A *rule* generates a list of one or more output packets, corresponding to the output port(s) to which the packet is sent, and defines how packet fields are modified. The rule abstraction models all real-world rules we know including IP forwarding (modifies port, checksum, and TTL, but not IP address); VLAN tagging (adds VLAN IDs to the header); and ACLs (block a header, or map to a queue). Essentially, a rule defines how a region of header space at the ingress (the set of packets matching the rule) is transformed into regions of header space at the egress [16].

Rule History: At any point, each packet has a *rule history*: an ordered list of rules $[r_0, r_1, \ldots]$ the packet matched so far as it traversed the network. Rule histories are fundamental to ATPG, as they provide the basic raw material from which ATPG constructs tests.

Topology: The *topology transfer function*, Γ, models the network topology by specifying which pairs of ports $(p_{\rm src}, p_{\rm dst})$ are

[2]We have written vendor and protocol-specific parsers to translate configuration files into header space representations.

Bit	$b = 0\|1\|\mathtt{x}$
Header	$h = [b_0, b_1, \ldots, b_L]$
Port	$p = 1\|2\|\ldots\|N\|\mathtt{drop}$
Packet	$pk = (p, h)$
Rule	$r : pk \rightarrow pk$ or $[pk]$
Match	$r.matchset : [pk]$
Transfer Function	$T : pk \rightarrow pk$ or $[pk]$
Topo Function	$\Gamma : (p_{src}, h) \rightarrow (p_{dst}, h)$

(a)

function $T_i(pk)$
  #Iterate according to priority in switch i
  for $r \in ruleset_i$ **do**
    if $pk \in r.matchset$ **then**
      $pk.history \leftarrow pk.history \bigcup \{r\}$
      return $r(pk)$
  return $[(\mathtt{drop}, pk.h)]$

(b)

Fig. 3. Network model: (a) basic types and (b) the switch transfer function.

function NETWORK($packets, switches, \Gamma$)
  for $pk_0 \in packets$ **do**
    $T \leftarrow$FIND_SWITCH($pk_0.p, switches$)
    for $pk_1 \in T(pk_0)$ **do**
      if $pk_1.p \in EdgePorts$ **then**
        #Reached edge
        RECORD(pk_1)
      else
        #Find next hop
        NETWORK($\Gamma(pk_1), switches, \Gamma$)

Fig. 4. Life of a packet: repeating T and Γ until the packet reaches its destination or is dropped.

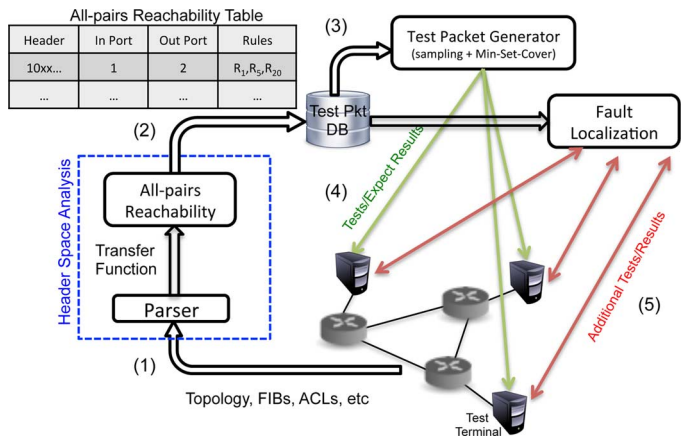


Fig. 5. ATPG system block diagram.

connected by links. Links are rules that forward packets from $p_{\rm src}$ to $p_{\rm dst}$ without modification. If no topology rules match an input port, the port is an edge port, and the packet has reached its destination.

B. Life of a Packet

The life of a packet can be viewed as applying the switch and topology *transfer functions* repeatedly (Fig. 4). When a packet pk arrives at a network port p, the switch function T that contains the input port $pk.p$ is applied to pk, producing a list of new packets $[pk_1, pk_2, \ldots]$. If the packet reaches its destination, it is recorded. Otherwise, the topology function Γ is used to invoke the switch function containing the new port. The process repeats until packets reach their destinations (or are dropped).

IV. ATPG System

Based on the network model, ATPG generates the minimal number of test packets so that every forwarding rule in the network is exercised and covered by at least one test packet. When an error is detected, ATPG uses a fault localization algorithm to determine the failing rules or links.

Fig. 5 is a block diagram of the ATPG system. The system first collects all the forwarding state from the network (step 1). This usually involves reading the FIBs, ACLs, and config files, as well as obtaining the topology. ATPG uses Header Space Analysis [16] to compute reachability between all the test terminals (step 2). The result is then used by the test packet selection algorithm to compute a minimal set of test packets that can test all rules (step 3). These packets will be sent periodically by the test terminals (step 4). If an error is detected, the fault localization algorithm is invoked to narrow down the cause of the error (step 5). While steps 1 and 2 are described in [16], steps 3–5 are new.

A. Test Packet Generation

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:* ATPG must 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. Formally, we have the following problem.

Problem 1 (Test Packet Selection): For a network with the switch functions, $\{T_1, \ldots, T_n\}$, 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

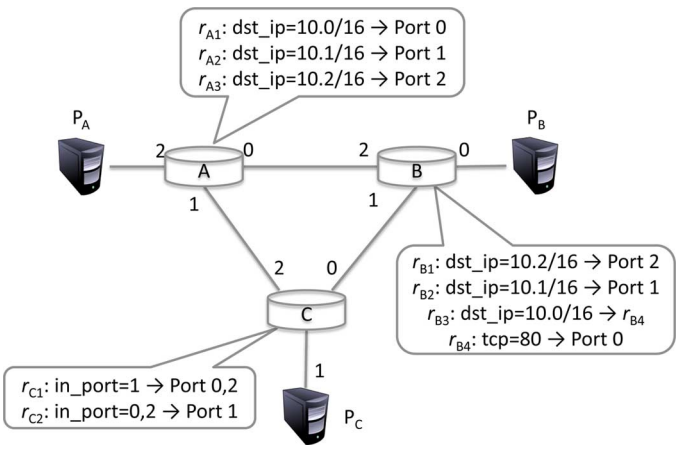


Fig. 6. Example topology with three switches.

TABLE III
ALL-PAIRS REACHABILITY TABLE: ALL POSSIBLE HEADERS FROM EVERY TERMINAL TO EVERY OTHER TERMINAL, ALONG WITH THE RULES THEY EXERCISE

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}	$[r_{n1}, r_{n2}, \ldots]$

then *samples* each class to choose test packets, and finally *compresses* the resulting set of test packets to find the minimum covering set.

Step 1: Generate All-Pairs Reachability Table: ATPG starts by computing the complete set of packet headers that can be sent from each test terminal to every other test terminal. For each such header, ATPG finds the complete set of rules it exercises along the path. To do so, ATPG applies the all-pairs reachability algorithm described in [16]: On every terminal port, an all-x header (a header that has all wildcarded bits) is applied to the transfer function of the first switch connected to each test terminal. Header constraints are applied here. For example, if traffic can only be sent on VLAN A, then instead of starting with an all-x header, the VLAN tag bits are set to A. As each packet pk traverses the network using the network function, the set of rules that match pk are recorded in $pk.history$. Doing this for all pairs of terminal ports generates an *all-pairs reachability table* as shown in Table III. For each row, the header column is a wildcard expression representing the equivalent class of packets that can reach an egress terminal from an ingress test terminal. All packets matching this class of headers will encounter the set of switch rules shown in the Rule History column.

Fig. 6 shows a simple example network, and Table IV is the corresponding all-pairs reachability table. For example, an all-x test packet injected at P_A will pass through switch A. A forwards packets with $dst\_ip = 10.0/16$ to B and those with $dst\_ip = 10.1/16$ to C. B then forwards $dst\_ip = 10.0/16$, $tcp = 80$ to P_B, and switch C forwards $dst\_ip = 10.1/16$ to P_C. These are reflected in the first two rows of Table IV.

Step 2: Sampling: Next, ATPG picks at least one test packet in an equivalence class to exercise every (reachable) rule. The simplest scheme is to randomly pick one packet per class. This scheme only detects faults for which all packets covered by the same rule experience the same fault (e.g., a link failure). At the other extreme, if we wish to detect faults specific to a header, then we need to select every header in every class. We discuss these issues and our fault model in Section IV-B.

Step 3: Compression: Several of the test packets picked in Step 2 exercise the same rule. ATPG therefore selects a minimum subset of the packets chosen in Step 2 such that the union of their rule histories cover all rules. The cover can be chosen to cover all links (for liveness only) or all router queues (for performance only). This is the classical Min-Set-Cover problem. While NP-Hard, a greedy $O(N^2)$ algorithm provides a good approximation, where N is the number of test packets. We call the resulting (approximately) minimum set of packets, the *regular test packets*. The remaining test packets not picked for the minimum set are called the *reserved test packets*. In Table IV, $\{p_1, p_2, p_3, p_4, p_5\}$ are regular test packets, and $\{p_6\}$ is a reserved test packet. Reserved test packets are useful for fault localization (Section IV-B).

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 T 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.

B. 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* R. For a rule r, the result function is defined as

$$R(r, pk) = \begin{cases} 0, & \text{if } pk \text{ fails at rule } r \\ 1, & \text{if } pk \text{ succeeds at rule } r. \end{cases}$$

TABLE IV
TEST PACKETS FOR THE EXAMPLE NETWORK DEPICTED IN FIG. 6. p_6 IS STORED AS A RESERVED PACKET

	Header	Ingress Port	Egress Port	Rule History
p_1	`dst_ip=10.0/16, tcp=80`	P_A	P_B	r_{A1}, r_{B3}, r_{B4}, link AB
p_2	`dst_ip=10.1/16`	P_A	P_C	r_{A2}, r_{C2}, link AC
p_3	`dst_ip=10.2/16`	P_B	P_A	r_{B2}, r_{A3}, link AB
p_4	`dst_ip=10.1/16`	P_B	P_C	r_{B2}, r_{C2}, link BC
p_5	`dst_ip=10.2/16`	P_C	P_A	r_{C1}, r_{A3}, link BC
(p_6)	`dst_ip=10.2/16, tcp=80`	P_C	P_B	r_{C1}, r_{B3}, r_{B4}, link BC

"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.

We divide faults into two categories: *action faults* and *match faults*. An action fault occurs when *every* packet matching the rule is processed incorrectly. Examples of action faults include unexpected packet loss, a missing rule, congestion, and miswiring. On the other hand, match faults are harder to detect because they only affect *some* packets matching the rule: for example, when a rule matches a header it should not, or when a rule misses a header it should match. Match faults can only be detected by more exhaustive sampling such that at least one test packet exercises each faulty region. For example, if a TCAM bit is supposed to be `x`, but is "stuck at 1," then all packets with a 0 in the corresponding position will not match correctly. Detecting this error requires at least two packets to exercise the rule: one with a 1 in this position, and the other with a 0.

We will only consider action faults because they cover most likely failure conditions and can be detected using only one test packet per rule. We leave match faults for future work.

We can typically only observe a packet at the edge of the network after it has been processed by every matching rule. Therefore, we define an end-to-end version of the result function

$$R(pk) = \begin{cases} 0, & \text{if } pk \text{ fails} \\ 1, & \text{if } pk \text{ succeeds.} \end{cases}$$

2) 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 $\forall r \in pk.history, R(r, pk) = 1$

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 r that satisfies $\exists pk_i,\ R(pk_i, r) = 0$.

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. Similarly, for every failing test, place all rules they exercise into a set of potentially failing rules F. By our assumption, one or more of the rules in F are in error. Therefore, $F - P$ is a set of *suspect rules*.

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 p exercises only rule r in the suspect set. If the sending of p fails, ATPG infers that rule r is in error; if p passes, r 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.

False positives are unavoidable in some cases. When two rules are in series and there is no path to exercise only one of them, we say the rules are *indistinguishable*; any packet that exercises one rule will also exercise the other. Hence, if only one rule fails, we cannot tell which one. For example, if an ACL rule is followed immediately by a forwarding rule that matches the same header, the two rules are indistinguishable. Observe that if we have test terminals before and after each rule (impractical in many cases), with sufficient test packets, we can distinguish every rule. Thus, the deployment of test terminals not only affects test coverage, but also localization accuracy.

V. USE CASES

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

A. 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.

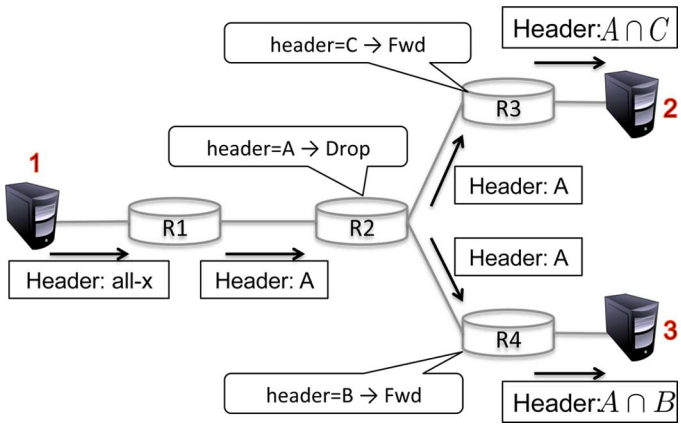


Fig. 7. Generate packets to test drop rules: "flip" the rule to a broadcast rule in the analysis.

Link Rule: A link rule is a special case of a forwarding rule. It can be tested by making sure a test packet passes correctly over the link without header modifications.

Drop 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 reachability analysis, we conceptually "flip" each *drop* rule to a *broadcast* 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.

As an example, consider Fig. 7. To test the drop rule in $R2$, we inject the all-x test packet at Terminal 1. If the drop rule was instead a broadcast rule, it would forward the packet to all of its output ports, and the test packets would reach Terminals 2 and 3. Now, we sample the resulting equivalent classes as usual: We pick one sample test packet from $A \cap B$ and one from $A \cap C$. Note that we have to test *both* $A \cap B$ and $A \cap C$ because the drop rule may have failed at $R2$, resulting in an unexpected packet to be received at either test terminal 2 $(A \cap C)$ or test terminal 3 $(A \cap B)$. Finally, we send and expect the two test packets *not* to appear since their arrival would indicate a failure of $R2$'s drop rule.

B. Performance Testing

We can also use ATPG to monitor the performance of links, queues, and QoS classes in the network, and even monitor SLAs.

Congestion: If a queue is congested, packets will experience longer queuing delays. This can be considered as a (performance) fault. ATPG lets us generate one way congestion tests to measure the latency between every pair of test terminals; once the latency passed a threshold, fault localization will pinpoint the congested queue, as with regular faults. With appropriate headers, we can test links or queues as in Alice's second problem.

Available Bandwidth: Similarly, we can measure the available bandwidth of a link, or for a particular service class. ATPG will generate the test packet headers needed to test every link, or every queue, or every service class; a stream of packets with these headers can then be used to measure bandwidth. One can use destructive tests, like `iperf/netperf`, or more gentle approaches like packet pairs and packet trains [19]. Suppose a manager specifies that the available bandwidth of a particular service class should not fall below a certain threshold; if it does happen, ATPG's fault localization algorithm can be used to triangulate and pinpoint the problematic switch/queue.

Strict Priorities: Likewise, ATPG can be used to determine if two queues, or service classes, are in different strict priority classes. If they are, then packets sent using the lower-priority class should never affect the available bandwidth or latency of packets in the higher-priority class. We can verify the relative priority by generating packet headers to congest the lower class and verifying that the latency and available bandwidth of the higher class is unaffected. If it is, fault localization can be used to pinpoint the problem.

VI. Implementation

We implemented a prototype system to automatically parse router configurations and generate a set of test packets for the network. The code is publicly available [1].

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 [13] 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, results 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 network 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 monitor 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, requiring 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

TABLE V
TEST PACKET GENERATION RESULTS FOR STANFORD BACKBONE (TOP) AND INTERNET2 (BOTTOM), AGAINST THE NUMBER OF PORTS SELECTED FOR DEPLOYING TEST TERMINALS. "TIME TO SEND" PACKETS IS CALCULATED ON A PER-PORT BASIS, ASSUMING 100 B PER TEST PACKET, 1 Gb/s LINK FOR STANFORD, AND 10 Gb/s FOR INTERNET2

Stanford (298 ports)	10%	40%	70%	100%	Edge (81%)
Total Packets	10,042	104,236	413,158	621,402	438,686
Regular Packets	725	2,613	3,627	3,871	3,319
Packets/Port (Avg)	25.00	18.98	17.43	12.99	18.02
Packets/Port (Max)	206	579	874	907	792
Time to send (Max)	0.165ms	0.463ms	0.699ms	0.726ms	0.634ms
Coverage	22.2%	57.7%	81.4%	100%	78.5%
Computation Time	152.53s	603.02s	2,363.67s	3,524.62s	2,807.01s
Internet2 (345 ports)	10%	40%	70%	100%	Edge (92%)
Total Packets	30,387	485,592	1,407,895	3,037,335	3,036,948
Regular Packets	5,930	17,800	32,352	35,462	35,416
Packets/Port (Avg)	159.0	129.0	134.2	102.8	102.7
Packets/Port (Max)	2,550	3,421	2,445	4,557	3,492
Time to send (Max)	0.204ms	0.274ms	0.196ms	0.365ms	0.279ms
Coverage	16.9%	51.4%	80.3%	100%	100%
Computation Time	129.14s	582.28s	1,197.07s	2,173.79s	1,992.52s

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 interface 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.

VII. EVALUATION

A. Data Sets: Stanford and Internet2

We evaluated our prototype system on two sets of network configurations: the Stanford University backbone and the Internet2 backbone, representing a mid-size enterprise network and a nationwide backbone network, respectively.[3]

Stanford Backbone: With a population of over 15 000 students, 2000 faculty, and five/16 IPv4 subnets, Stanford represents a large enterprise network. There are 14 operational zone (OZ) Cisco routers connected via 10 Ethernet switches to two backbone Cisco routers that in turn connect Stanford to the outside world. Overall, the network has more than 757 000 forwarding entries and 1500 ACL rules. Data plane configurations are collected through command line interfaces. Stanford has made the entire configuration rule set public [1].

Internet2: Internet2 is a nationwide backbone network with nine Juniper T1600 routers and 100-Gb/s interfaces, supporting over 66 000 institutions in US. There are about 100 000 IPv4 forwarding rules. All Internet2 configurations and FIBs of the core routers are publicly available [14], with the exception of ACL rules, which are removed for security concerns. Although IPv6 and MPLS entries are also available, we only use IPv4 rules in this paper.

B. Test Packet Generation

We ran ATPG on a quad-core Intel Core i7 CPU 3.2 GHz and 6 GB memory using eight threads. For a given number of test terminals, we generated the minimum set of test packets needed to test all the reachable rules in the Stanford and Internet2 backbones. Table V shows the number of test packets needed. For example, the first column tells us that if we attach test terminals to 10% of the ports, then all of the reachable Stanford rules (22.2% of the total) can be tested by sending 725 test packets. If every edge port can act as a test terminal, 100% of the Stanford rules can be tested by sending just 3,871 test packets. The "Time" row indicates how long it took ATPG to run; the worst case took about an hour, the bulk of which was devoted to calculating all-pairs reachability.

To put these results into perspective, each test for the Stanford backbone requires sending about 907 packets per port in the worst case. If these packets were sent over a single 1-Gb/s link, the entire network could be tested in less than 1 ms, assuming each test packet is 100 B and not considering the propagation delay. Put another way, testing the entire set of forwarding rules 10 times every second would use less than 1% of the link bandwidth.

Similarly, all the forwarding rules in Internet2 can be tested using 4557 test packets per port in the worst case. Even if the test packets were sent over 10-Gb/s links, all the forwarding rules could be tested in less than 0.5 ms, or 10 times every second using less than 1% of the link bandwidth.

We also found that 100% *link* coverage (instead of *rule* coverage) only needed 54 packets for Stanford and 20 for Internet2. The table also shows the large benefit gained by compressing the number of test packets—in most cases, the total number of test packets is reduced by a factor of 20–100 using the minimum set cover algorithm. This compression may make proactive link testing (that was considered infeasible earlier [30]) feasible for large networks.

Coverage is the ratio of the number of rules exercised to the total number of reachable rules. Our results shows that the coverage grows linearly with the number of test terminals available. While it is theoretically possible to optimize the placement of test terminals to achieve higher coverage, we find that the benefit is marginal for real data sets.

Rule Structure: The reason we need so few test packets is because of the structure of the rules and the routing policy. Most

[3]Each figure/table in Section VII (electronic version) is clickable, linking to instructions on reproducing results.

TABLE VI
TEST PACKETS USED IN THE FUNCTIONAL TESTING EXAMPLE. IN THE RULE HISTORY COLUMN, R IS THE IP FORWARDING RULE, L IS A LINK RULE, AND S IS THE BROADCAST RULE OF SWITCHES. R_1 IS THE IP FORWARDING RULE MATCHING ON 172.20.10.32/27, AND R_2 MATCHES ON 171.67.222.64/27. L_b^e IN THE LINK RULE FROM NODE b TO NODE e. THE TABLE HIGHLIGHTS THE COMMON RULES BETWEEN THE PASSED TEST PACKETS AND THE FAILED ONE. IT IS OBVIOUS FROM THE RESULTS THAT RULE R_1^{boza} IS IN ERROR

Header	Ingress	Egress	Rule History	Result
$dst\_ip = 172.20.10.33$	$goza$	$coza$	$[R_1^{goza}, L_{goza}^{S_1}, S_1, L_{S_1}^{bbra}, R_1^{bbra}, L_{bbra}^{S_4}, S_4, R_1^{coza}]$	Pass
$dst\_ip = 172.20.10.33$	$boza$	$coza$	$[R_1^{boza}, L_{boza}^{S_5}, S_5, L_{S_5}^{bbra}, R_1^{bbra}, L_{bbra}^{S_4}, S_4, R_1^{coza}]$	Fail
$dst\_ip = 171.67.222.65$	$boza$	$poza$	$[R_2^{boza}, L_{boza}^{S_5}, S_5, L_{S_5}^{bbra}, R_2^{bbra}, L_{bbra}^{S_2}, S_2, R_2^{poza}]$	Pass

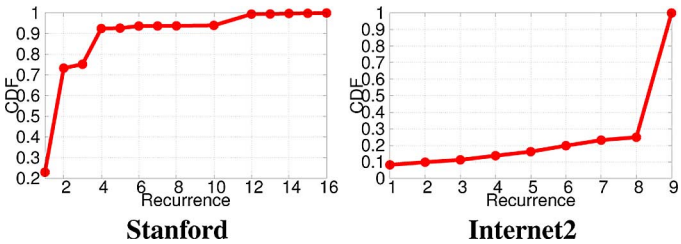


Fig. 8. Cumulative distribution function of rule repetition, ignoring different action fields.

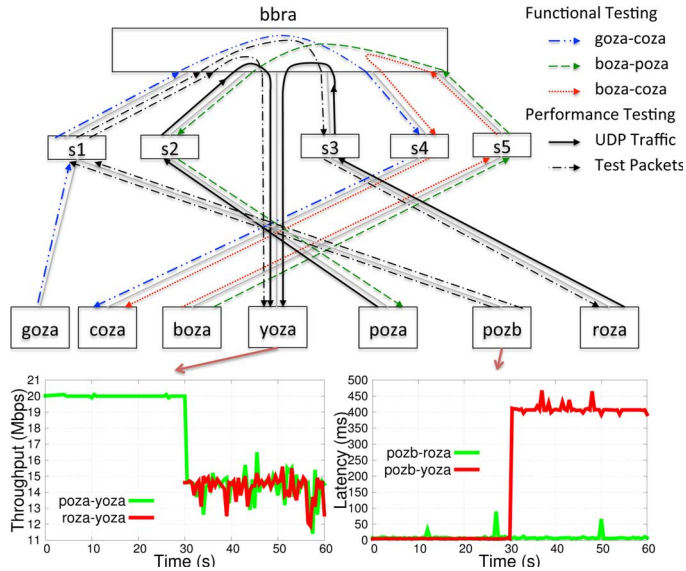


Fig. 9. Portion of the Stanford backbone network showing the test packets used for functional and performance testing examples in Section VII-C.

rules are part of an end-to-end route, and so multiple routers contain the same rule. Similarly, multiple devices contain the same ACL or QoS configuration because they are part of a network-wide policy. Therefore, the number of distinct regions of header space grow linearly, not exponentially, with the diameter of the network.

We can verify this structure by clustering rules in Stanford and Internet2 that match the same header patterns. Fig. 8 shows the distribution of rule repetition in Stanford and Internet2. In both networks, 60%–70% of matching patterns appear in more than one router. We also find that this repetition is correlated to the network topology. In the Stanford backbone, which has a two-level hierarchy, matching patterns commonly appear in two (50.3%) or four (17.3%) routers, which represents the length of edge-to-Internet and edge-to-edge routes. In Internet2, 75.1% of all distinct rules are replicated nine times, which is the number of routers in the topology.

C. Testing in an Emulated Network

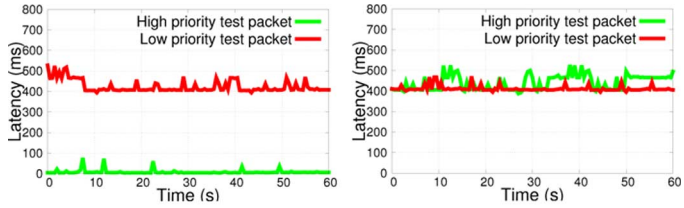
To evaluate the network monitor and test agents, we replicated the Stanford backbone network in Mininet [20], a container-based network emulator. We used Open vSwitch (OVS) [29] to emulate the routers, using the real port configuration information, and connected them according to the real topology. We then translated the forwarding entries in the Stanford backbone network into equivalent OpenFlow [27] rules and installed them in the OVS switches with Beacon [4]. We used emulated hosts to send and receive test packets generated by ATPG. Fig. 9 shows the part of network that is used for experiments in this section. We now present different test scenarios and the corresponding results.

Forwarding Error: To emulate a functional error, we deliberately created a fault by replacing the action of an IP *forwarding* rule in $boza$ that matched $dst\_ip = 172.20.10.32/27$ with a *drop* action (we called this rule R_1^{boza}). As a result of this fault, test packets from $boza$ to $coza$ with $dst\_ip = 172.20.10.33$ failed and were not received at $coza$. Table VI shows two other test packets we used to localize and pinpoint the fault. These test packets shown in Fig. 9 in $goza - coza$ and $boza - poza$ are received correctly at the end terminals. From the *rule history* of the passing and failing packets in Table III, we deduce that only rule R_1^{boza} could possibly have caused the problem, as all the other rules appear in the rule history of a received test packet.

Congestion: We detect congestion by measuring the one-way latency of test packets. In our emulation environment, all terminals are synchronized to the host's clock so the latency can be calculated with a single time-stamp and one-way communication.[4]

To create congestion, we rate-limited all the links in the emulated Stanford network to 30 Mb/s and created two 20-Mb/s UDP flows: $poza$ to $yoza$ at $t = 0$ and $roza$ to $yoza$ at $t = 30$ s, which will congest the link $bbra - yoza$ starting at $t = 30$ s. The bottom left graph next to $yoza$ in Fig. 9 shows the two UDP flows. The queue inside the routers will build up, and test packets will experience longer queuing delay. The bottom right graph next to $pozb$ shows the latency experienced by two test packets, one from $pozb$ to $roza$ and the other one from $pozb$ to $yoza$. At $t = 30$ s, the $bozb - yoza$ test packet experiences much higher latency, correctly signaling congestion. Since these two test packets share the $bozb - s_1$ and $s_1 - bbra$ links, ATPG concludes that the congestion is not happening in these two links;

[4]To measure latency in a real network, two-way communication is usually necessary. However, relative change of latency is sufficient to uncover congestion.



Congested Slice	Terminal	Result/Mbps
High	High	11.95
	Low	11.66
Low	High	23.22
	Low	11.78

(c)

Fig. 10. Priority testing: Latency measured by test agents when (a) low- or (b) high-priority slice is congested. (c) Available bandwidth measurements when the bottleneck is in low/high-priority slices.

hence, ATPG correctly infers that $bbra - yoza$ is the congested link.

Available Bandwidth: ATPG can also be used to monitor available bandwidth. For this experiment, we used Pathload [15], a bandwidth probing tool based on packet pairs/packet trains. We repeated the previous experiment, but decreased the two UDP flows to 10 Mb/s, so that the bottleneck available bandwidth was 10 Mb/s. Pathload reports that $bozb - yoza$ has an available bandwidth[5] of 11.715 Mb/s, $bozb - roza$ has an available bandwidth of 19.935 Mb/s, while the other (idle) terminals report 30.60 Mb/s. Using the same argument as before, ATPG can conclude that $bbra - yoza$ link is the bottleneck link with around 10 Mb/s of available bandwidth.

Priority: We created priority queues in OVS using Linux's `htb` scheduler and `tc` utilities. We replicated the previously "failed" test case $pozb - yoza$ for high- and low-priority queues, respectively.[6] Fig. 10 shows the result.

We first repeated the congestion experiment. When the low-priority queue is congested (i.e., both UDP flows mapped to low-priority queues), only low-priority test packets are affected. However, when the high-priority slice is congested, low- and high-priority test packets experience the congestion and are delayed. Similarly, when repeating the available bandwidth experiment, high-priority flows receive the same available bandwidth whether we use high- or low-priority test packets. However, for low-priority flows, the high-priority test packets correctly receive the full link bandwidth.

D. Testing in a Production Network

We deployed an experimental ATPG system in three buildings in Stanford University that host the Computer Science and Electrical Engineering departments. The production network consists of over 30 Ethernet switches and a Cisco router connecting to the campus backbone. For test terminals, we utilized

[5]All numbers are the average of 10 repeated measurements.

[6]The Stanford data set does not include the priority settings.

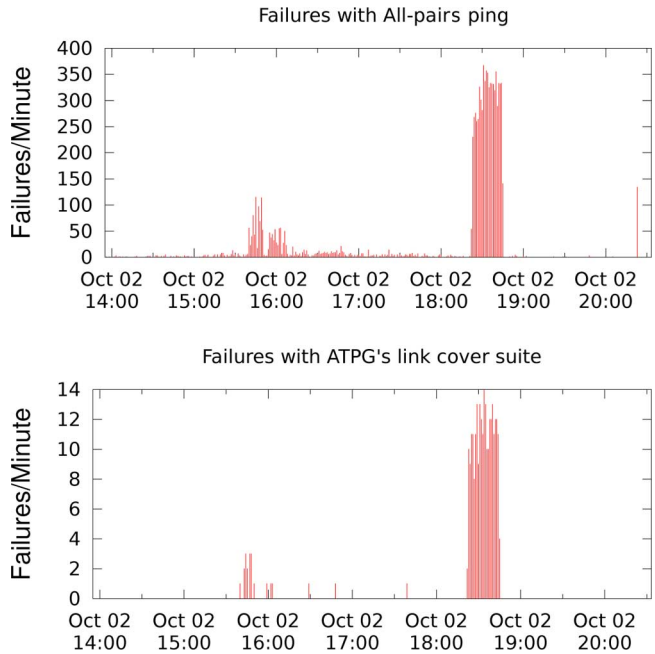


Fig. 11. October 2, 2012 production network outages captured by the ATPG system as seen from the lens of *(top)* an inefficient cover (all-pairs) and *(bottom)* an efficient minimum cover. Two outages occurred at 4 PM and 6:30 PM, respectively.

the 53 WiFi access points (running Linux) that were already installed throughout the buildings. This allowed us to achieve high coverage on switches and links. However, we could only run ATPG on essentially a Layer-2 (bridged) Network.

On October 1–10, 2012, the ATPG system was used for a 10-day `ping` experiment. Since the network configurations remained static during this period, instead of reading the configuration from the switches dynamically, we derived the network model based on the topology. In other words, for a Layer-2 bridged network, it is easy to infer the forwarding entry in each switch for each MAC address without getting access to the forwarding tables in all 30 switches. We only used `ping` to generate test packets. Pings suffice because in the subnetwork we tested there are no Layer-3 rules or ACLs. Each test agent downloads a list of `ping` targets from a central Web server every 10 min and conducts `ping` tests every 10 s. Test results were logged locally as files and collected daily for analysis.

During the experiment, a major network outage occurred on October 2. Fig. 11 shows the number of failed test cases during that period. While both all-pairs `ping` and ATPG's selected test suite correctly captured the outage, ATPG uses significantly less test packets. In fact, ATPG uses only 28 test packets per round compared to 2756 packets in all-pairs `ping`, a 100× reduction. It is easy to see that the reduction is from quadratic overhead (for all-pairs testing between 53 terminals) to linear overhead (for a set cover of the 30 links between switches). We note that while the set cover in this experiment is so simple that it could be computed by hand, other networks will have Layer-3 rules and more complex topologies requiring the ATPG minimum set cover algorithm.

The network managers confirmed that the later outage was caused by a loop that was accidentally created during switch testing. This caused several links to fail, and hence more than

300 pings failed per minute. The managers were unable to determine why the first failure occurred. Despite this lack of understanding of the root cause, we emphasize that the ATPG system correctly detected the outage in both cases and pinpointed the affected links and switches.

VIII. 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 *incremental* state updater can significantly reduce both the retrieval time and the time to calculate reachability. We are working on a real-time version of ATPG that incorporates both techniques.

Test agents within terminals incur negligible overhead because 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

As with all testing methodologies, ATPG has 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 across all matching headers. In the worst case (when only one header is in error), exhaustive testing is needed.

IX. Related Work

We are unaware of earlier techniques that automatically generate test packets from configurations. The closest related works we know of are offline tools that check invariants in networks. In the control plane, NICE [7] attempts to exhaustively cover the code paths symbolically in controller applications with the help of simplified switch/host models. In the data plane, Anteater [25] models invariants as boolean satisfiability problems and checks them against configurations with a SAT solver. Header Space Analysis [16] uses a geometric model to check reachability, detect loops, and verify slicing. Recently, SOFT [18] was proposed to verify consistency between different OpenFlow agent implementations that are responsible for bridging control and data planes in the SDN context. ATPG complements these checkers by directly *testing* the data plane and covering a significant set of dynamic or performance errors that cannot otherwise be captured.

End-to-end probes have long been used in network fault diagnosis in work such as [8]–[10], [17], [23], [24], [26]. Recently, mining low-quality, unstructured data, such as router configurations and network tickets, has attracted interest [12], [21], [34]. By contrast, the primary contribution of ATPG is not fault localization, but determining a compact set of end-to-end measurements that can cover every rule or every link. The mapping between Min-Set-Cover and network monitoring has been previously explored in [3] and [5]. ATPG improves the detection granuality to the rule level by employing router configuration and data plane information. Furthermore, ATPG is not limited to liveness testing, but can be applied to checking higher level properties such as performance.

There are many proposals to develop a measurement-friendly architecture for networks [11], [22], [28], [35]. Our approach is complementary to these proposals: By incorporating input and port constraints, ATPG can generate test packets and injection points using existing deployment of measurement devices.

Our work is closely related to work in programming languages and symbolic debugging. We made a preliminary attempt to use KLEE [6] and found it to be 10 times slower than even the unoptimized header space framework. We speculate that this is fundamentally because in our framework we directly *simulate* the forward path of a packet instead of *solving constraints* using an SMT solver. However, more work is required to understand the differences and potential opportunities.

X. 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 [30]. 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 reachability), and finally determining a minimum set of test packets

(Min-Set-Cover). Even the fundamental problem of automatically generating test packets for efficient liveness testing requires 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 $<40\,000$ for Internet2).

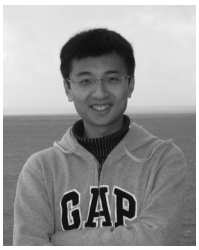
Network managers today use primitive tools such as `ping` and `traceroute`. Our survey results indicate that they are eager for more sophisticated tools. Other fields of engineering indicate that these desires are not unreasonable: For example, both the ASIC and software design industries are buttressed 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 discovered to our surprise that ATPG was a well-known acronym in hardware chip testing, where it stands for Automatic Test *Pattern* Generation [2]. We hope network ATPG will be equally useful for automated dynamic testing of production networks.

ACKNOWLEDGMENT

The authors would like to thank D. Kostic and the anonymous reviewers for their valuable comments. They thank J. van Reijendam, C. M. Orgish, J. Little (Stanford University) and Thomas C. Knoeller and Matthew P. Davy (Internet2) for providing router configuration sets and sharing their operation experience.

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, and D. 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/shenker-tue.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.



Hongyi Zeng (M'08) received the Bachelor's degree in electronic engineering from Tsinghua University, Beijing, China, in 2008, and the Master's degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2010, and is currently pursuing the Ph.D. degree in electrical engineering at Stanford University.

His research focuses on formal network testing, software-defined network, high-performance router architecture, and programmable hardware.

Peyman Kazemian (M'09) received the B.Sc. degree in electrical engineering from Sharif University of Technology, Tehran, Iran, in 2007, and will receive the Ph.D. degree from Stanford University, Stanford, CA, USA, in 2013.

His research interests include network debugging and troubleshooting and software defined networking.

Mr. Kazemian was awarded a Stanford Graduate Fellowship (SGF), a Numerical Technologies Founders Fellowship, and a Maitra-Luther Fellowship while studying at Stanford.

George Varghese (M'94) received the Ph.D. degree in computer science from the Massachusetts Institute of Technology (MIT), Cambridge, MA, USA, in 1992.

He worked from 1993 to 1999 with Washington University, St. Louis, MO, USA, and from 1999 to 2012 with the University of California, San Diego (UCSD), La Jolla, CA, USA, both as a Professor of computer science. In 2004, he cofounded NetSift, Inc., which was acquired by Cisco Systems in 2005. He joined Microsoft Research, Mountain View, CA, USA, in 2012. He wrote a book on fast router and end-node implementations called *Network Algorithmics* (Morgan-Kaufman, 2004).

Dr. Varghese was elected to be a Fellow of the Association for Computing Machinery (ACM) in 2002. He won the ONR Young Investigator Award in 1996.

Nick McKeown (S'91–M'95–SM'97–F'05) received the B.E. degree from the University of Leeds, Leeds, U.K., in 1986, and the M.S. and Ph.D. degrees from the University of California, Berkeley, CA, USA, in 1992 and 1995, respectively, all in electrical engineering and computer science.

He is the Kleiner Perkins, Mayfield and Sequoia Professor of Electrical Engineering and Computer Science with Stanford University, Stanford, CA, USA, and Faculty Director of the Open Networking Research Center. His current research interests include software defined networks (SDNs), how to enable more rapid improvements to the Internet infrastructure, troubleshooting, and formal verification of networks.

Prof. McKeown is a member of the US National Academy of Engineering (NAE), a Fellow of the Royal Academy of Engineering (UK), and a Fellow of the Association for Computing Machinery (ACM).