Automatic Test Packet Generation

Hongyi Zeng[†‡],Peyman Kazemian[†‡], George Varghese[*], Nick McKeown[†]
[†] {hyzeng,kazemian,nickm}@stanford.edu, *Stanford University, Stanford, CA, USA*
[*] varghese@cs.ucsd.edu, *UCSD, La Jolla and Microsoft Research, Mountain View, CA, USA*
[‡] These authors contributed equally to this work

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 is enough to cover all links. Sending 4000 test packets 10 times per second consumes lesss than 1% of link capacity. ATPG code and the data sets are publicly available[1] [1].

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operation—*Network monitoring*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Algorithm, Reliability

[1]Each figure/table in Section 7 (electronic version) is clickable, linking to instructions on reproducing results.

Co-NEXT'12, December 10-13, 2012, Nice, France.

Keywords

Test Packet Generation, Data Plane Analysis, Network Troubleshooting

1. 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, mis-labeled 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 100Gb/s long haul link may carry 100,000 flows), and are getting *more complicated* (with over 6,000 RFCs, router software is based on millions of lines of source code, and network chips often contain billions of gates). Small wonder that network engineers have been labeled "masters of complexity" [28]. 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 frustrated 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

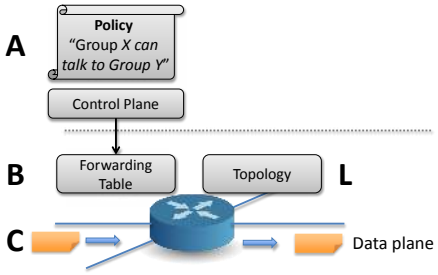


Figure 1: Static versus Dynamic Checking: A policy is compiled to forwarding state, which is then executed by the forwarding plane. Static checking (*e.g.* [12]) confirms that $A = B$. Dynamic checking (e.g. ATPG in this paper) confirms that the topology is meeting liveness properties (L) *and* that $B = C$.

Alice uses `ping` and `traceroute`, she is using a crude lens to examine the current forwarding state for clues to track down the failure.

Figure 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 model [28]), 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 1Mb/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* [3, 12, 21, 27]. 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 say 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 1 in Section 2) 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 latencies. 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 [10] *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 it will be non-optimal and unscalable for large networks such as PlanetLab [26].

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 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 4,000 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 ten 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 msec using 1% of network bandwidth.

The main contributions of our work are: (1) A survey of network operators revealing common network failures and their root cause (Section 2), (2) the test packet generation algorithm (Section 4.1), (3) A fault localization algorithm to analyze results and isolate the faulty device and rule (Section 4.2), (4) use cases of ATPG framework for functional and performance testing (Section 5), (5) Evaluation of a prototype ATPG system using rulesets collected from the Stanford University and Internet2 backbones (Section 6 and Section 7).

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) Symptoms of network failure.

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) Causes of network failure.

Table 1: Ranking of symptoms and causes reported by administrators (5=most often, 1=least often). The right column shows the percentage who reported ≥ 4.

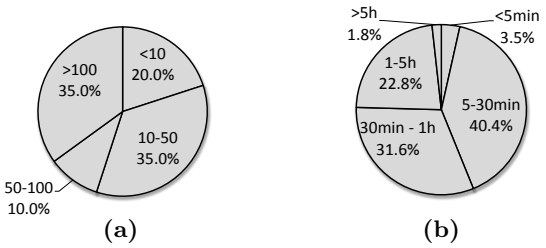


Figure 2: Reported number of network related tickets generated per month (a) and time to resolve a ticket (b).

2. CURRENT PRACTICE

To understand the problems network engineers encounter, and how they currently troubleshoot them, we invited subscribers to the NANOG[2] mailing list to complete a survey in May-June 2012. Of the 61 who responded, 12 administer small networks (<1k hosts), 23 medium networks (1k-10k hosts), 11 large networks (10k-100k hosts) and 12 very large networks (>100k hosts). All responses (anonymized) are reported in [29], and are summarized in Table 1. The most relevant findings are:

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.

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 (Figure 2). 35% of networks generate more than 100 tickets per month. 40.4% of respondents estimate it takes under 30 minutes to resolve a ticket. But 24.6% report that it takes over an hour on average.

[2]North American Network Operators' Group.

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%

Table 2: Tools used by network administrators (5=most often, 1=least often).

Tools: Table 2 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.

3. NETWORK MODEL

ATPG uses the *header space* framework — a geometric model of how packets are processed we described in [12] (and used in [27]). 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[3] that simplifies the packet generation process significantly.

3.1 Definitions

The definitions in this network model are summarized in Figure 3.

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).

Switch: 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 none match.

Rules: A *rule* generates a list of one or more output packets, corresponding to the output port(s) the packet is sent to; and defines how packet fields are modified. The rule

[3]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	$\Gamma : (p_{src}, h) \rightarrow (p_{dst}, h)$

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 $[(\mathtt{drop}, pk.h)]$

Figure 3: The network model - basic types (left) and the switch transfer function (right)

abstraction models all real-world rules we know including IP forwarding (modify port, checksum and TTL, but not IP address); VLAN tagging (adds VLAN IDs into 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 [12].

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

3.2 Life of a packet

The life of a packet can be viewed as applying the switch and topology *transfer functions* repeatedly (Figure 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).

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($\Gamma(pk_1), switches, \Gamma$)

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

4. ATPG SYSTEM

Based on the network model, ATPG systematically generates the minimal number of test packets based on network state, 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 find the failing rules or links.

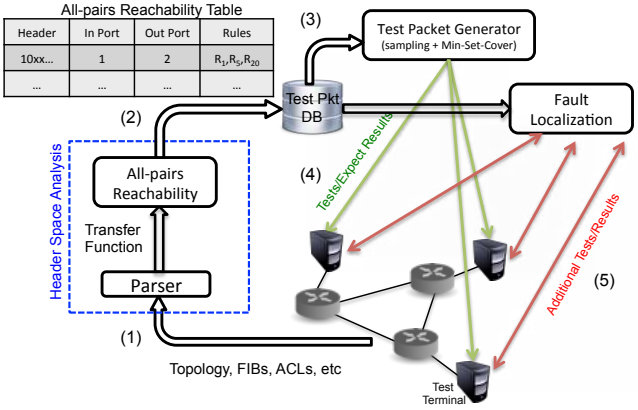


Figure 5: ATPG system block diagram.

Figure 5 shows the block diagram of ATPG system. The system first collects all the forwarding states from the network (step 1). This usually involves reading the FIBs, ACLs or config files and obtaining the topology. ATPG uses Header Space Analysis [12] to find reachability between all the test terminals (step 2). The result is then used by the test packet selection algorithm to find a minimal set of test packets necessary for complete testing of all the rules in the network (step 3). These packets will be sent periodically in the network by the test terminals (step 4). Once an error is detected, the fault localization algorithm is invoked to narrow down the cause of the error (step 5).

4.1 Test Packet Generation

4.1.1 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 the program. This goal can be specialized to testing every link.

When generating test packets, there are two main constraints: (1) *Port:* ATPG must use only test terminals that are available; (2) *Header:* ATPG must use only headers that each test terminal is permitted to send. For example, the network administrator may only allow using a specific set of VLANs. Formally:

PROBLEM 1 (TEST PACKET SELECTION). *For a network with the switch functions, $\{T_1, ..., T_n\}$, and topology function, Γ, find the minimum set of test packets to exercise all reachable rules, subject to the port and header constraints.*

We choose test packets using an algorithm we call *Test Packet Selection (TPS)* algorithm. TPS first finds all the *equivalent classes* between each pair of available ports. An equivalent class is a set of packets that exercise the same combination of rules. It then *samples* each class to find the test packets, and finally *compresses* the resulting set of test packets to find the minimum covering set.

Step 1: Generate an all-pairs reachability table. We start by determining the complete set of packet headers that can be sent from each test terminal, to every other test terminal. For each packet header, we find the complete set of rules it exercises along the path. For this, we directly apply the all-pairs reachability algorithm described in [12]: On every terminal port, we apply an all-`x` header (a header which has all wildcarded bits) to the transfer function of the first hub boxes connected to test terminals. If there is a header constraint, we apply it here. For example, if we can only send traffic on VLAN A, then instead of starting with an all-`x` header, we set the VLAN tag bits to A. We follow the packet through the network using the network function and record the rule history of each resulting packet using *pk.history*. Once this is done for all pairs of terminal ports, we can generate an *all-pairs reachability table* as shown in Table 3. For each row, the header column is a wildcard expression representing the equivalent class that can reach an egress test terminal from an ingress test terminal. Packets matching this header will follow the same set of switch rules, as shown in the right-hand column.

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

Table 3: All-pairs reachability table: all possible headers from every terminal to every other terminal, along with the rules they exercise.

Figure 6 shows a simple example network and Table 4 is the corresponding all-pairs reachability table. For example, if we inject all-`x` test packets at P_A, they will pass through box A. It forwards packets with $dst\_ip = 10.0/16$ to B and those with $dst\_ip = 10.1/16$ to C. Box B then forwards $dst\_ip = 10.0/16, tcp = 80$ to P_B and box C forwards $dst\_ip = 10.1/16$ to P_C. These are reflected in the first two rows of Table 4.

Step 2: Sampling. Next, we need to pick at least one test packet to exercise every (reachable) rule. In fact, by picking one packet in an equivalence class, we can test all of the rules reached by the class. The simplest scheme is to randomly pick one packet per class. This scheme only catches faults for which all packets covered by the same rule will experience the same fault (*e.g.* a link failure). At the other extreme, if we want to catch a fault for a specific header within a equivalence class, then we need to test every header in that class. We discuss these issues, and our fault model, in Section 4.2.

Step 3: Compression. Several of the test packets picked in Step 2 exercise the same rule. We therefore find the minimum subset of the test packets for which the union of their rule histories covers all rules in the network. The cover can be changed to cover all links (for liveness only) or all router queues (for performance only). This is the

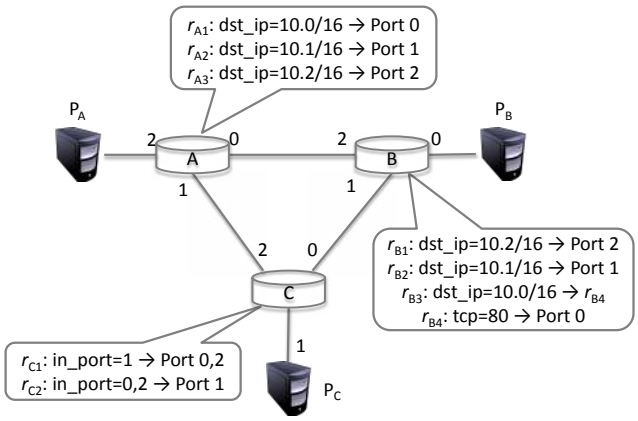


Figure 6: Example topology with three network devices.

classical Min-Set-Cover problem. While NP-Complete, a greedy $O(N^2)$ algorithm provides a very 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 min set are called the *reserved test packets*. In Table 4, $\{p_1, p_2, p_3, p_4, p_5\}$ are regular test packets and $\{p_6\}$ is a reserved test packet. Reserved test packets are useful for fault localization (Section 4.2).

4.1.2 Properties

TPS algorithm has the following useful properties:

Property 1 (Maximum Coverage). *The set of test packets exercise all reachable rules, given the port and header constraints.*

Proof Sketch: We 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 one of the test terminals. If a rule is reachable, it will be in the all-pairs reachability table, and set cover will pick at least one packet that exercises that rule.

Some rules are not reachable. One rule can obscure another: for example one IP address prefix might be obscured by a set of more specific prefixes. Sometimes these are deliberate (to provide backup when a higher priority rule fails or is removed); sometimes they are due to misconfiguration.

Property 2 (Completeness). *For a given set of port and header constraints, the pre-sampled set of test packets selected by TPS represents all possible tests giving complete coverage with minimum test packets.*

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 typical number of rules in each box.*

Proof Sketch: As explained in [12], the complexity of finding reachability from one input port to every port in the network is $O(DR^2)$. We repeat this process once per test terminal.

4.2 Fault Localization

ATPG picks and periodically sends a set of test packets. If test packets fail we also need to pinpoint the fault(s) that caused the problem.

	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

Table 4: Test packets for the example network depicted in Figure 6. p_6 is stored as a reserved packet.

4.2.1 Fault model

A rule fails if its observed behavior is different from what we expected. We keep track of where rules fail using a *result function* R. For a particular rule, r, the result function is defined as

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

"Success" and "failure" depend on the nature of the rule: A failed forwarding rule means a test packet is not delivered to its intended output port, whereas a drop rule behaves correctly when packets are dropped. Similarly, a link failure is a forwarding rule failure in the topology function, and if an output link is congested, it will be captured by increased travel time of a packet.

If we consider all the packets matching a rule, we can divide faults into two categories: *action faults* and *match faults*. An action fault is when *every* packet matching the rule is processed the wrong way. Examples of action faults include unexpected packet loss, a missing rule, congestion, and mis-wiring. On the other hand, match faults are harder to find because they only affect *some* of the packets matching the rule, for example when a rule matches a header that it should not match, or a rule misses a header that it should match. If we want to catch match faults we must sample carefully, and at least one test packet must exercise 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 at that location will be ignored. To catch the error, we need at least two packets to exercise the bit: one with 1 and the other with 0.

We will only consider action faults, because they cover a large number of likely failure conditions, and will be detected by only one test packet per rule. We leave match faults for future work.

We usually can only observe a packet at the edge of the network, and cannot observe it after it has been processed by every rule. Therefore, we define an end-to-end version of the result function

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

4.2.2 Algorithm

Our algorithm for pinpointing faulty rules assumes that a test packet will succeed only if it succeeds at every hop. For intuition, consider `ping` - a `ping` will succeed 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:

ASSUMPTION 1 (FAULT PROPAGATION). *$R(pk) = 1$ if and only if $\forall r \in pk.history$, $R(r, pk) = 1$*

To pinpoint a faulty rule, we start by finding the minimum set of potentially faulty rules. Formally:

PROBLEM 2 (FAULT LOCALIZATION). *Given a list of (pk_0, $R(pk_0)$), (pk_1,$R(pk_1)$), ... 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 our regular test packets. For every passing test, place all the rules they exercise into the set of passing rules, P. If we similarly define all rules traversed by failing test packets F, then one or more of the rules in F are in error. Therefore $F - P$ is a set of *suspect rules*.

Step 2: We want to make the set of suspect rules as small as possible by weeding out all the correctly working rules. For this we make use of the *reserved packets* (which were the packets eliminated by the Min-Set-Cover). From the reserved packets, we find those whose rule history contains *exactly one* rule from the suspect set and send them. If the test packet fails, it shows that the exercised rule is for sure in error. If it passes, we can remove that rule from the suspect set. We then repeat the same process for the rest of the suspect set.

Step 3: In most cases we have a small enough suspect set that we can stop here and report them all. However, we can further narrow down the suspect set by sending test packets that exercise two or more of the rules in the suspect set using Step 2's technique. If these test packets pass, it shows that none of the exercised rules are in error and we can remove them 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: The localization method may introduce *false positives*, which are 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.

When two rules are in series and there is no other path to exercise only one of them, they are *indistinguishable*; any packet that exercises one of them will also exercise the other. Hence if only one rule fails, we cannot tell which. This can happen when, for example, an ACL rule is followed immediately by a forwarding rule, both matching the same header. Indistinguishable rules can be reduced by adding more test terminals. In the extreme, imagine that if we have test terminals before and after each rule, with sufficient test packets, we can distinguish every rule. Hence, the deployment of test terminals not only affects test coverage, but also the accuracy with which we can pinpoint the fault.

5. USE CASES

We can use ATPG for both functional and performance testing, including the cases we list below.

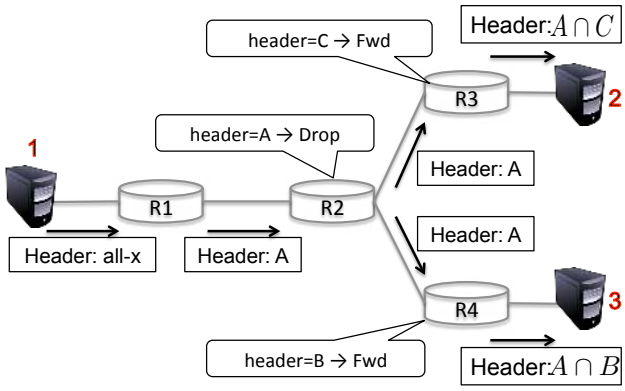


Figure 7: Generate packets to test drop rules: "flip" the rule to a broadcast rule in the analysis.

5.1 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:

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

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 are verifying 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 "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 failure to identify all the ways a packet could reach the egress test terminals if the drop rule was to fail.

As an example consider Figure 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 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 test packets *not* to appear, since their arrival would indicate a failure of $R2$'s drop rule.

5.2 Performance testing

We can 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 queueing 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 we need to test every link, or every queue, or every service class; we then send a stream of packets with these headers to measure the bandwidth. We can use destructive tests, like `iperf/netperf`, or more gentle approaches like packet pairs and packet trains [15]. If we know the available bandwidth of a particular service class should not fall below a certain threshold, when it happens we can use the fault localization algorithm to triangulate and pinpoint the problematic switch/queue.

Strict priorities: Likewise, we can determine if two queues, or service classes, are in different strict priority classes. If they are, then packets sent in 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 verify that the latency and available bandwidth of the higher class is unchanged. If it is, we use fault localization to pinpoint the problem switch/queue.

6. 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].

6.1 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 [9] 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.

6.2 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.

6.3 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 in-

struct 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 interface to control them.

SDN-based testing: In a software defined network (SDN) such as OpenFlow [23], 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.

7. EVALUATION

7.1 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.

Stanford Backbone: With a population of over 15,000 students, 2,000 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 2 backbone Cisco routers that in turn connect Stanford to the outside world. Overall, the network has more than 757,000 forwarding entries and 1,500 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 9 Juniper T1600 routers and 100 Gb/s interfaces, supporting over 66,000 institutions in United States. There are about 100,000 IPv4 forwarding rules. All Internet2 configurations and FIBs of the core routers are publicly available [10], 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.

7.2 Test Packet Generation

We ran our ATPG tool on a quad core Intel Core i7 CPU 3.2GHz and 6GB memory, using 8 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 5 shows the number of test packets we need to send. For example, the first row tells us that if we attach test terminals to 10% of the ports, then we can test all of the reachable rules (22.2% of the total) by sending 725 test packets. If every edge port can act as a test terminal, then we can test 100% of the rules by sending just 3,871 test packets. The "Time" row indicates how long it takes for our ATPG algorithm to run and tells us the test packets we need to send — the worst case took about an hour (most of the time is spent calculating the 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 1Gb/s link we could test the entire network in less than 1ms (assuming each test packet is 100bytes, not considering the propagation delay). Put another way, if we test the entire set of forwarding rules ten times every second, we would use less than 1% of the link bandwidth!

Similarly, we can test all the forwarding rules in Internet2 by sending 4,557 test packets per port in the worst case. Even if the test packets were sent over 10Gb/s links, we could test the entire forwarding rules in less than 0.5ms, or ten times every second using less than 1% of the link bandwidth.

We also found that for 100% *link* coverage (instead of *rule* coverage), we only need 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.

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 possible to carefully select the placement of test terminals to achieve higher coverage, we find that the benefit is marginal with real data sets.

Rule structure: The reason we need so few test packets is because of the structure of the rules and the routing. Most rules are part of an end-to-end route, and so multiple routers hold 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.

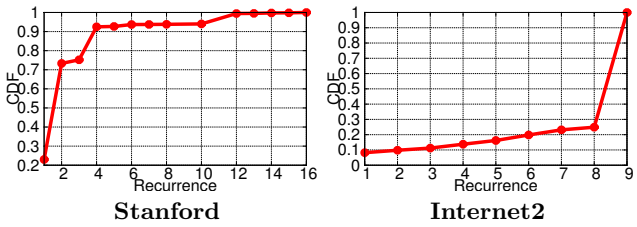


Figure 8: The cumulative distribution function of rule repetition, ignoring different action fields.

We can verify this structure by clustering rules in Stanford and Internet2 that match the same header patterns. Figure 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 2 (50.3%) or 4 (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 9 times, which is the number of routers in the topology.

7.3 Testing in Emulated Network

To evaluate the network monitor and test agents, we replicated the Stanford backbone network in Mininet [16], a container based network emulator. We used Open vSwitch (OVS) [25] 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 [23] rules and installed them in the OVS switches with Beacon [2]. We used emulated hosts to send and receive test packets generated by ATPG. Figure 9 shows the part of network that is used for experiments in this section. We now present different test scenarios and the corresponding results:

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

Table 5: 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 100B per test packet, 1Gbps link for Stanford and 10Gbps for Internet2.

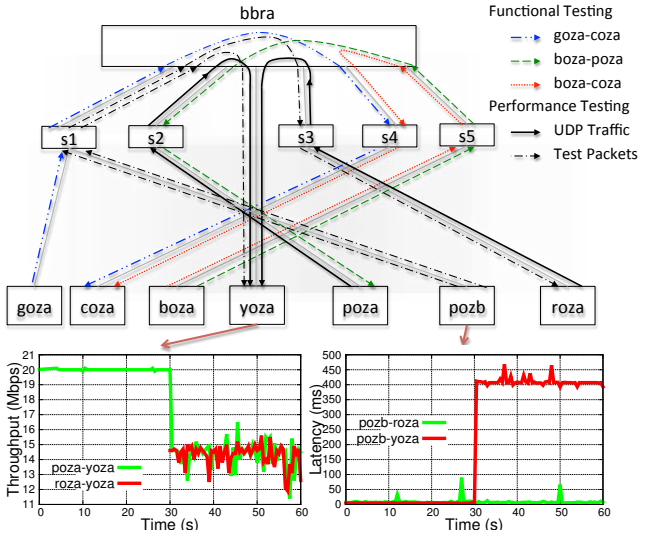


Figure 9: A portion of the Stanford backbone network showing the test packets used for functional and performance testing examples in Section 7.3.

Forwarding Error: To emulate a functional error, we deliberately created a fault by replacing the action of an IP *forwarding* rule in *boza* matching on $dst\_ip = 172.20.10.32/27$ 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 6 shows two other test packets we used to localize and pinpoint the fault. These test packets shown in Figure 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 3, 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 30Mb/s, and create two 20Mb/s UDP flows: *poza* to *yoza* at $t = 0$ and *roza* to *yoza* at $t = 30s$, which will congest the link $bbra - yoza$ starting at $t = 30s$. The bottom left graph next to *yoza* in Figure 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 $t = 30s$, the $bozb - yoza$ test packet experience a much higher latency, correctly signaling congestion. Since these two test packets share the $bozb - s_1$ and $s_1 - bbra$ links, we can conclude that the congestion is not happening in these two links, therefore we can correctly infer that $bbra - yoza$ is the congested link.

Available Bandwidth: ATPG can also be used to monitor available bandwidth. For this experiment, we used Pathload [11], a bandwidth probing tool based on packet pairs/packet trains. We repeated the previous experiment, but decreased the two UDP flows to 10Mb/s, so that the bottleneck available bandwidth was 10Mb/s. Pathload reports that $bozb - yoza$ has an available bandwidth[5] of 11.715Mb/s, $bozb - roza$ has an available bandwidth of 19.935Mb/s, while the other (idle) terminals report 30.60Mb/s. Using the same argument as before, we automatically conclude that $bbra - yoza$ link is the bottleneck link with 10Mb/s available bandwidth.

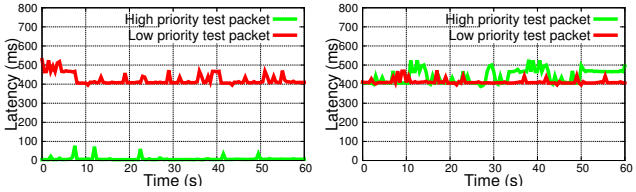
Priority: We created priority queues in OVS using Linux's `htb` scheduler and `tc` utilities. We replicated the previously "failed" test case $pozb - yoza$ for high and low priority queues respectively.[6] Figure 10 shows the result.

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

[4]To measure latency in a real network, two-way communication is usually necessary. However, relative change of latency is sufficient to uncover congestion.
[5]All numbers are the average of 10 repeated measurements.
[6]The Stanford data set does not include the priority settings.

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

Table 6: 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.



(a) Low **(b) High**

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

(c) Available bandwidth

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

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. But for low priority flows, the high priority test packets correctly receive the full link bandwidth.

7.4 Production Network Deployment

We deployed an experimental ATPG system in 3 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 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 minutes, and conducts `ping` tests every 10 seconds. Test results were logged locally as files and collected daily for analysis.

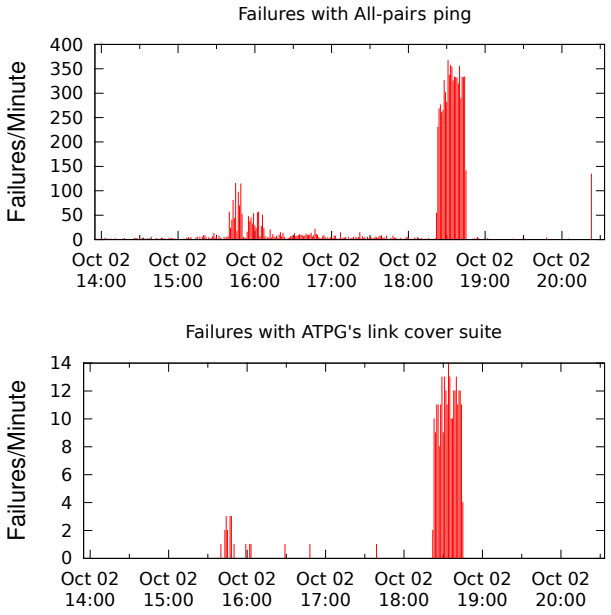


Figure 11: The Oct 2, 2012 production network outages captured by the ATPG system as seen from the lens of an inefficient cover (all-pairs, top picture) and an efficient minimum cover (bottom picture). Two outages occured at 4PM and 6:30PM respectively.

During the experiment, a major network outage occurred on October 2. Figure 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 with 2756 packets in all-pairs `ping`, a 100x 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 occured. 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.

8. DISCUSSION

8.1 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 rerunning 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/multi-machine 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 system that incorporates both techniques.

Test agents within terminals incur minimal overhead because they need to merely demultiplex test packets addressed to their IP address at a modest rate (say 1 per msec) compared to the link speeds most modern CPUs and operating systems are capable of receiving (1 Gbps or higher).

8.2 Limitation

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, a NAT that dynamically assigns TCP ports to outgoing packets can confuse the online monitor as the same test packet can give different results. **2) Non-deterministic 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.

9. RELATED WORK

We are unaware of earlier techniques that automatically generate test packets from configurations. The closest related work we know of are offline tools that check invariants of different components in networks. On the control plane, NICE [3] attempts to exhaustively cover the code paths symbolically in controller applications with the help of simplified switch/host models. On the data plane, Anteater [21] models invariants as boolean satisfiability problems and checks them against configurations with a SAT solver. Header Space Analysis [12] uses a geometric model to check reachability, detect loops, and verify slicing. Recently, SOFT [14] is proposed to verify the 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 be otherwise captured.

End-to-end probes have long been used in network diagnosis. Duffield uses Binary Tomography [5, 6] to detect the smallest set of failed links that explains end-to-end measurements. NetDiagnoser [4] further combines end-to-end probes with routing data. Researchers also use various models to correlate network metrics with network events [13, 19, 20, 22]. Recently, mining low-quality, unstructured data, such as router configurations and network tickets has attracted interest [8, 17, 30]. 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 link or every rule. Further, ATPG is not limited to liveness testing but can be applied to checking higher level properties such as performance.

Many approaches to develop a measurement-friendly architecture are proposed for managing large networks [18, 24, 31]. It is also suggested that routers should be able to sample the packets for measurement [7]. Our approach is complementary to these proposals: ATPG does not dictate the locations of injecting network probes and how these probes should be constructed. By incorporating input and port constraints, ATPG can generate test packets and injection points using existing deployment of measurement devices.

10. 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. It suffices to find a minimal set of end-to-end packets that traverse each link. 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 goes much further than liveness testing, however, 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 with minimum 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 (<4,000 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 demands 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 rules) and dynamic (e.g., timing) verification. We hope ATPG is a useful starting point for automated dynamic testing of production networks.

11. ACKNOWLEDGEMENT

We would like to thank our shepherd, Dejan Kostic, and the anonymous reviewers for their valuable comments. We thank Johan van Reijendam, Charles M. Orgish, Joe Little (Stanford University) and Thomas C. Knoeller, Matthew P. Davy (Internet2) for providing router configuration sets and sharing their operation experience. This research was funded by NSF grants CNS-0832820, CNS-0855268, CNS-1040593, and Stanford Graduate Fellowship.

12. REFERENCES

[1] ATPG code repository. `http://eastzone.github.com/atpg/`.

[2] Beacon. `http://www.beaconcontroller.net/`.

[3] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test openflow applications. *Proceedings of the 9th conference on Symposium on Networked Systems Design & Implementation*, 2012.

[4] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot. Netdiagnoser: troubleshooting network unreachabilities using end-to-end probes and routing data. In *Proceedings of the 2007 ACM CoNEXT conference*, CoNEXT '07, pages 18:1–18:12, New York, NY, USA, 2007. ACM.

[5] N. Duffield. Network tomography of binary network performance characteristics. *Information Theory, IEEE Transactions on*, 52(12):5373 –5388, dec. 2006.

[6] N. Duffield, F. Lo Presti, V. Paxson, and D. Towsley. Inferring link loss using striped unicast probes. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 915 –923 vol.2, 2001.

[7] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Trans. Netw.*, 9(3):280–292, June 2001.

[8] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 350–361, New York, NY, USA, 2011. ACM.

[9] Hassel, the header space library. `https://bitbucket.org/peymank/hassel-public/`.

[10] The Internet2 Observatory Data Collections. `http://www.internet2.edu/observatory/archive/data-collections.html`.

[11] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with tcp throughput. *IEEE/ACM Trans. Netw.*, 11(4):537–549, Aug. 2003.

[12] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: static checking for networks. *Proceedings of the 9th conference on Symposium on Networked Systems Design & Implementation*, 2012.

[13] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Ip fault localization via risk modeling. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 57–70, Berkeley, CA, USA, 2005. USENIX Association.

[14] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT way for openflow switch interoperability testing. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, CoNEXT '12, 2012.

[15] K. Lai and M. Baker. Nettimer: a tool for measuring bottleneck link, bandwidth. In *Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems - Volume 3*, USITS'01, pages 11–11, Berkeley, CA, USA, 2001. USENIX Association.

[16] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets '10, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[17] 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.*, 17(1):66–79, Feb. 2009.

[18] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iplane: an information plane for distributed services. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 367–380, Berkeley, CA, USA, 2006. USENIX Association.

[19] 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 *Proceedings of the Eighth COnference on emerging Networking EXperiments and Technologies*, CoNEXT '12, pages 13:1–13:12, New York, NY, USA, 2011. ACM.

[20] A. Mahimkar, J. Yates, Y. Zhang, A. Shaikh, J. Wang, Z. Ge, and C. T. Ee. Troubleshooting chronic conditions in large ip networks. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 2:1–2:12, New York, NY, USA, 2008. ACM.

[21] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. *SIGCOMM Comput. Commun. Rev.*, 41(4):290–301, Aug. 2011.

[22] 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.*, 16(4):749–762, Aug. 2008.

[23] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, March 2008.

[24] OnTimeMeasure. `http://ontime.oar.net/`.

[25] Open vSwitch. `http://openvswitch.org/`.

[26] All-pairs ping service for PlanetLab ceased. `http://lists.planet-lab.org/pipermail/users/2005-July/001518.html`.

[27] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference*, SIGCOMM '12. ACM, 2012.

[28] S. Shenker. The future of networking, and the past of protocols. `http://opennetsummit.org/talks/shenker-tue.pdf`.

[29] Troubleshooting the Network Survey. `http://eastzone.github.com/atpg/docs/NetDebugSurvey.pdf`.

[30] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: understanding the causes and impact of network failures. *SIGCOMM Comput. Commun. Rev.*, 41(4):–, Aug. 2010.

[31] P. Yalagandula, P. Sharma, S. Banerjee, S. Basu, and S.-J. Lee. S3: a scalable sensing service for monitoring large networked systems. In *Proceedings of the 2006 SIGCOMM workshop on Internet network management*, INM '06, pages 71–76, New York, NY, USA, 2006. ACM.