Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 4, Issue. 6, June 2015, pg.447 – 455

RESEARCH ARTICLE

Self-Generation of Test Packet for Heterogeneous IP Network

Dr. Shubhangi D C
Department of Computer Science and Engineering, VTU RO Kalaburagi, India
Drshubhangipatil1972@gmail.com

Hanamanth B
Department of Computer Science and Engineering, VTU RO Kalaburagi, India
hb_malage@rediffmail.com

Abstract— Chronic network conditions are caused by performance impairing events that occur intermittently over an extended period of time. Such conditions can cause repeated performance degradation to customers, and sometimes can even turn into serious hard failures. It is therefore critical to troubleshoot and repair chronic network conditions in a timely fashion in order to ensure high reliability and performance in large IP networks. Networks are getting larger and more complex, yet administrators rely on rudimentary tools such as ping and trace 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 (mini-mally) 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: Using this technique we analyze over five years of failure events in a large regional network consisting of over 200 routers; to our knowledge, this is the largest study of its kind.

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

• Introduction

The demand for sophisticated tools for monitoring network utilization and performance has been growing rapidly as Internet Service Providers (ISPs) offer their customers more services that require quality of service (QoS) guarantees and as ISP networks become increasingly complex. Tools for monitoring link delays and faults in an IP network are critical for numerous important network management tasks, including providing QoS guarantees to end applications (e.g., voice over IP), traffic engineering, ensuring service level agreement (SLA) compliance, fault and congestion detection, performance debugging, network operations, and dynamic replica selection on the Web. Consequently, there has been a recent flurry of both research and industrial activity in the area of developing novel tools and infrastructures for measuring network parameters. It is notoriously hard to debug networks. Every day, network engineers wrestle with router misconfigurations, fiber cuts, faulty interfaces, mislabeled cables, software bugs,

© 2015, IJCSMC All Rights Reserved

intermittent links, and a myriad other reasons that cause net-works to misbehave or fail completely. Network engineers hunt down bugs using the most rudimentary tools (e.g Ping and Traceroute, SNMP) 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 small wonder that network engineers have been labeled "masters of complexity".

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.

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 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 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 net-work behaves as designed, all three steps should remain consistent at all times, i.e., $A \leftrightarrow B \leftrightarrow C$. 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 $A \leftrightarrow B$, enforcing consistency between *policy* and the *configuration*. 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 $B \leftrightarrow C$. Alice's first problem was with L (link not working), and her second problem was with $B \leftrightarrow C$ (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 net-work failure are hardware failures and software bugs, and that problems manifest themselves *both* as reachability failures and throughput/latency degradation. Our goal is to automatically 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

© 2015, IJCSMC All Rights Reserved

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. How-ever, all-pairs Ping does not guarantee testing of all links and has been found to be unsalable for large networks such as Planet Lab [30].

We tested our method on two real-world data sets the back-bone 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 rule sets, 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 net-work bandwidth.

The contributions of this paper are as follows:

- a survey of network operators revealing common failures and root causes (Section II);
- a test packet generation algorithm (Section IV-A);
- a fault localization algorithm to isolate faulty devices and rules (Section IV-B);
- ATPG use cases for functional and performance testing (Section V);

Evaluation of a prototype ATPG system using rule sets collected from the Stanford and Internet2 backbones (Sections VI and VII).

- **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 Open Flow 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 granularity 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.

© 2015, IJCSMC All Rights Reserved

- **System architecture**

Based on the network model, ATPG generates the minimal number of test packets so that every forwarding rule in the net-work 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. 1 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 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, 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:

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, , determine the minimum set of test packets to exercise all reachable rules, subject to the port and header constraints.

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

Bit b=0|1|x
Header h=[b_0,b_1,….,b_l]
Port p=1|2|…|N|drop
Packet pk=(p,h)
Rule r:pk→pk or [pk]
Match r.matchset : [pk]
Transfer Function T : pk→pk or [pk]
Topo Function Γ : (p_{src},h) → (p_{dst},h)

Fig.2. Network model: Basic types

Function T_i(pk)
#Iterate according to priority in switch i
For r ϵ $ruleset_i$ do
If pk ϵ r.matchset then
Pk.history ← pk.history U {r}
Return r(pk)
Return [(drop,pk.h)]

Fig.3. Network model: The switch transfer function.

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 P_k arrives at a network port, the switch function that contains the input port $P_{k.p}$ is applied to P_k,

© 2015, IJCSMC All Rights Reserved

producing a list of new packets [P_{k1},…., P_{kn}]. 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).

input: Topology *T* ; End-to-end measurements *fXkgk2R*;
Y0 = 1;
W = ;;
recurse(1);
output: *W*;
subroutine recurse(*k*) {
if(*k 2 R*) *fYk* = *Xkg*;
else *f*
Yk = max*j2d(k)* *Yj* ;
foreach(*j 2 d*(*k*))*{*
if (*Yj* = 0 && *Yk* = 1)*{*
W = *W [fjg*;
}
}
}

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

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: 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- header, the VLAN tag bits are set to A. As each packet P_k traverses the network using the network function, the set of rules that match P_k are recorded in P_k.history. Doing this for all pairs of terminal ports generates an all-pairs reachability table as shown in Table I. 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.

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.

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 I
All-Pairs Reachability Table: All Possible Headers from Every Terminal To Every Other Terminal, Along With The Rules They Exercise

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 covers 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, are regular test packets, and $\{p_6\}$ is a re-served test packet. Reserved test packets are useful for fault localization.

© 2015, IJCSMC All Rights Reserved

This section defines the algorithm for inferring the identity of bad links. The Smallest Consistent Failure Set (SCFS) algorithm designates as bad only those links nearest the root that are consistent with the observed pattern of bad paths. Define an indicator variable *Zk* to be 1 if link *k* is good, and 0 if it is bad; for the root node 0 set *Z*0 = 1 by convention. For each path from the root 0 to the node *k*, let *Xk* = 1 if the path is good, and 0 if it is bad. Under the separability assumption, we can write *Xk* = Y *j*º*k Zj* (3) i.e. the product of the link indicators *Xj* for ancestors *j* of *k* (including *k* itself). Let *Rk* denote the set of leaf nodes in *R* that are descended from *k*. Write *Yk* = max*j2Rk Xj* for *k 2 U* and set *Y*0 = 1 by convention. *Yk* = 1 if and only if at least one source-destination path routed through *k* is good. Clearly, if *Yk* = 1, then the path segment from 0 to *k* is composed entirely of good links. If *Yk* = 0 but *Yf*(*k*) = 1 we call the subtree rooted at *k* a *maximal bad subtree*. A cautious approach would be to declare as bad link *k* and all links in the subtree. In practice, this is probably not very useful due to the cost of inspecting all the links for badness.

The SCFS algorithm takes the other extreme by designating as bad the link in the subtree that is most likely to be amongst the set of bad links, namely, the link *k*. For suppose, on the other hand, that *k* is not bad. Then all the path segments from *k* to the destinations in *R*(*k*) must be bad. This is, of course, possible, which is why we cannot pin down the bad links with certainty. However, if the rate of occurrence of bad links is sufficiently small, then, as we shall see, it is far more likely that the link *k* is bad. Anticipating this, we form an inference algorithm which designates link *k* to be bad and its entire descendant links good. Put another way, we estimate *Zk* by b *Zk* = 0, while for all links *j* with *j* ¹ *k* we estimate *Zj* by b *Zj* = 1.

• 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.

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

As an example, consider Fig. 1. 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 pin-point 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 avail-able 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, or more gentle approaches like packet pairs and packet trains. Suppose a manager specifies that the available bandwidth of a

© 2015, IJCSMC All Rights Reserved

particular service class should not fall below a certain threshold; if it does happen, ATPG's fault localization algorithm can be used to tri-angulate 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.

• 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.

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 net-work monitor reads the database and constructs test packets and instructs each agent to send the appropriate packets. Currently, test agents separate test packets by IP Proto field and TCP/UDP port number, but other fields, such as IP option, can also be used. If some of the tests fail, the monitor selects additional test packets from reserved packets to pinpoint the problem. The process repeats until the fault has been identified. The 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 router to send/receive test packets. In fact, for manufacturing testing purposes, it is likely that almost every commercial switch/router can already do this; we just need an open inter-face to control them.

SDN -Based Testing: In a software defined network (SDN) such as OpenFlow, 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.

• 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 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.

© 2015, IJCSMC All Rights Reserved

B. Limitations

As with all testing methodologies, ATPG has limitations.

- *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.
- *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.
- *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.
- *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.
- *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.

• 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 reach-ability), and finally determining a minimum set of test packets (Min- Set-Cover).

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 <4000 0 for Internet2).

Network managers today use primitive tools such as Ping and tracerout. 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 bil-lion-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 *Pat-tern* Generation [2]. We hope network ATPG will be equally useful for automated dynamic testing of production networks.

• REFERENCES

- "ATPG code repository," [Online]. Available: http://eastzone.github. com/atpg/
- "Automatic Test Pattern Generation," 2013 [Online]. Available:http://en.wikipedia.org/wiki/Automatic_test_pattern_generation
- P. Barford, N. Duffield, A. Ron, and J. Sommers, "Network perfor-mance anomaly detection and localization," in *Proc. IEEE INFOCOM*, Apr. , pp. 1377–1385.
- "Beacon,"[Online].Available: http://www.beaconcontroller.net/
- 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.
- 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.

© 2015, IJCSMC All Rights Reserved

- 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.
- 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..
- N. Duffield, "Network tomography of binary network performance characteristics," *IEEE Trans. Inf. Theory*, vol. 52, no. 12, pp. 5373–5388, Dec. 2006.
- 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.
- 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
- Hassel, the Header Space Library," [Online]. Available: https://bit-bucket.org/peymank/hassel-public/
- Internet2, Ann Arbor, MI, USA, "The Internet2 observatory data col-lections," [Online]. Available: http://www.internet2.edu/observatory/ archive/data-collections.html
- M. Jain and C. Dovrolis, "End-to-end available bandwidth: Measure-ment methodology, dynamics, and relation with TCP throughput," *IEEE/ACM Trans. Netw.*, vol. 11, no. 4, pp. 537–549, Aug. 2003.
- P. Kazemian, G. Varghese, and N. McKeown"Header space analysis: Static checking for networks," in *Proc. NSDI*, 2012, pp. 9–9.
- 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.
- 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.
- 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.
- 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.
- 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.
- H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Kr-ishnamurthy, and A. Venkataramani, "iplane: An information plane for distributed services," in *Proc. OSDI*, Berkeley, CA, USA, 2006, pp. 367–380.
- 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.
- 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.
- 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.
- 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.
- 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.
- "OnTimeMeasure," [Online]. Available: http://ontime.oar.net/
- "Open vSwitch," [Online]. Available: http://openvswitch.org/
- H. Weatherspoon, "All-pairs ping service for PlanetLab ceased," 2005 [Online]. Available: http://lists.planet-lab.org/pipermail/users/2005-July/001518.html
- M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Ab-stractions for network update," in *Proc. ACM SIGCOMM*, 2012, pp. 323–334.
- S. Shenker, "The future of networking, and the past of protocols," 2011 [Online]. Available: http://opennetsummit.org/archives/oct11/shenker-tue.pdf
- "Troubleshooting the network survey," 2012 [Online]. Available: http://eastzone.github.com/atpg/docs/NetebugSurvey.pdf
- 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.
- 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.

© 2015, IJCSMC All Rights Reserved