AUTOMATIC DIAGNOSIS USING TEST PACKET GENERATION

Sri Gowthem.S [1] **Manivannan.D [2]**

[1]Assistant Professor, Department of CSE, Bharath University, Chennai, srigowthem.am@bharathuniv.ac.in

[2]PG Student, Dept. of CSE, Bharath University, Chennai, mail2manivan@gmail.com

ABSTRACT- Networks are getting larger and more complex.Which results in improper maintenance of the networks. Yet administrators rely on rudimentary tools like ping and traceroute to debug problems, despite it being getting lesser and lesser reliable. It is notoriously hard to debug networks. Everyday network engineers wrestle with router misconfigurations, fiber cuts, software bugs and myriad of reasons that cause the networks to misbehave and fail completely. An Automatic Test Packet Generation (ATPG) framework, which is an automated and a systematic approach to for testing and debugging networks, automatically generates a minimal set of packets to exercise every link in the network as well as every rule in the network. 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. It used for testing the liveness of the underlying topology and the congruence between data plane state and configuration specification. It also complements but goes beyond the earlier work in static checking and fault localization. The tool can automatically generate packets to test performance assertions such as packet latency.

Keywords - Dataplane analysis, network troubleshooting ,test packet generation, fault localization,network monitoring.

1. INTRODUCTION

Whenever networking comes into picture, questions that we come across are about "How to secure your network? Is my network secure? What do I need to do make network secure?" But network security does not limit only by implementing new firewall optimizing techniques or to secure the information, rather it also includes monitoring the packets, forwarding entries etc. Now, this would arise the question of how this would help to secure the network. The answer to this is, the security could be easily breached by tampering the rules and exploiting the errors.

Until now it is the network administrator's problem to tackle with such issues. 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. (See Fig.1) But creating a tool using ATPG algorithm would automate the entire process.

Therefore our goal is "To build a system which would automatically monitor functional and performance faults in network. To detect and diagnose errors by independently and exhaustively testing all forwarding entries, firewall rules, and any packet processing rules in the network. To check the liveness and fault localization of the network.

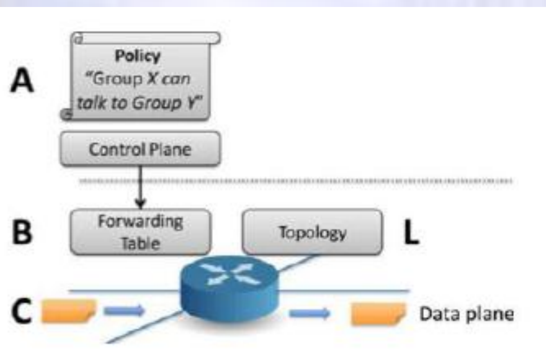

Fig.1. Static versus dynamic checking: A policy is compiles forwarding state, which is the n executed by the forwarding plane. Static checking (e.g., confirms that A=B. Dynamic checking (e.g., APTG in this paper) confirms that the topology is meeting liveness properties (L) and that B=C

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. While using ping and traceroute the network administrators usually use a crude lenses to examine the current forwarding state for clues to track down failure. This would be a cumbersome method since the forwarding state is hard to observe because it typically requires login into every box in the network. Recently, researchers have proposed tools to check that $A=B$, enforcing consistency between *policy* and the *configuration* [1], [2], [3], [4]. While these approaches can find (or prevent) software logic errors in the control plane, they are *not* designed to identify liveness failures caused by failed links and routers, bugs caused by faulty router hardware or software, or performance problems caused by network congestion. Such failures require checking for L and whether $B=C$. Network Administrator's problem was with $B=C$. (low level token bucket state not reflecting policy for video bandwidth). Instead of manually deciding which packet to send 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 [5] *proactively* check the health of their

network using pings between all pairs of sources. However, all-pairs does not guarantee testing of all links and has been found to be unscalable for large networks such as PlanetLab [6].

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. The contributions of this paper are as follows: 1) A test packet generation algorithm (Section 3); 2) A fault localization algorithm to isolate faulty devices and rules (Section 4);

II. NETWORK MODEL

Let’s get familiar with some keywords.

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

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

Bit	$b = 0 \mid 1 \mid \mathrm{x}$
Header	$h = [b_0, b_1, \ldots, b_L]$
Port	$p = 1 \mid 2 \mid \ldots \mid N \mid \mathtt{drop}$
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	$\Gamma : (p_{src}, h) \rightarrow (p_{dst}, h)$

Fig. 2 summarizes the definitions in our model.

Fig.2. Summarizes the definitions in our model

Rules: A *rule* generates a list of one or more output packets, corresponding to the output port(s) to which the packet is sent, and defines how packet fields are modified. The rule abstraction models all real-world rules we know including IP forwarding (modifies port, checksum, and TTL, but not IPaddress); VLAN tagging (adds VLAN IDs to the header); and ACLs (block a header, or map to a queue). Essentially, a rule defines how a region of header space at the ingress (the set of packets matching the rule) is transformed into regions of header space at the egress [1].

Rule History: At any point, each packet has a *rule history (r0, r1,…)* an ordered list of rules 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*, , models thenetwork 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 match an input port, the port is an edge port, and the packet has reached its destination.

```
function T_i(pk)
    #Iterate according to priority in switch i
    for r ∈ ruleset_i do
        if pk ∈ r.matchset then
            pk.history ← pk.history ⋃{r}
            return r(pk)
    return [(drop, pk.h)]
```

Fig 3 Switch transfer function

III. ATPG

Let's consider a scenario where an administrator maps video traffic to a specific queue in a router, and packets are dropped because the token bucket rate is low. What would the network administrator do in such case?

Current system

- The administrator manually decides which ping packets to send.
- Here, the approaches designed can prevent software logic errors but fails to detect failures caused by failed links and routers.

ATPG system

- Instead of the administrator, the ATPG tool would do so periodically on his or her behalf.
- Whereas here, ATPG automatically detects the failures by testing the liveness of the underlying topology.

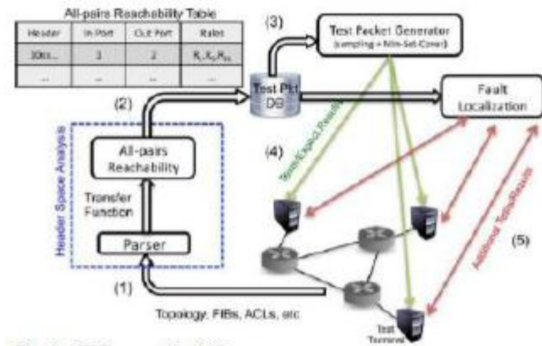

Fig. 4. ATPG system block diagram.

When an error is detected, ATPG goes through the following steps:

1. The system first collects all the forwarding state from the network
2. ATPG uses Header Space Analysis to compute reachability between all the test terminals.
3. The result is then used by the test packet selection algorithm to compute a minimal set of test packets that can test all rules.
4. These packets will be sent periodically by the test terminals.
5. If an error is detected, the fault localization algorithm is invoked to narrow down the cause of the error.

Step 1: *Collect all forwarding states:* Forwarding table which usually involves reading the FIBs (Forwarding Information States), ACLs (Access Control Lists), and config files, as well as obtaining the topology.

Step 2: *Generate All-Pairs Reachability Table:* ATPG Start's 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 as follows:

1. Header constraints are applied. For example, if traffic can be sent on VLAN A, then instead of starting with an all- x header, the VLAN tag bits are set to A.
2. Set of rules that match the packet are recorded in packet history. Hence all-pairs reachability table as shown in table1.

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

Table1 ALL-PAIRS REACHABILITY TABLE: ALL POSSIBLE HEADERS FROM EVERY TERMINAL TO EVERY OTHER TERMINAL, ALONG WITH THE RULES THEY EXERCISE

Therefore all packets matching this class of header will encounter the set of switch rules.

	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

Step 3: *Test Packet Generation:* We assume a set of *test terminals* in the network can send and receive test packets. Our goal is to generate a set of test packets to exercise *every* rule in *every* switch function, so that *any* fault will be observed by at least one test packet. This is analogous to software test suites that try to test every possible branch in a program. The broader goal can be limited to testing every link or every queue.

When generating test packets, ATPG must respect two key Constraints:

1. *Port:* ATPG must only use test terminals that are available;
2. *Header:* ATPG must only use headers that each test terminal is permitted to send.

For example, the network administrator may only allow using a specific set of VLANs. Formally, we have the following problem.

Problem (Test Packet Selection): For a network with the switch functions $\{T_1, T_2..T_n\}$, and topology function, T, 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.

IV. FAULT LOCALIZATION ALGORITHM

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

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

We divide faults into two categories: *action faults* and *match Faults.* An action fault occurs when *every* packet matching the rule is processed incorrectly. Action faults include unexpected packet loss, a missing rule, congestion, and miswiring. On the other hand, match faults are harder to detect because they

only affect *some* packets matching the rule: for example, when a rule matches a header it should not, or when a rule misses a header it should match. We will only consider action faults because they cover most likely failure conditions and can be detected using only one test packet per rule.

2) Problem 2 (Fault Localization): Given a list of $(pk_0, (R(pk_0),(pk_1, (R(pk_1))$ … tuples, find all that satisfies

□ $pk_i, R(pk_i, r)=0$.

Step 1: Consider the results from sending the regular test packets. For every passing test, place all rules they exercise into a set of passing rules, *P*. Similarly, for every failing test, place all rules they exercise into a set of potentially failing rules *F*. By our assumption, one or more of the rules *F* are in error. Therefore *F-P*, is a set of *suspect rules*.

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

Step 3: In most cases, the suspect set is small enough after Step 2, which ATPG can terminate and report the suspect set. f needed, ATPG can narrow down the suspect set further by sending test packets that exercise two or more of the rules inthe suspect set using the same technique underlying Step 2. If these test packets pass, ATPG infers that none of the exercised rules are in error and removes these rules from the suspect set. If our Fault Propagation assumption holds, the method will notmiss any faults, and therefore will have no *false negatives*.

False Positives: Note that the localization method may introduce *false positives*, rules left in the suspect set at the end of Step 3. Specifically, one or more rules in the suspect set may in fact behave correctly. False positives are unavoidable in some cases. When two rules are in series and there is no path to exercise only one of them, we say the rules are *indistinguishable*; any packet that exercises one rule will also exercise the other. Hence, if only one rule fails, we cannot tell which one. For example, if an ACL rule is followed immediately by a forwarding rule that matches the same header, the two rules are indistinguishable. Observe that if we have test terminals before and after each rule (impractical in many cases), with sufficient test packets, we can distinguish every rule. Thus, the deployment of test terminals not only affects test coverage, but also localization accuracy.

V. CONCLUSIONS

Current System uses a method which is neither exhaustive nor scalable. Even though it reaches all the pairs of edge nodes it fails to detect faults in liveness properties. 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.

VI. REFERENCES

[1] Hongyi Zeng, *Member, IEEE*, Peyman Kazemian, *Member, IEEE*, George Varghese, *Member, IEEE, Fellow, ACM*, and Nick McKeown, *Fellow, IEEE, ACM,* "Automatic Test Packet Generation".

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

[3] 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.

[4] 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.

[5] Hongyi Zeng, Peyman Kazemian,George Varghese,and Nick McKeown,"Automatic Test Packet Generation",VOL. 22, NO. 2,
APRIL 2014.

International Journal for Computer Science