Formal Network Testing

Hongyi Zeng[†‡], Peyman Kazemian[†‡], George Varghese[*], Nick McKeown[†]
[†] {kazemian,hyzeng,nickm}@stanford.edu, *Stanford University, Stanford, CA USA*
[*] varghese@cs.ucsd.edu, *UCSD, San Diego and Yahoo! Labs, Santa Clara, CA, USA*
[‡] These authors contributed equally to this work

ABSTRACT

Certain types of errors, such as hardware failure, link failure or congestion can only be detected at runtime of network as these are failures that happen when the network is in operation. Today, there exist only preliminary tools to detect and pinpoint such kind of errors, typified by ping and trace rout. In this paper we introduce a formal and comprehensive way to fully test networks at runtime which enables us to pinpoint the exact rule that is causing the problem. Our test packet generation algorithm first picks a relatively small number of test packets and corresponding input ports so that these packets exercise all the rules in network. By periodically sending these packets through the network, we can constantly monitor the health of the network. Whenever an error happens, the fault localization algorithm uses the result of all test packets to find out the exact rule that is in fault.

1. INTRODUCTION

Networks can suffer from two types of errors. The first type is configuration problems where due to a configuration mistake, a server is unreachable or a security hole exists. These sort of errors which are static and often times is a result of human mistakes, can be detected offline and statically using methods such as [1] and [3]. It is equivalent to compile-time checking in a computer program. The second type of errors are dynamic problems that happen at the runtime of the network. Problems such as link failure, hardware failure, buffer overflow, over utilization of resources in a switch or bugs in software of devices fall in this category. Due to run-time nature of these errors, it is impossible to detect these sort of failures statically. Finding and resolving these sort of problems are often hard and requires a lot of experience and deep understanding of different components and protocols in the network.

Today, dynamic checking of networks is black magic . Usually a combinations of ping and traceroute together with CLI commands on the device itself is being used. The debug time depends on the technical sophistication and experience of the admin and the complexity of the network.

In this paper, we develop a systematic approach toward runtime testing of networks. Specifically, our testing mechanism exercises every rule in the network periodically to ensure complete coverage of the network, the same way as a good test suite for a program covers all branches of the program. We monitor the "health" of the network by periodically sending carefully-chosen test packets from terminal points of the network. If a test packet fails to reach to its intended destination, it signals an error in the network, which then triggers more test packets to pinpoint the failure to the exact box and rule.

In summary, the contributions of this paper and an outline of the rest of this paper are as follows:

- *Test Packet Generation:* We first review Header Space Analysis in section 2. Then in section 3 we introduce an algorithm for picking a minimum set of test packets to maximize the test coverage of the network.
- *Finding Errors:* In section 4 we talk about fault localization: once an error has happened, how to pinpoint exactly which part of the network is causing the problem.
- *Implementation and Evaluation:* in section 5 we talk about our initial implementation of the framework and discuss some experimental results on Stanford backbone network in section 6.

Finally we discuss related works in section 7 and conclude in section 8.

2. BACKGROUND

2.1 Router configuration on data plane

Most network device configurations can be divided into two portions - control plane and data plane. Control plane rules include both individual device information (e.g. name, SNMP), and information about inter-device protocols (e.g. OSPF, STP). These rules can be considered as "meta states" of the network since they do not directly reflect forwarding. Control plane correctness can be verified using various methods. [4, 5]

For the rest of this paper, we will focus on testing data plane rules, which directly control the packet forwarding and/or rewrites packet headers. We take router as an example to examine its internals. In a modern router, these rules can be classified into three categories:

Port configurations: Ports of a router can serve different purposes, some of them may affect the router's data plane behavior. For example, Most routers todays support Virtual LAN (VLAN) tagging. A router port can be configured as an "access port" that tags incoming packets with particular VLAN IDs, a "trunk port" that accepts certain VLAN IDs, or an "untagged port" that is oblivious to VLAN IDs. A router can also create a virtual "tunnel port" that connects to a remote router through IP encapsulation or other tunnel mechanisms.

Forwarding rules: Layer 3 Forwarding rules can be generated by routing protocols such as BGP and OSPF. These rules usually consist an IP prefix and the next hop interface/IP address. MAC address learning and STP among interfaces within the same VLAN can imply Layer 2 forwarding rules. Besides dynamic generation, router administrators can install static rules manually. Forwarding rules are often ordered by priority. When a packet matches multiple rules, only the one with highest priority will effect.

Access lists (ACLs): Router administrators can achieve finer configuration granularity by composing ACLs. An ACL is an ordered list defining a specific class of packets, with each entry in the list one header tuple. It can be used in conjunction with port configurations to enforce whether this class of packets should be accepted or dropped. Depending on the position that ACLs are placed, they can be classified as input ACLs (before the forwarding table) and output ACLs (after the forwarding table).

2.2 Header space analysis

Header space analysis (HSA) [1] is a convenient tool to analyze data plane rules geometrically. In HSA, a packet header h is abstracted as a bitvector, and a globally unique number p represents the port where this packet is presented. Using this notation, the tuple (h, p) fully reflects the state of the packet. We can view (h, p) as a *point* in the *header space* $\mathcal{H}$.

We can define *network transfer functions* T_s with respect to each network device s, which transfer packets from one point to another in the header space. For example, at one time instance, a packet (h, p) is present at s's input port p. Assume that at the next time instance, this packet is forwarded to port p' and its header modified to h'. This can be seen as a transfer function that transforms (h, p) to (h', p'). Mathematically, this can be denoted as

$$T_s(h, p) = (h', p')$$

If a packet is dropped, we consider T undefined on this point.

We can define the domain and the range for a transfer function (with respect to a network device):

Domain: A transfer function's domain (of definition) is the set of points in $\mathcal{H}$ where the function is defined. In other words, the packets represented by these points will be forwarded, rather than dropped, by this network device.

Range: A transfer function's range is the set of points in $\mathcal{H}$, each is the output of the transfer function. The range is all possible outputs by this network device.

The same technique can be used to describe network topology. The *topology transfer function* Γ accepts all packets and does not modify the packet header. A link l connects port p and p' can be formulated as

$$\Gamma(h, p) = (h, p')$$

2.3 Reachability in HSA

Combining the topology transfer function and transfer functions from all routers in a network topology, we can answer the *reachability question*, i.e. what kind of packets (if any) can traverse from a to b?

Define the reachability function R between a and b as:

$$R_{a\to b}(*) = \bigcup_{a\to b \text{ paths}} \{T_n(\Gamma(T_{n-1}(\ldots(\Gamma(T_1(*)\ldots))\}$$

where for each path between a and b, $\{T_1, \ldots, T_{n-1}, T_n\}$ are the transfer functions along the path. The switches in each path are denoted by:

$$a \to S_1 \to ... \to S_{n-1} \to S_n \to b.$$

Let us denote the range of $R_{a\to b}$ as Q, which is the class of headers that can reach b from a in $\mathcal{H}$. To calculate Q, we can iterate $R_{a\to b}$ over all possible inputs. Optionally, we can limit the domain so that we can target a specific class of packets. For example, if we mandate the domain fall in "TCP packets", the same process will answer the question "what kind of TCP packets (if any) can traverse from a to b?"

Notice that these headers are *seen at* b, and not necessarily headers transmitted by a, since headers may change in transit. We are more interested in finding out the domain of $R_{a\to b}$, which are packet headers that can leave a and reach b. By reversing the reachability function (and the transfer functions underlying), the original header sent by a is:

$$Q' = T_1^{-1}(\Gamma(...(T_{n-1}^{-1}(\Gamma(T_n^{-1}(Q)\ldots))$$

using the fact that $\Gamma = \Gamma^{-1}$. Then Q' is the result we are looking for.

3. GENERATING TEST PACKETS

3.1 Overview

From section 2, we know that each router's data plane are controlled by a list of rules. Using HSA, the set of rules of each router can be formulated as a transfer function T (in the case of network topology, Γ). The domain and the range of a transfer function is the result of *all* data plane rules within a network device. By computing the reverse of $R_{a\to b}$'s range, we can get it's domain Q'. These are *all* the packets that can reach from a to b. Q' is a set of *polyhedrons* in $\mathcal{H}$. Each polyhedron is defined by a set of rules along one of the paths from a to b.

From a network testing point of view, by sending *one* sample packet per Q''s polyhedrons, we can test the set of rules that defines each polyhedron. By doing this sampling across all the available terminal ports, we can exercise 100% of "testable" rules from those ports.

The process to generate test packets contains four steps:

1) Collect and parse the data plane information to construct transfer functions. The information includes configuration files, forwarding information base (FIB), access lists (ACLs), and network topology. These information can be collected locally on each device, or remotely (e.g. with SNMP [18]).

2) Define a set of test ports P in the network. During the testing, one can only send or receive packets with these ports. Each port can optionally have an input constraint that restricts the test packets to certain types. For example, some test ports may only accepts test packets to VLAN 74. Other ports may allow for TCP packets only. The whole process will take these user defined constraints into account when generating test packets. This step is usually determined by the network administrator based on high-level requirements, such as where to place the network tester terminals.

3) Run reachability tests between each pair of ports in P, with any input constraints. Collect all reachability results, reverse transfer functions and find Q' polyhedrons. Each polyhedron represents a class of packets that exercise rules in the network. By tracking each step a polyhedron traverses the network, we know the set of rules that it exercises.

4) Pick one packet header per polyhedron. All packet headers in a polyhedron are equivalent in the sense that they exercise the same rules in the network in the same order. In other words, they are treated by the network equivalently.

3.2 Example

To provide intuition, we generate test packets for a small network depicted in Figure 1. In this example, we have three network devices A, B, and C. Note that A is a miniature router, B is a router with output ACL, and C behaves like a Ethernet switch. The rules come from different sources: the routing tables are filled by control plane protocols (e.g. OSPF), ACLs configured by network administrators, and Layer 2 rules are learned by the device itself. We deploy three testers on the edge to send and receive test packets.

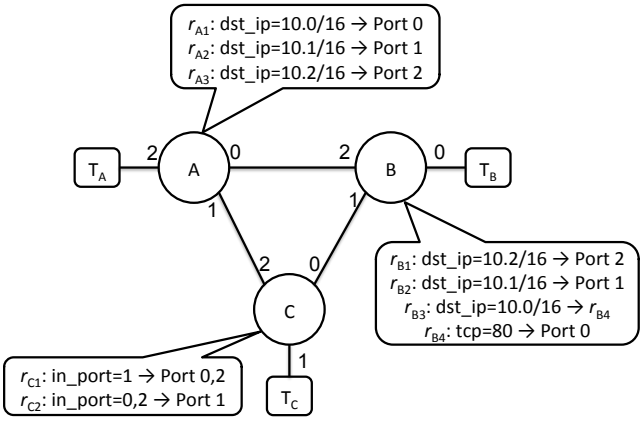

Figure 1: Example topology with three network devices.

After collecting the rule information from the devices, as well as the network topology, all-pairs reachability test leads to the results in Table 1. With that, we can easily track the rules that each test packet exercised. In this example, 6 packets are sufficient to cover all 9 rules.

Note that this small number of test packets is a result of *linear fragmentation* of header space [1], i.e. the number of rule combinations grows roughly linearly (rather than exponentially) with the number of rules. For example, r_{A1} and r_{B1} cannot appear in one combination since they are matched to different IP prefix. Moreover, the routing protocols generally limit the number of paths between any two ports, which further reduces the number of the combinations. We expect this property generally holds in many real networks (it does in Stanford backbone network, as detailed in section 6.

3.3 Guarantees

While simple and intuitive at the first glance, test packets generated by the process described above have some strong guarantees.

GUARANTEE 1. *The set of test packets represents all possible types of tests, under the port and input constraints.*

Explanation: This is backed by the fact that HSA reachability test can find out all classes of packets that traverse from one port to another. Since we collect all-pair reachability results, any packet will fall into one and only one polyhedron and hence can be represented by an equivalent test packet stored in the database.

	Source	Destination	Header	Rules
p_1	T_A	T_B	`dst_ip=10.0/16, tcp=80`	r_{A1}, r_{B3}, r_{B4}, link AB
p_2	T_A	T_C	`dst_ip=10.1/16`	r_{A2}, r_{C2}, link AC
p_3	T_B	T_A	`dst_ip=10.2/16`	r_{B2}, r_{A3}, link AB
p_4	T_B	T_C	`dst_ip=10.1/16`	r_{B2}, r_{C2}, link BC
p_5	T_C	T_A	`dst_ip=10.2/16`	r_{C1}, r_{A3}, link BC
$[p_6]$	T_C	T_B	`dst_ip=10.2/16, tcp=80`	r_{C1}, r_{B3}, r_{B4}, link BC

Table 1: Test packets for the example network depicted in Figure 1. p_6 is redundant.

This guarantee also implies that any other test suite, no matter machine generated or hand-crafted, will be a strict subset of the set we generated. In other words, our set of test packets is *sufficient*.

GUARANTEE 2. *The set of test packets exercise all testable rules, under the port and input constraints.*

Explanation: We define a rule to be testable when at least one packet can exercise the rule, and be forwarded to one of test ports. If a rule is testable, it will produce a resulting region in the range of the reachability function, and a polyhedron in the domain will be discovered. This guarantee means that this set of test packets has highest possible test coverage, since it covers 100% of testable rules.

Not all data plane rules are testable. For example, in an ACL, a lower priority tuple may fully fall in the class described by a higher priority tuple, making it unreachable by any packets. These rules may be installed as a "safety backup", or not get removed due to legacy issues. Misconfiguration is another source of unreachable rules. [1]

GUARANTEE 3. *For any testable rule, the process can find all classes of packets that can exercise it.*

Explanation: Since we know all the polyhedrons and all the rules that can exercise them, we can find every possible packet header, from the available terminal ports that can reach each rule.

3.4 Redundancy removal

We have shown that the set of test packets are sufficient to achieve 100% testable rules in a given network. However, the set is not necessary. In other words, we may use fewer packets in the whole database to achieve the same coverage. This is because multiple test packets may exercise the same rule.

Back to the network we gave in Figure 1 and the test packets we generated in Table 1, p_6 is redundant since r_{C1}, r_{B3}, r_{B4} and link BC have been exercised by the rest of test packets; hence, it does not provide additional information.

Removing redundant test packets can reduce the size of test suite packets sent out periodically. However, it should be noted that the redundancy is useful in fault localization, as shown in section 4.

The necessity problem can be formulated as a Set Cover problem [8].

PROBLEM 1. *Given N test packets $1, 2, \ldots, N$, each corresponding to a set of rules $R_1, R_2, \ldots, R_N$. Pick the set of test packets $k_1, k_2, \ldots, k_M$, so that $R_{k_1} \cup R_{k_2} \cup \ldots \cup R_{k_M} = R_1 \cup R_2 \cup \ldots \cup R_N$.*

When M is minimized, the above problem becomes a well-known Min-Set-Cover question, which corresponds to the minimum number of test packets needed to reach 100% test coverage. The Min-Set-Cover question is known to be NP-Complete. However, a greedy algorithm of complexity $O(N^2)$ can approximate the optimal results, where N is the number of test packets. [8]

4. REALTIME MONITORING AND FAULT LOCALIZATION

In the last section, we discussed an approach to pick the test packets to exercise as much testable rules in the network as possible. In this section we discuss a complementary question: If some test packets fail, how do we pinpoint the error or set of errors that has happened.

First we need to define what a failure means. A failure can have different meaning depending on the type of error. If the test packet is exercising a set of forwarding rules (vs. drop rule), then a failure means that the packet is not delivered to its intended destination. On the other hand, testing a drop rule means that the packet should not be received on any of the end terminals. If the error involves some sort of congestion on an output link, then it will be captured by an increase in travel time of the packet. Also a link or port failure is similar to a forwarding rule failure as a link is represented by a forwarding rule in the topology transfer function.

When an error is detected at any of the end terminals, we know that one of the rules exercised by that test packet is in error. The questions is how do we find out exactly which rule is causing problem? We do the fault

[1] A special case of non-testable rules are drop rules. We can define a "blackhole" port to "forward" all dropped packets to. If the black hole has some kind of visibility, for example, the packet headers are logged, then the rules become testable.

localization in 3 steps:

1. Each passing test packet indicates that its corresponding traversed rules are working correctly. Therefore if call the set of all rules exercised by the passing test packets P, then we know for sure that rules in P are correct. If we similarly define all rules traversed by failing test packets F, then one or more rules in F might be in error. Therefore $F - P$ is the *suspect rule set*.

2. Now we want to make the suspect set is as small as possible. Remember in 3.4, from the possible set of N distinct test packets, we picked the minimum M test packets to exercise all the rules. Let's call the remaining unused $N - M$ test packets the *reserved packets*. We can use these reserved packets to narrow down the suspect set: From the reserved packets, find those that exercise only one rule from the suspect set and send them out [2]. 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 new suspect set [3].

3. For most practical cases, at this step we already have a small enough suspect set that we can stop here and report all the remaining rules. 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. If these test packets pass, it shows that those rules are not in error and we can remove them from the suspect set. But, a failing test packet doesn't give us any more information. (Hence there is no benefit in sending a test packet which exercises a rule that is already know to be in error at step 2)

It might be the case that two or more rules are indistinguishable, i.e. if a packet exercises one rule, it will exercise the other rule as well. Therefore it is not always possible to pinpoint the exact error.

5. IMPLEMENTATION

Our prototype implementation consists of an offline test packet generator and an online network monitor. Figure 2 shows the block diagram of the system.

5.1 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 command line interface (Cisco IOS) or XML files (Junos). The generator then uses Hassel [2], an HSA library, to construct transfer functions.

All-pairs reachability checking is conducted in parallel using the `Multiprocess` module shipped with Python. Each process is responsible for a subset of test ports as source ports and independently checks their reachability to all other ports. After reachability tests are completed, each process execute first pass Min-Set-Cover algorithm ("local compression"), to reduce the size of the results before returning to the master process. And then results are collected and the master process execute second pass Min-Set-Cover ("global compression"). On each stage, results including polyhedrons and corresponding rules are stored in a centralized SQLite database.

The whole test packet generation process can be called periodically or triggered manually by network administrators if necessary.

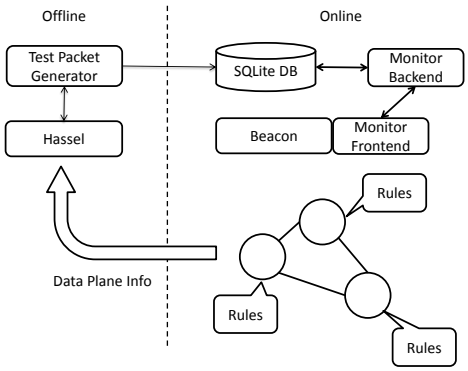

Figure 2: The prototype system includes an offline test packet generator, and an online network monitor.

5.2 Network monitor

The online network monitor is divided into a frontend that coordinates tests and collects the results, and a backend that analyzes the test reports and locates the faults if any.

In our prototype, the frontend is built on Beacon [17], a Java-based network controller designed to manage OpenFlow [6] networks. OpenFlow is a remote control protocol for network devices. We use OpenFlow's `packet-in` and `packet-out` messages to inject test packets into the network and collect test results via OpenFlow-enabled devices. The frontend reads the database, sends out test packets, and collect the results periodically (e.g. every 3-5s). Note that although

[2]It can exercise many rules that are not in the suspect set, but only one rule from the suspect set

[3]Note that we don't remove the already-known failed rules from the suspect set

OpenFlow is originally designed for switching devices, endhosts such as desktops and laptops can also communicate with Beacon in OpenFlow so that they can participate in testing. Each test packet is tagged using IPv4's Class of Service (CoS) field so that they can be demultiplexed from normal traffic.

The Python-based backend implements the fault localization algorithm. When part of the tests are failing, the front end sends the failing test packets to the backend, which then selects another set of test packets from the reserved packet set and sends them back to the front end. The front end sends out the new set of test packets and collects the result again. This exchange iterates 2-3 times, depending on the user settings. Then backend will conclude and generate report.

The communication between the frontend and the backend is conducted through JSON. Because the two components share the same database, instead of exchanging the full packet header, they can merely exchange the index of test packet within the database.

6. EVALUATION

In this section, we first demonstrate the functionality of our test packet generator on the backbone network of Stanford University. We benchmark the performance of the generation and report both the statistics of test packets generated and the untestable rules in this production network. In the second part, we replicate the Stanford network in Mininet [7], a container-based network emulator, in which we use our online network monitor to send and receive test packets. We manually inject different types of errors into this emulated network evaluate the monitor's ability to discover these errors.

6.1 The Stanford network

With a population of over 15,000 students, 2,000 faculty, and five /16 IPv4 subnets, Stanford is a sizable enterprise network. Figure X shows the network that connects departments and student dorms to the outside word. There are 14 operational zone (OZ) routers at the bottom connected via 10 switches to 2 backbone routers which connect Stanford to the outside world. Overall, the network has more than 757,000 forwarding entries and 1,500 ACL rules. The data plane configuration was collected through command line interfaces. We do not provide exact IP addresses or ports to meet privacy concerns.

6.2 Test packet generation

We generate test packets for the Stanford backbone based on no port and input constraints. Hence, the test packets of any forms can be injected from any ports. The result represents the upper bound of the size of generated packets, as well as all testable rules in the data plane.

Task	Approx. runtime
Parse configuration files	20s
Generate transfer functions	30s
All-pairs reachability check	1600s

Table 2: Runtime of each stage in test packet generation for Stanford backbone network.

We have 3 experimental goals: first, we measure our prototype's performance in this real network in terms of runtime; second, we wish to understand the size of total test packets generated, as well as monitoring test packets needed; finally, a byproduct of our tools is that we are able to discover all unreachable rules - collecting and understanding why they exist in the data plane helps network administrators to better plan their networks.

6.2.1 Performance

The tool runs on a quad-core Intel Core i7 CPU at 2.67GHz with 6GB memory. Thanks to the off-the-shelf `multiprocess` module in Python, the generation is in 8-way parallel, which takes about 30 minutes to generate all test packets for the Stanford backbone. Table 2 summarizes the time consumed at each stage. We can see the dominant portion of runtime comes from the reachability checking.

Note that the only parallelism we have utilized so far is on the port level, i.e. the reachability from each port to all other ports is evaluated in a separate process. There are several other potential parallelism that can be explored. For example, when evaluating the result of a transfer function, each rule within the function can be evaluated independently.

6.2.2 Packets and rules

Table Y shows the statistics of packets generated against Stanford network. Before local compression and global compression, the result of all-pairs reachability check lead to 4 million test packets. Consider 60 byte as a typical test packet size, the set of test packets can be translated to only 240MB. This means that in a Gigabit Ethernet, the time it takes to send out all test packets is less than a second. The time is further amortized by the number of test ports available in the network. After two stage of compression, the size of test packets can be reduced to around 4 thousands, with 3 order of magnitude of compression. This is small enough to allow the network monitor to repeat the testing every *microsecond*!

The surprisingly small size of the test suite is a result of the linear fragmentation of rules. Recall that each test packet corresponds to a combination of rules. While the number of combinations can grow exponentially in adversarial settings, we argue that in real networks, the

number of fragments scales linearly with the number of rules. The linear fragmentation nature of Stanford network has been shown in [1].

The length of rules traversed by a test packet implies the efficiency of Min-Set-Cover algorithm. Similarly, for each rule, we can count the number of test packets that exercise it. We found that most of packets exercise 3-6 rules, and each rule is exercised by 2-4 packets. The slight redundancy is useful during fault localization.

6.2.3 *Untestable rules*

In our evaluation, a network device is formulated as three stages - an input ACL table that filters packets immediately after they enter the device, a forwarding table that routes the packet to its output port, and finally an output ACL stage that filters packets again before they leave the device. When no input ACL or output ACL is not configured for a port, a default rule is added to simply accepts all packets unmodified. Each stage consists a ordered list of rules that are executed by test packets.

We find that among all rules, less than 10% are not testable by any packets. In other words, they are not exercisable by *any* packets injected from *any* ports in the network. Here are some examples we discovered using our test packet generator.

```
remark CSDCF: Allow in CS nets
remark CSDCF: All of campus
permit ip 10.0.0.0 0.3.255.255 any
permit ip 10.1.0.0 0.3.255.255 any
...
remark ### Four lines for the XXX project
permit tcp host 10.0.65.21 host 10.0.78.203 eq 22
permit tcp host 10.1.76.233 host 10.0.78.203 eq 22
permit tcp host 10.0.65.21 host 10.0.78.204 eq 22
permit tcp host 10.1.76.233 host 10.0.78.204 eq 22
```

Figure 3: Unreachable ACL: the four lines for the specific project are already included in the more general rules, hence will never be executed. IP addresses anonymized.

Unreachable ACL: An ACL is an ordered list of rules containing packet header definitions and actions. Some rules are not testable because their matching conditions are subset of ones of the higher order rules. For example, in Figure 3, there are four lines of rules to allow SSH access between four machines for a specific project. However, these four machines all belong to the CS department subnet, and are already allowed to communicate with each other. These rules are likely to be "backup rules" to ensure the continued connectivity in case the primary rules are removed; or they can be historical rules installed in the past when machines within the subnet were not allowed to communicate freely.

```
interface GigabitEthernet3/8
 description psg-tapsw 2/0/26
 switchport
 switchport trunk encapsulation dot1q
 switchport trunk allowed vlan none
 switchport mode trunk
```

Figure 4: Unused port: this switch port is configured as a VLAN port. However, no VLAN is allowed to use it.

Unused ports: Figure 4 shows a switch port configured as a VLAN port, with no VLAN allowed to use this port. Hence, this port is unreachable by any packets and hence should have been "shut down". This is situation is not uncommon among enterprise networks, data centers, and even in backbone networks. [11,12] It can result from a bootstrapping process where ports are used temporary for testing, or a rewiring where a link is torn down but the port was not shut down.

Mismatched ACL and forwarding rule: While the first two types of untestable rules are constrained in a single stage, this type of untestable rule results from the mismatched configuration between stages. In one router, we find that the input ACL explicitly denies a certain subnet from entering the device, whereas the subnet prefix appears in the forwarding table. In this case, the rule in the forwarding table will never be exercised. This situation could cause by the inconsistent configurations between the control plane protocol (OSPF or BGP) and the data plane packet filtering.

7. RELATED WORK

Understanding network failures with data mining: Data mining through system logs is a natural way to characterize the network failures. Markopoulou et al. [9] analyze Sprint backbone network's IS-IS routing update sequence to discover failures that affect the IP connectivity. Le et al. [10] apply data mining to router configurations to detect errors in user accounts, interfaces, and BGP sessions. Turner et al. [11] include "low-quality" data sources, such as router configurations, syslogs, and semi-structured data such as email logs in administrator's mailing list to recreate a history of failure events in CENIC network and Internet2. On data centers, a similar study conducted by Gill et al. [12] analyzes SNMP/syslog, NOC tickets and traffic data of Microsoft's data centers.

While producing convincing results, these data mining approaches commonly suffers data consistency problems, making (sometimes manual and difficult) "data cleaning" necessary to remove the bias. This is primarily due to the logs are not designed to understand and track

the network failures. For example, it is observed [12] that many devices may emit multiple "down" events simultaneously. With formal network testing, the test cases are designed to cover all rules, hence the logs of testing results are sufficient to detect and reproduce the network failures.

Network monitoring and fault localization: Monitoring based on active measurements has been receiving researchers' attention for a long time, and many approaches to develop a measurement-friendly architecture within the Internet are proposed. [13, 16] On a router basis, it is also suggested that they should be able to sample the packets for measurement. [14] mPlane [15] breaks end-to-end paths into segments and conduct measurements on a per-segment basis, so that segment-level measurements can be at lower frequency with higher fidelity.

Our approach is complementary to these proposals: formal network testing does not dictate the locations of injecting network probes and how these probes should be constructed. By incorporating input and port constraints, formal network testing can generate test packets and injection points using arbitrary existing deployment of measurement devices (whether standalone or embedded in routers).

8. CONCLUSION

Our paper introduces *Formal Network Testing*: a general framework to reason about the correctness of network data plane, both online and offline. By answering the basic reachability quesition in the computer network – Can A talk to B? – we show how to generate a test suite that guarantees the highest possible rule coverage, discovers redundant rules in the network, and locates the failures when they happen. Our approach gives network operators confidence that the goals and requirements are continuously met during the operation; and the abnormality can be detected and located in a timely fashion.

9. REFERENCES

[1] P. Kazemian, G. Varghese, N. McKeown, *Header Space Analysis: Static Checking For Networks*, In NSDI 2012.

[2] P. Kazemian, J. H. Zeng, G. Varghese, N. McKeown, *Header Space Library (Hassel)*, Repository at: git clone https://bitbucket.org/peymank/hassel-public.git

[3] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, S. T. King, *Debugging the data plane with anteater* In SIGCOMM. 2011.

[4] N. Feamster, H. Balakrishnan, *Detecting BGP configuration faults with static analysis*, In NSDI. 2005.

[5] F. Le, G. Xie, D. Pei, J. Wang, and H. Zhang, *Shedding Light on the Glue Logic of the Internet Routing Architecture*, In SIGCOMM. 2008.

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, *OpenFlow: Enabling Innovation in Campus Networks*, In ACM SIGCOMM Computer Communication Review, Volume 38, Number 2, 2008.

[7] B. Lantz, B. Heller, N. McKeown, *A Network in a Laptop: Rapid Prototyping for Software-Defined Networks*, In HotNets-IX.

[8] V. V. Vazirani, *Approximation Algorithms*, Springer, 2001.

[9] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C. Chuah, , C. Diot, *Characterization of Failures in an IP Backbone*, In INFOCOM 2004.

[10] F. Le, S. Lee, T. Wong, H. Kim, D. Newcomb, *Minerals: using data mining to detect router misconfigurations*, In MineNet 2006.

[11] D. Turner, K. Levchenko, A. C. Snoeren, S. Savage, *California Fault Lines: Understanding the Causes and Impact of Network Failures*, In SIGCOMM 2010.

[12] P. gill, N. Jain, N. Nagappan, *Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications*, In SIGCOMM 2011.

[13] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, A. Venkataramani, *iPlane: an information plane for distributed services* In OSDI 2006.

[14] N. G. Duffield, M. Grossglauser, *Trajectory Sampling for Direct Traffic Observation*, In IEEE/ACM transaction on Networking, June 2001.

[15] R. R. Kompella, A C. Snoeren, G. Varghese, *mPlane: An Architecture for Scalable Fault Localization*, In ACM ReARCH 2009.

[16] S. Machiraju, D. Veitch *A Measurement-Friendly Network (MFN) Architecture*, In INM 2006.

[17] Beacon Openflow Controller, https://openflow.stanford.edu/display/Beacon/Home

[18] RFC 3411, `http://www.ietf.org/rfc/rfc3411.txt`