AUTOMATIC DATA PLANE TESTING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Hongyi Zeng
February 2014

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Nick McKeown, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Sachin Katti

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

George Varghese

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Today's networks require much human intervention to keep them working. Every day network engineers wrestle with router misconfigurations, fiber cuts, faulty interfaces, mislabeled cables, software bugs, intermittent links and a myriad other issues that cause networks to misbehave, or fail completely. Network engineers hunt down bugs using the most rudimentary tools (e.g., `ping`, `traceroute`, SNMP, and `tcpdump`) and track down root causes using a combination of accrued wisdom and intuition. Debugging networks is only becoming harder as networks are getting bigger and more complicated. As networks keep growing, there is considerable interest in *automating* control, error-reporting, troubleshooting and debugging.

We found that many network problems are associated with data plane behaviors, i.e., how the network transports data plane packets. This dissertation discusses the design and implementation of automatic data plane testing tools under various network scenarios.

We first present Automatic Test Packet Generation (ATPG), a foundation framework of data plane testing when all data plane information is available and accurate. ATPG reads router configurations and generates a device-independent model. The model is used to generate a minimum set of test packets to (minimally) exercise every link in the network or (maximally) exercise every rule in the network. Test packets are sent periodically, and detected failures trigger a separate mechanism to localize the fault. ATPG works well for small to medium size networks that use simple, deterministic routing. We describe our prototype ATPG implementation and results on two real-world data sets: Stanford University's backbone network and Internet2. We find that a small number of test packets suffices to test all rules in these networks: For example, 4,000 packets can

cover all rules in Stanford's backbone network, while 54 are enough to cover all links. Sending 4,000 test packets 10 times per second consumes less than 1% of link capacity.

NetSonar extends ATPG by allowing incomplete or inaccurate data plane information as inputs. Earlier test techniques were either white box (assumes complete forwarding knowledge) or black box (assumes no knowledge). We argue that the former is infeasible in large networks, and the latter is inefficient and incomplete. NetSonar is the first *gray box* tester for networks and the first tester deployed in a production network. NetSonar uses only coarse forwarding information and does not require knowledge of load balancing hash functions. It deals with non-determinism by computing probabilistic path covers. In addition, NetSonar allows accurate fault localization and minimizes test overhead with diagnosable link covers. We describe our experience deploying NetSonar in a global inter-DC network. From Feb to March 2013, NetSonar triggered 66 inter-DC alerts, of which 56 were independently verified. In April 2013, the number of alerts based on router counters would have been 87 times higher than what NetSonar reported.

Finally, we move our focus to data center networks with thousands of switches and millions of forwarding table entries. Data center owners use static analysis tools that examine the topology and forwarding tables to check for loops, black-holes and reachability failures. However, the existing tools do not scale to a large data center network. Moreover, no existing tool addresses the problem of potential false positives when analyzing a network snapshot, when the network state is constantly in flux. We present Libra, a tool for verifying forwarding tables in large data center networks that simplifies ATPG's data plane model to significantly improve verification performance. Libra takes a stable snapshot across the entire network and divides the data plane verification problem into multiple independent tasks by *slicing* forwarding tables, and analyzing each slice on a different process. Libra can be implemented on MapReduce, a general parallel computing framework. We show that Libra can verify a large data center network with 10,000 switches in less than 1 minute.

Acknowledgments

My advisor, Prof. Nick McKeown, taught me what research means. He always encouraged me to be bold, to strive for perfection, and to think big. During my PhD, I was very fortunate to witness SDN evolve from a lab project to a disruptive movement across the network industry. Nick has demonstrated, by example, how research can change the world. This was the most precious gift a PhD student could ask for – the determination to make a real-world impact.

Prof. George Varghese taught me to stay curious. His broad knowledge on almost every aspect of computer science often brings fresh, exciting ideas to our numerous discussions. George introduced me to people from different backgrounds and inspired me to think about problems in an interdisciplinary way. His optimism and enthusiasm helped me overcome difficulties and obstacles in research and finally graduate with self-confidence.

I would like to thank other members of my oral exam and reading committees: Prof. Sachin Katti, Prof. David Dill, and Prof. John Ousterhout. Your questions, comments, and feedback help improve the final version of this dissertation. I valued the discussion with you.

The works described in this dissertation would not be possible without my collaborators: Peyman Kazemian and Vimalkumar Jeyakumar from Stanford; Ming Zhang, Ratul Mahajan, Lihua Yuan from Microsoft; Amin Vahdat, Shidong Zhang, Fei Ye, Mickey Ju, Junda Liu from Google. I am very grateful to have met and worked with many talented people during my back-to-back internships from October 2012 to September 2013 at Microsoft and Google. I also feel very privileged to have experienced production deployment of many parts of my research. Special thanks to Peyman Kazemian, who carefully

reviewed an early draft of this dissertation and provided many constructive comments and suggestions.

The NetFPGA team accompanied me through initial years of my PhD: Adam Covington, Jonathan Ellithorpe, Tatsuya Yabe, Glen Gibb, Jad Naous from Stanford; Andrew Moore, Muhammad Shahbaz, David Miller from University of Cambridge; Michaela Blott, Paul Hartke, Kees Vissers from Xilinx. Together we have built the most popular open source platform for network research and education. Putting open source tools in people's hands and seeing cool projects come out has been amazing.

I want to thank the rest of the McKeown group, past and present: Neda Beheshti, David Erickson, Nikhil Handigol, Brandon Heller, Te-Yuan Huang, Masayoshi Kobayashi, Rob Sherwood, Kok-Kiong Yap, and Yiannis Yiakoumis. Thank you for helping me refine my publication and presentation, inspiring me with new ideas, and giving me so many fun memories.

My parents, Huaqun Zeng and Qinglan Lin, have always supported me unconditionally in every possible way, even though they are on the other side of the ocean and know little about computer science. This dissertation is my gift to them, complementing our small collection of doctors (beside Doctor of Judicial Science and Doctor of Medicine). I would also like to thank my other family members, including Jingui Lin, Guochen Zeng, Xiuye Chen, Jinzhi Lin, Lianying Zhang, my parents in-law Junsheng Huang and Lijing Su.

Lastly, I want to thank my wife, Chuan Huang. Her encouragement and caring took me through all the ups and downs in my PhD study. Thank you for always standing by me with endless love and being my strongest motivation to move forward.

To my parents, Huaqun Zeng and Qinglan Lin

&

To my wife, Chuan Huang

Contents

List of Tables

List of Figures

Chapter 1

Introduction

"My dear friend Copperfield," said Mr. Micawber, "accidents will occur in the best-regulated families."

Charles Dickens (1812-1870)

OPERATING a modern network is no easy task. Every day network engineers have to wrestle with misconfigured routers, fiber cuts, faulty interfaces, mislabeled cables, software bugs, intermittent links and a myriad other reasons that cause networks to misbehave, or fail completely. Network engineers hunt down bugs using the most rudimentary tools (e.g., `ping`, `traceroute`, SNMP, and `tcpdump`), and track down root causes using a combination of accrued wisdom and intuition. Debugging networks is only becoming harder as networks are getting *bigger* (modern data centers may contain 10,000 switches, a campus network may serve 50,000 users, a 100Gb/s long-haul link may carry 100,000 flows) and getting *more complicated* (with over 6,000 RFCs, router software is based on millions of lines of source code, and network chips often contain billions of gates). Small wonder that network engineers have been labeled "masters of complexity" [69].

Troubleshooting a network is difficult for good 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 via the Command Line Interface (CLI). Third, there are many different programs, protocols, and humans updating the forwarding state simultaneously.

Facing this hard problem, network engineers deserve better tools than `ping` and `traceroute`. In fact, in other fields of engineering testing tools have been evolving for a long time. For example, both the ASIC and software design industries are buttressed by billion-dollar tool businesses that supply techniques for both static (e.g., design rule) and dynamic (e.g., timing) verification.

1.1 The Goal

Modern computer networks can be divided into the data plane and the control plane. The *data plane* consists of a number of interconnected switches, each contains forwarding rules that determine the flow of packets. For example, the forwarding rule in an Ethernet switch looks at a packet's destination MAC address, and decides its next port. On top of the data plane is the *control plane* that runs routing protocols such as OSPF or BGP. The control plane populates the data plane with forwarding rules based on its global network knowledge.

The goal of this dissertation is to develop automatic tools that help network engineers test and debug the *data plane*. Some representative problems that we hope these tools will answer include:

- Can host A talk to host B?
- Host A cannot talk to host B, so where the packets are dropped?
- Do we have correct rules across the network to prevent A from talking to B?
- Are there forwarding loops in my network?

- The application owners are complaining about increased latency. What causes the latency spike – the router, link, or host?
- How do I know my router configuration correctly pushed down to the forwarding ASIC in the router?

Specifically, the goal of data plane testing includes two parts: 1) verifying forwarding rules correctness given topology and policy (e.g., the back-end database cannot talk to the front-end web server directly); 2) verifying network performance given the service level agreement, which is a contract between the network service provider and its users. For example, the latency between point A and point B should not be larger than 5 ms.

Note that we restrict ourselves to focusing on *data plane problems*, i.e., problems where the network is incorrectly transporting data plane packets. We are not attempting to verify the correctness of routing processes such as BGP or OSPF. Neither do we attempt to answer "why" the data plane is in a certain incorrect state. We focus on data plane problems for the following reasons:

1. Any network problems should result in data plane problems. If a problem does not affect the data plane, it is invisible to the application that uses the network.
2. Correctly identifying and diagnosing data plane problems help in troubleshooting control plane problems. For example, the network may drop certain classes of packets due to a lack of corresponding forwarding rules. If we know some rules are missing, we may trace them back to the control plane, e.g., a BGP misconfiguration.
3. Data plane testing can detect performance and hardware failures, which cannot be detected by analyzing the control logic, as these failures could occur even if the control logic is working perfectly.

Before setting out to build these tools, we decided to step back and try to understand the problems network engineers encounter, and how they currently troubleshoot them. As we will see in Section 1.2, the answers to the data plane questions directly correspond to many types of network problems existing today.

Category	Avg	% of ≥ 4
Reachability Failure	3.67	56.90%
Throughput/Latency	3.39	52.54%
Intermittent Connectivity	3.38	53.45%
Router CPU High Utilization	2.87	31.67%
Congestion	2.65	28.07%
Security Policy Violation	2.33	17.54%
Forwarding Loop	1.89	10.71%
Broadcast/Multicast Storm	1.83	9.62%

(a) Symptoms of network failure.

Category	Avg	% of ≥ 4
Switch/Router Software Bug	3.12	40.35%
Hardware Failure	3.07	41.07%
External	3.06	42.37%
Attack	2.67	29.82%
ACL Misconfig.	2.44	20.00%
Software Upgrade	2.35	18.52%
Protocol Misconfiguration	2.29	23.64%
Unknown	2.25	17.65%
Host Network Stack Bug	1.98	16.00%
QoS/TE Misconfig.	1.70	7.41%

(b) Causes of network failure.

Table 1.1: Ranking of symptoms and causes reported by administrators (5=most often, 1=least often). The right column shows the percentage who reported ≥ 4.

1.2 Network Troubleshooting Today

1.2.1 A survey of network engineers

We invited subscribers to the NANOG[1] mailing list to complete a survey in May-June 2012. Of the 61 who responded, 12 administer small networks (< 1k hosts), 23 medium networks (1k-10k hosts), 11 large networks (10k-100k hosts) and 12 very large networks (> 100k hosts). All responses (anonymized) are reported in [72], and are summarized in Table 1.1. The most relevant findings are:

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

Causes: The two most common symptoms (switch and router software bugs and hardware failures) are best found by dynamic testing.

[1] North American Network Operators' Group.

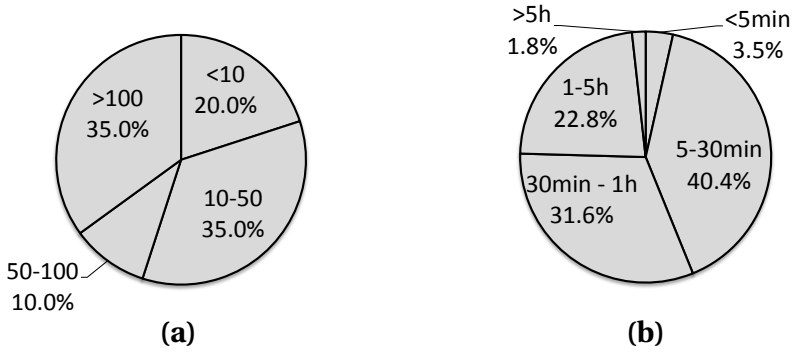


Figure 1.1: Reported number of network related tickets generated per month (a) and time to resolve a ticket (b).

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

Table 1.2: Tools used by network administrators (5=most often, 1=least often).

Cost of troubleshooting: Two metrics capture the cost of network debugging - the number of network-related tickets per month and the average time consumed to resolve a ticket (Figure 1.1). 35% of networks generate more than 100 tickets per month. 40.4% of respondents estimate it takes under 30 minutes to resolve a ticket, but 24.6% report that it takes over an hour on average.

Tools: Table 1.2 shows that `ping`, `traceroute`, and SNMP are by far the most popular tools. When asked what the ideal tool for network debugging would be, 70.7% reported a desire for automatic test generation to check performance and correctness. Some added a desire for "long running tests to detect jitter or intermittent issues", "real-time link capacity monitoring", and "monitoring tools for network state".

In summary, while our survey is small, it supports the hypothesis that network administrators face complicated symptoms and causes. The cost of debugging is nontrivial, due to the frequency of problems and the time to solve these problems, and while classical tools such as `ping` and `traceroute` are still heavily used, administrators desire

more sophisticated tools.

1.3 Data Plane Testing

The unfortunate realities of network operation make automatic, systematic data plane troubleshooting a necessity. However, there is more than one way to approach this problem. Moreover, different networks may call for different approaches. Any data plane tester design should answer the following three questions:

- **Method**: Do we only read and analyze forwarding tables (static analysis), or do we actually send out test packets to observe the network's behavior (dynamic analysis)?
- **Knowledge**: How much we know about the network under test? Do we know all the forwarding tables and topology, just part of them, or none of them?
- **Coverage**: Which network components do we cover, links or rules? How do we achieve 100% coverage? Is that even possible?

These questions are not unique to network testing. We can see persuasive analogies in other engineering disciplines: in digital circuit design, either the high-level hardware design language (HDL), such as Verilog/VHDL, or the low-level circuit schematic defines the behavior of the circuit, rather like to the role of forwarding rules and topology in data plane. A static tester can inspect the HDL, the schematic, or the circuit layout to ensure the all design requirements and rules are met. A dynamic tester - testbench - generates test vectors to exercise all the internal gates in the circuit. The knowledge assumed varies with the testing - sometimes the tester knows the design of the circuit, but in other cases the tester regards the device under test as a "black box".

We can think of the data plane as a "network program" as well. In the classic IPv4 network, for example, each switch is a *function* that takes the packet's header as input. Inside the function there is a list of `if-else` statements, which contains all forwarding rules in the switch's forwarding table. When there is a match found, the function returns the output port. The main program, encoded with the network topology, inspects the

	Digital Circuit	Program	Data Plane
Static Testing	Functional simulation	Compile-time	Libra
Dynamic Testing	Hardware testbench	Run-time	ATPG, NetSonar
Knowledge	HDL/Schematic	Source Code	Rules + Topology
Coverage	Gates/Flip-flops	Execution paths	Rules/links/interfaces

Table 1.3: Comparison among digital circuit testing, program testing, and data plane testing. ATPG, NetSonar and Libra is discussed in the following chapters respectively.

output port and calls another switch function as appropriate until the packet is delivered to an edge-facing port (or dropped). To test the program, the three questions boil down to whether we "execute" the program, whether we know the "source code", and how much coverage the tester can achieve. Table 1.3 compares digital circuit design and software testing to data plane testing.

1.3.1 Method

Static program analysis inspects programs *without* executing them, i.e., compile-time checking. This is based on the assumption that the source code (in our case, the forwarding rules and topology) contains all the information to deduce the behavior during execution. Formal methods, such as model checking and symbolic execution, has been successfully used in the software and hardware design industry. Recently, we have also seen these methods being used in verifying network correctness [13, 49].

Unfortunately, networks are more complex than programs. There are many "run-time" problems that cannot be predicted by simply inspecting the forwarding rules. For example, static analysis cannot detect performance problems such as congestion and packet loss. Moreover, unexpected incidents, such as fiber cuts, table overflow, or device failures will be missed. These problems requires dynamic analysis, i.e., executing the "network program" with real input (test packets). This is usually done by deploying a set of *test agents* on the edge of networks, which sends carefully-designed test packets through the network. The delivered test packets are timestamped and aggregated to draw some conclusions about the network state.

1.3.2 Knowledge

Conventional *black box* testers, such as `ping` and `traceroute`, assume no knowledge about the network, except that switches and hosts have an IP-compliant network stack that will respond to ICMP packets. This minimal set of requirements contributes to these tools' popularity, but also limits their capability. They share the shortcomings of any black box tester – lack of scalability and inability to reason about coverage. To effectively test a large network, many test agents are needed, which leads to an intractably large number of probes per test interval. Even then, it is difficult to determine what fraction of links and routers were covered by the probes.

To overcome these shortcomings, researchers have proposed the equivalent of *white box* testing [8, 23, 70]. These techniques need detailed information about the network such as the entire forwarding information base (FIB) from each router, which describes how a router processes a given packet. Using this information, they compute a subset of all possible probes that will cover all the routers and links in the network. Many of these tools requires using `traceroute` to bootstrap the testing process or reading FIBs directly.

However, in a very large network, white-box testing needs to gather a large amount of information. Fundamentally, the techniques assume that the FIB state is a consistent snapshot across all routers, which is hard to obtain in a large network with high churn. Furthermore, today's router CPUs tend to be underpowered, and it is generally not possible to read the entire FIB state frequently. In this case, testers may have to rely on partial, coarse information for diagnosis, and can thus be classified as *gray box* testers.

1.3.3 Coverage

A tester should achieve high test coverage, meaning that all rules, links, and interfaces in the network should be tested. A static tester can potentially achieve 100% test coverage by inspecting all forwarding rules.

By contrast, there is no guarantee that a dynamic tester can exercise all rules. For example, a "backup" rule that takes affect only when the primary rule is deleted is not exercisable by definition. The placement of test agents also affects the coverage. Ideally,

	ATPG	NetSonar	Libra
Target Networks	Enterprise	Backbone	Data Center
# of Devices	20	100	10,000
Method	Dynamic	Dynamic	Static
Knowledge	White box	Gray box	White box
Coverage	Rules + Links	Links (Probabilistic)	Rules

Table 1.4: Tools proposed in this dissertation.

the test agents should be deployed across all the edge ports, but this goal is sometimes impractical for management reasons. A partially-deployed set of test agents may not reach certain areas of the network.

Another issue when striving for high coverage in the dynamic context is testing overhead. Recall that dynamic testing sends out packets to uncover run-time problems. A naive all-pairs test, while achieving highest possible coverage, scales to $O(N^2)$ where N is the number of test agents. We show in Section 2.2.1 that a Min-Set-Cover algorithm can select a subset of N agents and still achieve the same coverage.

The coverage problem is further complicated by multipath routing (e.g., ECMP). With ECMP, the FIB does not fully describe how a given packet will be forwarded; the outgoing link of a packet is decided by hashing packet headers (usually the 5-tuple). Today, the hash function is often not known to the network operator, so a gray box dynamic tester has to handle this uncertainty.

1.4 Organization

The three systems proposed in this dissertation - ATPG, NetSonar, and Libra - are summarized in Table 1.4. We will discuss them in the following chapters:

Chapter 2 describes Automatic Test Packet Generation (ATPG), a white box dynamic tester that 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. We show that ATPG itself works well in networks of tens of switches with complex routing and

access control. The content of this chapter was originally published in [78].

Chapter 3 discusses NetSonar, which extends the idea of exhaustive testing in ATPG to MSN, a global production backbone network. NetSonar has to handle many uncertainties that occur modern large scale networks such as unknown hash functions in multipath routing and constantly changing paths. These imperfections arise due to both the unfortunate reality of less than ideal router design, and the intrinsic nature of large networks. Chapter 3 introduces probabilistic path cover and diagnosable link cover to tackle uncertainty algorithmically and achieve high coverage.

Chapter 4 proposes Libra, a static tester designed for immense data center networks with tens of thousands of switches. Libra addresses two challenges that other static checkers have not yet dealt with because of scale issues: how to take a stable snapshot across a large number of devices; how to verify data plane properties, such as loop freedom, from a large number of forwarding tables collected. Libra greatly simplifies ATPG's data plane model and leverages the nature of data center networks, so it can handle networks with tens of thousands of switches. The content of this chapter was originally published in [79].

Finally, Chapter 5 summarizes the dissertation and main contributions and suggests several potential future directions for study.

Chapter 2

Whitebox Data Plane Testing

Only strong trees stand the test of a storm.

Chinese Idiom

LET us start by solving the simple white box, dynamic testing problem:

> Given all forwarding rules and network topology, what is the minimum set of test packets that can cover 100% of rules, links and interfaces? Moreover, how to use this set to localize and diagnose data plane problems?

This kind of problems occur frequently in today's network management. Consider two examples:

Example 1. Suppose a router with a faulty line card starts dropping packets silently. Alice, who administers 100 routers, receives a ticket from several unhappy users complaining about connectivity. First, Alice examines each router to see if the configuration was changed recently, and concludes that the configuration was untouched. Next, Alice uses her knowledge of the topology to triangulate the faulty device with `ping` and `traceroute`. Finally, she calls a colleague to replace the line card.

Example 2. Suppose that video traffic is mapped to a specific queue in a router, but

packets are dropped because the token bucket rate is too low. It is not at all clear how Alice can track down such a *performance fault* using `ping` and `traceroute`.

Automatic Test Packet Generation (ATPG) is one answer to this dynamic testing problem: ATPG *automatically* generates a minimal set of packets to test the liveness of the underlying topology *and* the congruence between data plane state and configuration specifications. The tool can also automatically generate packets to test *performance* assertions such as packet latency. In Example 1 instead of Alice manually deciding which `ping` packets to send, the tool does so periodically on her behalf. In Example 2, the tool determines that it must send packets with certain headers to "exercise" the video queue, and then determines that these packets are being dropped.

ATPG detects and diagnoses errors by independently and exhaustively *testing* all forwarding entries, firewall rules, and any packet processing rules in the network. In ATPG, test packets are generated algorithmically from the device configuration files and FIBs, with the minimum number of packets required for complete coverage. Test packets are fed into the network so that every rule is exercised directly from the data plane. Since ATPG treats links just like normal forwarding rules, its full coverage guarantees testing of every link in the network. It can also be specialized to generate a minimal set of packets that merely test every link for network liveness. At least in this basic form, we feel that ATPG or some similar technique is fundamental to networks: Instead of *reacting* to failures, many network operators such as Internet2 [33] *proactively* check the health of their network using pings between all pairs of sources. However all-pairs `ping` does not guarantee testing of all links, and has been found to be unscalable for large networks such as PlanetLab [62].

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 health care network may dedicate more test packets to Firewall rules to ensure HIPPA compliance.

We tested our method on two real world data sets - the backbone networks of Stanford University and Internet2, representing an enterprise network and a nationwide ISP. The results are encouraging: thanks to the structure of real world rulesets, the number of test packets needed is surprisingly small. For the Stanford network with over 757,000 rules and more than 100 VLANs, we only need 4,000 packets to exercise all forwarding rules and ACLs. On Internet2, 35,000 packets suffice to exercise all IPv4 forwarding rules. Put another way, we can check every rule in every router on the Stanford backbone ten times every second, by sending test packets that consume less than 1% of network bandwidth. The link cover for Stanford is even smaller, around 50 packets which allows proactive liveness testing every millisecond using 1% of network bandwidth.

2.1 Network Model

ATPG uses the *header space* framework — a geometric model of how packets are processed we described in [38] (and used in [64]). In header space, protocol-specific meanings associated with headers are ignored: a header is viewed as a flat sequence of ones and zeros. A header is a point (and a flow is a region) in the $\{0,1\}^L$ space, where L is an upper bound on header length. By using the header space framework, we obtain a unified, vendor-independent and protocol-agnostic model of the network[1] that simplifies the packet generation process significantly.

2.1.1 Definitions

Figure 2.1 and Figure 2.2 summarize the definitions in our model.

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

Switches: A *switch transfer function*, T, models a network device, such as a switch or router. Each network device contains a set of forwarding rules (e.g., the forwarding table) that determine how packets are processed. An arriving packet is associated with

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

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

Figure 2.1: The network model: basic types.

```
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)]
```

Figure 2.2: The network model: the switch transfer function.

exactly one rule by matching it against each rule in descending order of priority, and is dropped if no rule matches.

Rules: A *rule* generates a list of one or more output packets, corresponding to the output port(s) the packet is sent to; and defines how packet fields are modified. The rule abstraction models all real-world rules we know including IP forwarding (modifies port, checksum and TTL, but not IP address); VLAN tagging (adds VLAN IDs to the header); and ACLs (block a header, or map to a queue). Essentially, a rule defines how a region of header space at the ingress (the set of packets matching the rule) is transformed into regions of header space at the egress [38].

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

Topology: The *topology transfer function*, Γ, models the network topology by specifying which pairs of ports (p_{src}, p_{dst}) are connected by links. Links are rules that forward

```
function NETWORK(packets, switches, Γ)
    for pk_0 ∈ packets do
        T ←FIND_SWITCH(pk_0.p, switches)
        for pk_1 ∈ T(pk_0) do
            if pk_1.p ∈ EdgePorts then
                #Reached edge
                RECORD(pk_1)
            else
                #Find next hop
                NETWORK(Γ(pk_1), switches, Γ)
```

Figure 2.3: Life of a packet: repeating T and Γ until the packet reaches its destination or is dropped.

packets from p_{src} to p_{dst} without modification. If no topology rules matches an input port, the port is an edge port, and the packet has reached its destination.

2.1.2 Life of a packet

The life of a packet can be viewed as applying the switch and topology *transfer functions* repeatedly (Figure 2.3). When a packet pk arrives at a network port p, the switch function T that contains the input port $pk.p$ is applied to pk, producing a list of new packets $[pk_1, pk_2, \ldots]$. If the packet reaches its destination, it is recorded. Otherwise, the topology function Γ is used to invoke the switch function containing the new port. The process repeats until packets reach their destinations (or are dropped).

2.2 ATPG System

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

Figure 2.4 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 [38]

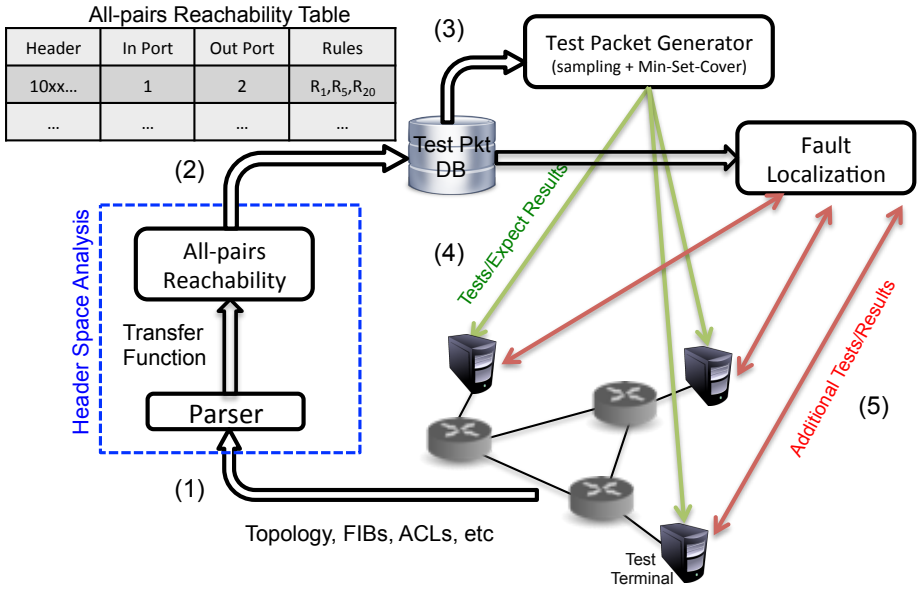


Figure 2.4: ATPG system block diagram.

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 [38], steps 3 through 5 are new.

2.2.1 Test Packet Generation

Algorithm

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

When generating test packets, ATPG must respect two key constraints: (1) *Port:* ATPG

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

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:

Problem 1 (Test Packet Selection) *For a network with the switch functions, $\{T_1, ..., T_n\}$, and topology function, Γ, determine the minimum set of test packets to exercise all reachable rules, subject to the port and header constraints.*

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

Step 1: Generate all-pairs reachability table. ATPG starts by computing the complete set of packet headers that can be sent from each test terminal to every other test terminal. For each such header, ATPG finds the complete set of rules it exercises along the path. To do so, ATPG applies the all-pairs reachability algorithm described in [38]: on every terminal port, an all-`x` header (a header which has all wildcarded bits) is applied to the transfer function of the first switch connected to each test terminal. Header constraints are applied here. For example, if traffic can only be sent on VLAN A, then instead of starting with an all-`x` header, the VLAN tag bits are set to A. As each packet pk traverses the network using the network function, the set of rules that match pk are recorded in $pk.history$. Doing this for all pairs of terminal ports generates an *all-pairs reachability table* as shown in Table 2.1. For each row, the header column is a wildcard

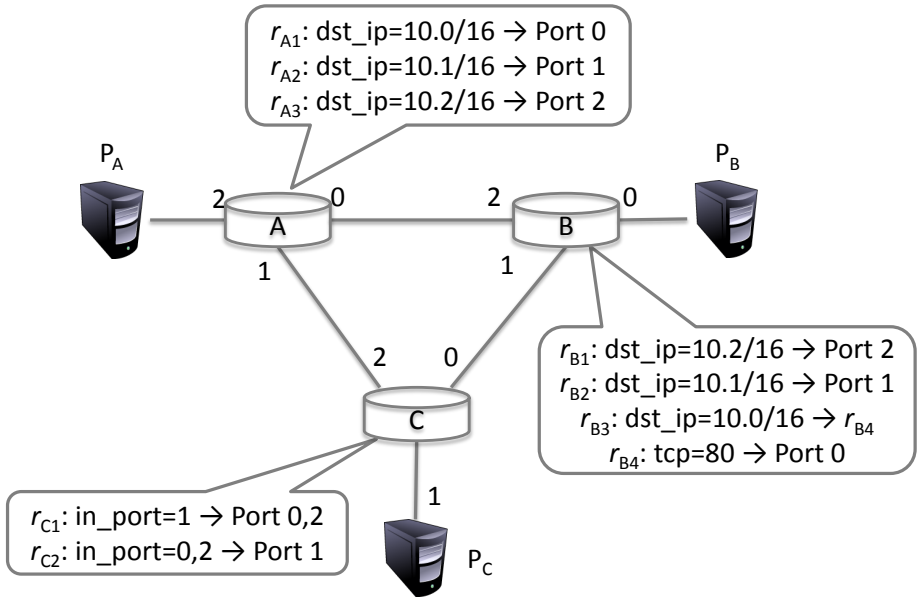


Figure 2.5: Example topology with three switches.

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.

Figure 2.5 shows a simple example network and Table 2.2 is the corresponding all-pairs reachability table. For example, an all-x test packet injected at P_A will pass through switch A. A forwards packets with $dst\_ip = 10.0/16$ to B and those with $dst\_ip = 10.1/16$ to C. B then forwards $dst\_ip = 10.0/16, tcp = 80$ to P_B, and switch C forwards $dst\_ip = 10.1/16$ to P_C. These are reflected in the first two rows of Table 2.2.

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, which 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, which may not be feasible due to the possible exponential state explosion. The sampling scheme chosen is closely related to the fault model used later in the fault localization. We will discuss the fault model in detail in Section 2.2.2.

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

Table 2.2: Test packets for the example network depicted in Figure 2.5. p_6 is stored as a reserved packet.

Step 3: Compression. Several of the test packets picked in Step 2 exercise the same rule. ATPG therefore selects a minimum subset of the packets chosen in Step 2 such that the union of their rule histories cover all rules. The cover can be chosen to cover all links (for liveness only) or all router queues (for performance only). This is the classical Min-Set-Cover problem. While NP-Hard, a greedy $O(N^2)$ algorithm provides a good approximation, where N is the number of test packets. We call the resulting (approximately) minimum set of packets, the *regular test packets*. The remaining test packets not picked for the minimum set are called the *reserved test packets*. In Table 2.2, $\{p_1, p_2, p_3, p_4, p_5\}$ are regular test packets and $\{p_6\}$ is a reserved test packet. Reserved test packets are useful for fault localization (Section 2.2.2).

Properties

The TPS algorithm has the following useful properties:

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

Proof Sketch: Define a rule to be *reachable* if it can be exercised by at least one packet satisfying the header constraint, and can be received by at least one test terminal. A reachable rule must be in the all-pairs reachability table; thus set cover will pick at least one packet that exercises this rule. Some rules are not reachable: for example, an IP prefix may be made unreachable by a set of more specific prefixes either deliberately (to provide backup) or accidentally (due to misconfiguration).

Property 2 (Near-Optimality) *The set of test packets selected by TPS is optimal within logarithmic factors among all tests giving complete coverage.*

Proof Sketch: This follows from the logarithmic (in the size of the set) approximation factor inherent in Greedy Set Cover.

Property 3 (Polynomial Runtime) *The complexity of finding test packets is $O(TDR^2)$ where T is the number of test terminals, D is the network diameter, and R is the average number of rules in each switch.*

Proof Sketch: The complexity of computing reachability from one input port is $O(DR^2)$ [38], and this computation is repeated for each test terminal.

2.2.2 Fault Localization

ATPG periodically sends a set of test packets. If test packets fail, ATPG pinpoints the fault(s) that caused the problem.

Fault model

A rule fails if its observed behavior differs from its expected behavior. ATPG keeps track of where rules fail using a *result function R*. For a rule r, the result function is defined as

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

"Success" and "failure" depend on the nature of the rule: a forwarding rule fails if a test packet is not delivered to the intended output port, whereas a drop rule behaves correctly when packets are dropped. Similarly, a link failure is a failure of a forwarding rule in the topology function. On the other hand, if an output link is congested, failure is captured by the latency of a test packet going above a threshold.

We can divide faults into two categories: *action faults* and *match faults*. An action fault occurs when *every* packet matching the rule is processed incorrectly. Examples of action faults include unexpected packet loss, a missing rule, congestion, and mis-wiring.

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

We will only consider action faults, because they cover most likely failure conditions, and can be detected using only one test packet per rule. In other words, we assume a rule will treat *all* traversing packets equally. We leave match faults for future work.

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

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

Algorithm

Our algorithm for pinpointing faulty rules assumes that a test packet will succeed only if it succeeds at every hop. For intuition, a `ping` succeeds only when all the forwarding rules along the path behave correctly. Similarly, if a queue is congested, any packets that travel through it will incur higher latency and may fail an end-to-end test. Formally:

Assumption 1 (Fault propagation) $R(pk) = 1$ *if and only if* $\forall r \in pk.history, R(r, pk) = 1$

ATPG pinpoints a faulty rule, by first computing the minimal set of potentially faulty rules. Formally:

Problem 2 (Fault Localization) *Given a list of* $(pk_0, R(pk_0), (pk_1, R(pk_1), \ldots$ *tuples, find all* r *that satisfies* $\exists pk_i, R(pk_i, r) = 0$.

We solve this problem opportunistically and in steps.

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

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

Step 3: In most cases, the suspect set is small enough after Step 2, that ATPG can terminate and report the suspect set. If needed, ATPG can narrow down the suspect set further by sending test packets that exercise two or more of the rules in the suspect set using the same technique underlying Step 2. If these test packets pass, ATPG infers that none of the exercised rules are in error and removes these rules from the suspect set. If our Fault Propagation assumption holds, the method will not miss any faults, and therefore will have no *false negatives*.

False positives: Note that the localization method may introduce *false positives*, rules left in the suspect set at the end of Step 3. Specifically, one or more rules in the suspect set may in fact behave correctly.

False positives are unavoidable in some cases. When two rules are in series and there is no path to exercise only one of them, we call 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.

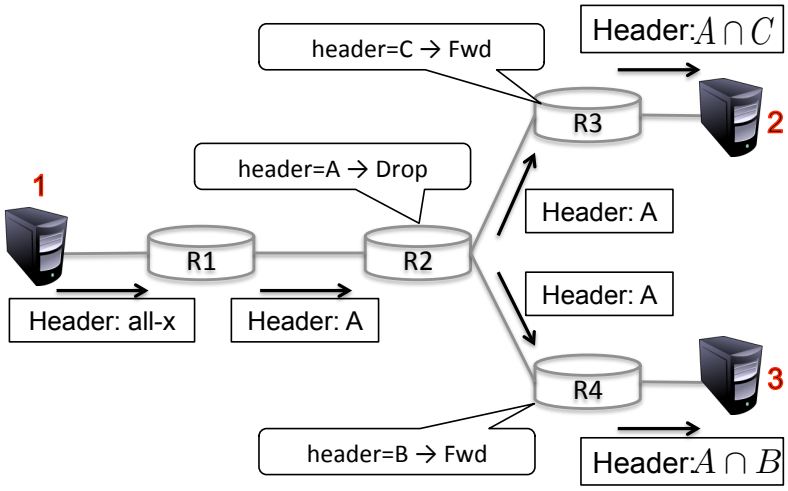


Figure 2.6: Generate packets to test drop rules: "flip" the rule to a broadcast rule in the analysis.

2.3 Use cases

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

2.3.1 Functional testing

We can test the functional correctness of a network by testing that every reachable forwarding and drop rule in the network is behaving correctly:

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

As an example, consider Figure 2.6. 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.

2.3.2 Performance testing

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

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

Available bandwidth: Similarly, we can measure the available bandwidth of a link, or for a particular service class. ATPG will generate the test packet headers needed to test every link, or every queue, or every service class; a stream of packets with these headers can then be used to measure bandwidth. One can use destructive tests, like `iperf/netperf`, or more gentle approaches like packet pairs and packet trains [43]. Suppose a manager specifies that the available bandwidth of a particular service class should not fall below a certain threshold; if it does happen, ATPG's fault localization algorithm can be used to triangulate and pinpoint the problematic switch/queue.

Strict priorities: Likewise, ATPG can be used to determine if two queues, or service classes, are in different strict priority classes. If they are, then packets sent using the lower priority class should never affect the available bandwidth or latency of packets in

the higher priority class. We can verify the relative priority by generating packet headers to congest the lower class, and verifying that the latency and available bandwidth of the higher class is unaffected. If it is, fault localization can be used to pinpoint the problem.

2.4 Implementation

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

2.4.1 Test packet generator

The test packet generator, written in Python, contains a Cisco IOS configuration parser and a Juniper Junos parser. The data plane information, including router configurations, FIBs, MAC learning tables, and network topologies, is collected and parsed through the command line interface (Cisco IOS) or XML files (Junos). The generator then uses the Hassel [30] 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.

2.4.2 Network monitor

The network monitor assumes there are special test agents in the network that are able to send/receive test packets. The network monitor reads the database and constructs test packets, and instructs each agent to send the appropriate packets. Currently, test agents separate test packets by IP Proto field and TCP/UDP port number, but other fields, such as IP option, can also be used. If some of the tests fail, the monitor selects additional test packets from reserved packets to pinpoint the problem. The process repeats until the fault has been identified. The monitor uses JSON to communicate with the test agents, and uses SQLite's string matching to lookup test packets efficiently.

2.4.3 Alternate implementations

Our prototype was designed to be minimally invasive, requiring no changes to the network except to add terminals at the edge. In networks requiring faster diagnosis, the following extensions are possible:

Cooperative routers: A new feature could be added to switches/routers, so that a central ATPG system can 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 interface to control them.

SDN-based testing: In a software defined network (SDN) such as OpenFlow [52], 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.

2.5 Evaluation

2.5.1 Data Sets: Stanford and Internet2

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

Stanford Backbone: With a population of over 15,000 students, 2,000 faculty, and five /16 IPv4 subnets, Stanford represents a large enterprise network. There are 14 operational zone (OZ) Cisco routers connected via 10 Ethernet switches to 2 backbone Cisco routers that in turn connect Stanford to the outside world. Overall, the network has more than 757,000 forwarding entries and 1,500 ACL rules. Data plane configurations are collected through command line interfaces. Stanford has made the entire configuration rule set public [1].

Internet2: Internet2 is a nationwide backbone network with 9 Juniper T1600 routers and 100 Gb/s interfaces, supporting over 66,000 institutions in United States. There are about 100,000 IPv4 forwarding rules. All Internet2 configurations and FIBs of the core routers are publicly available [33], with the exception of ACL rules, which are removed

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

Table 2.3: Test packet generation results for Stanford backbone (top) and Internet2 (bottom), against the number of ports selected for deploying test terminals. “Time to send” packets is calculated on a per port basis, assuming 100B per test packet, 1Gbps link for Stanford and 10Gbps for Internet2.

for security concerns. Although IPv6 and MPLS entries are also available, we only use IPv4 rules in this paper.

2.5.2 Test Packet Generation

We ran ATPG on a quad core Intel Core i7 CPU 3.2GHz and 6GB memory using 8 threads. For a given number of test terminals, we generated the minimum set of test packets needed to test all the reachable rules in the Stanford and Internet2 backbones. Table 2.3 shows the number of test packets needed. For example, the first column tells us that if we attach test terminals to 10% of the ports, then all of the reachable Stanford rules (22.2% of the total) can be tested by sending 725 test packets. If every edge port can act as a test terminal, 100% of the Stanford rules can be tested by sending just 3,871 test

packets. The "Time" row indicates how long it took ATPG to run; the worst case took about an hour, the bulk of which was devoted to calculating all-pairs reachability.

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

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

We also found that 100% *link* coverage (instead of *rule* coverage) only needed 54 packets for Stanford and 20 for Internet2. The table also shows the large benefit gained by compressing the number of test packets — in most cases, the total number of test packets is reduced by a factor of 20-100 using the minimum set cover algorithm. This compression may make proactive link testing (that was considered infeasible earlier [62]) feasible for large networks.

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

Rule structure: The reason we need so few test packets is because of the structure of the rules and the routing policy. Most rules are part of an end-to-end route, and so multiple routers contain the same rule. Similarly, multiple devices contain the same ACL or QoS configuration because they are part of a network wide policy. Therefore, the number of distinct regions of header space grow linearly, not exponentially, with the diameter of the network.

We can verify this structure by clustering rules in Stanford and Internet2 that match the same header patterns. Figure 2.7 shows the distribution of rule repetition in Stanford

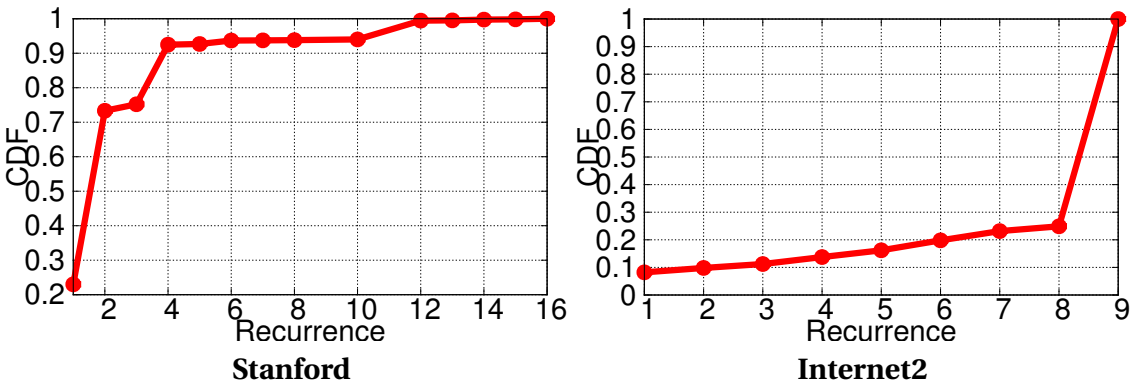


Figure 2.7: The cumulative distribution function of rule repetition, ignoring different action fields.

and Internet2. In both networks, 60%-70% of matching patterns appear in more than one router. We also find that this repetition is correlated to the network topology. In the Stanford backbone, which has a two-level hierarchy, matching patterns commonly appear in 2 (50.3%) or 4 (17.3%) routers, which represents the length of edge-to-Internet and edge-to-edge routes. In Internet2, 75.1% of all distinct rules are replicated 9 times, which is the number of routers in the topology.

2.5.3 Testing in an Emulated Network

To evaluate the network monitor and test agents, we replicated the Stanford backbone network in Mininet [44], a container-based network emulator. We used Open vSwitch (OVS) [59] to emulate the routers, using the real port configuration information, and connected them according to the real topology. We then translated the forwarding entries in the Stanford backbone network into equivalent OpenFlow [52] rules and installed them in the OVS switches with Beacon [7]. We used emulated hosts to send and receive test packets generated by ATPG. Figure 2.8 shows the part of network that is used for experiments in this section. We now present different test scenarios and the corresponding results:

Forwarding Error: To emulate a functional error, we deliberately created a fault by replacing the action of an IP *forwarding* rule in $boza$ that matched $dst\_ip = 172.20.10.32/27$ with a *drop* action (we called this rule R_1^{boza}). As a result of this fault, test packets from

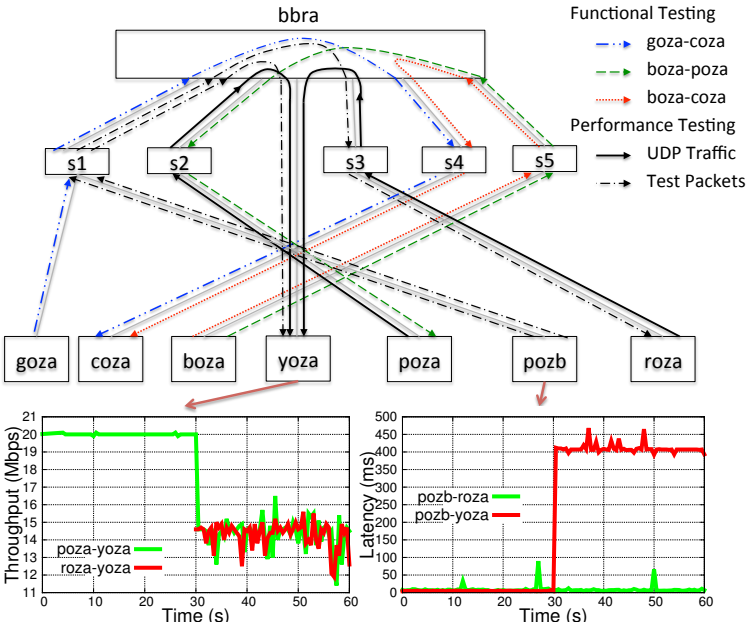


Figure 2.8: A portion of the Stanford backbone network showing the test packets used for functional and performance testing examples in Section 2.5.3.

boza to *coza* with $dst\_ip = 172.20.10.33$ failed and were not received at *coza*. Table 2.4 shows two other test packets we used to localize and pinpoint the fault. These test packets shown in Figure 2.8 in $goza-coza$ and $boza-poza$ are received correctly at the end terminals. From the *rule history* of the passing and failing packets in Table 2.1, we deduce that only rule R_1^{boza} could possibly have caused the problem, as all the other rules appear in the rule history of a received test packet.

Congestion: We detect congestion by measuring the one-way latency of test packets. In our emulation environment, all terminals are synchronized to the host's clock so the latency can be calculated with a single time-stamp and one-way communication[2].

To create congestion, we rate-limited all the links in the emulated Stanford network to 30 Mb/s, and created two 20 Mb/s UDP flows: *poza* to *yoza* at $t = 0$ and *roza* to *yoza* at $t = 30s$, which will congest the link $bbra - yoza$ starting at $t = 30s$. The bottom left graph next to *yoza* in Figure 2.8 shows the two UDP flows. The queue inside the routers

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

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

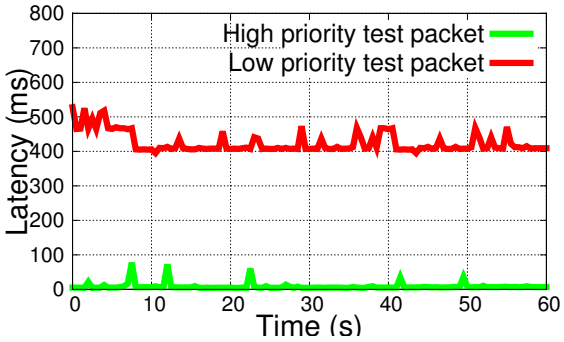
Table 2.4: Test packets used in the functional testing example. In the rule history column, R is the IP forwarding rule, L is a link rule and S is the broadcast rule of switches. R_1 is the IP forwarding rule matching on 172.20.10.32/27 and R_2 matches on 171.67.222.64/27. L_b^e in the link rule from node b to node e. The table highlights the common rules between the passed test packets and the failed one. It is obvious from the results that rule R_1^{boza} is in error.

will build up and test packets will experience longer queuing delay. The bottom right graph next to $pozb$ shows the latency experienced by two test packets, one from $pozb$ to $roza$ and the other one from $pozb$ to $yoza$. At $t = 30s$, the $bozb - yoza$ test packet experiences much higher latency, correctly signaling congestion. Since these two test packets share the $bozb - s_1$ and $s_1 - bbra$ links, ATPG concludes that the congestion is not happening in these two links; hence ATPG correctly infers that $bbra - yoza$ is the congested link.

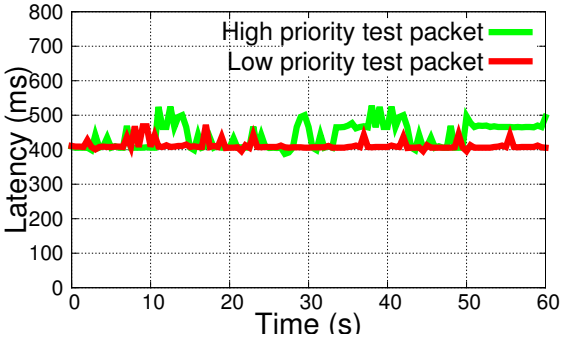
Available Bandwidth: ATPG can also be used to monitor available bandwidth. For this experiment, we used Pathload [34], a bandwidth probing tool based on packet pairs/packet trains. We repeated the previous experiment, but decreased the two UDP flows to 10 Mb/s, so that the bottleneck available bandwidth was 10 Mb/s. Pathload reports that $bozb - yoza$ has an available bandwidth[3] of 11.715 Mb/s, $bozb - roza$ has an available bandwidth of 19.935 Mb/s, while the other (idle) terminals report 30.60 Mb/s. Using the same argument as before, ATPG can conclude that $bbra - yoza$ link is the bottleneck link with around 10 Mb/s of available bandwidth.

Priority: We created priority queues in OVS using Linux's `htb` scheduler and `tc` utilities. We replicated the previously "failed" test case $pozb - yoza$ for high and low priority queues respectively.[4] Figure 2.9 shows the result.

[3]All numbers are the average of 10 repeated measurements.
[4]The Stanford data set does not include the priority settings.



(a) Low



(b) High

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

(c) Available bandwidth

Figure 2.9: Priority testing: Latency measured by test agents when low (a) or high (b) priority slice is congested; available bandwidth measurements when the bottleneck is in low/high priority slices (c).

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

2.5.4 Testing in a Production Network

We deployed an experimental ATPG system in 3 buildings in Stanford University that host the Computer Science and Electrical Engineering departments. The production network consists of over 30 Ethernet switches and a Cisco router connecting to the campus backbone. For test terminals, we utilized the 53 WiFi access points (running Linux) that were already installed throughout the buildings. This allowed us to achieve high

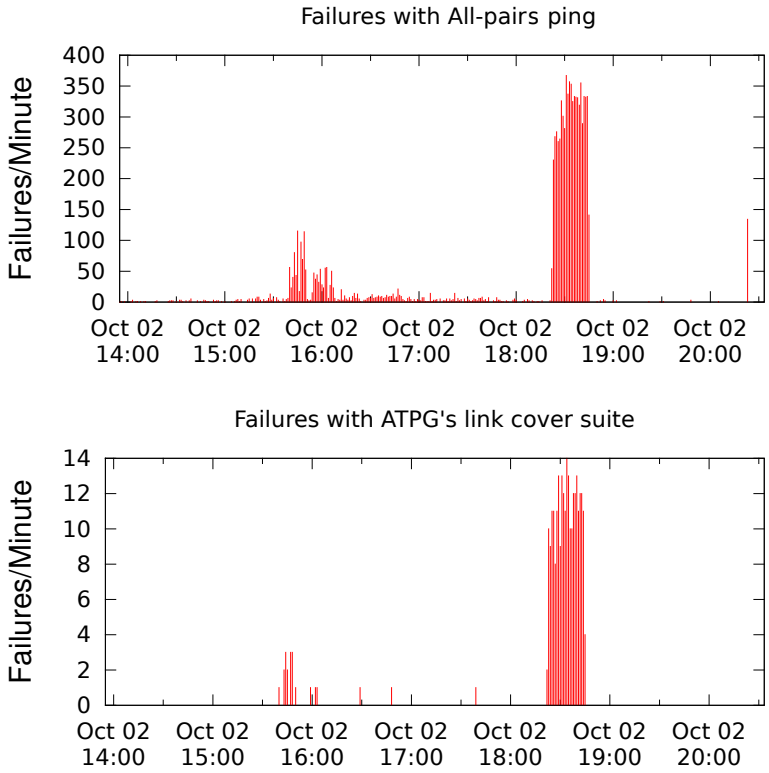


Figure 2.10: The Oct 2, 2012 production network outages captured by the ATPG system as seen from the lens of an inefficient cover (all-pairs, top picture) and an efficient minimum cover (bottom picture). Two outages occured at 4PM and 6:30PM respectively.

coverage on switches and links. However, we could only run ATPG on essentially a Layer 2 (bridged) Network.

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

During the experiment, a major network outage occurred on October 2. Figure 2.10 shows the number of failed test cases during that period. While both all-pairs `ping` and ATPG's selected test suite correctly captured the outage, ATPG uses significantly less test packets. In fact, ATPG uses only 28 test packets per round compared with 2756 packets in all-pairs `ping`, a 100x reduction. It is easy to see that the reduction is from quadratic overhead (for all-pairs testing between 53 terminals) to linear overhead (for a set cover of the 30 links between switches). We note that while the set cover in this experiment is so simple that it could be computed by hand, other networks will have Layer 3 rules and more complex topologies requiring the ATPG minimum set cover algorithm.

The network managers confirmed that the later outage was caused by a loop that was accidentally created during switch testing. This caused several links to fail and hence more than 300 pings failed per minute. The managers were unable to determine why the first failure occured. Despite this lack of understanding of the root cause, we emphasize that the ATPG system correctly detected the outage in both cases and pinpointed the affected links and switches.

2.6 Discussion

2.6.1 Overhead and Performance

The principal sources of overhead for ATPG are polling the network periodically for forwarding state and performing all-pairs reachability. While one can reduce overhead by 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/multi-machine header space library. Second, instead of extracting the complete network state every time ATPG is triggered, an *incremental* state updater can significantly reduce both the retrieval time and the time to calculate reachability. We are working on a real-time version of ATPG 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 (> 1Gbps) most modern CPUs are capable of receiving.

2.6.2 Limitations

As with all testing methodologies, ATPG has limitations: **1) Dynamic boxes**: ATPG cannot model boxes whose internal state can be changed by test packets. For example, a NAT that dynamically assigns TCP ports to outgoing packets can confuse the online monitor as the same test packet can give different results. **2) Non-deterministic boxes**: Boxes can load-balance packets based on a hash function of packet fields, usually combined with a random seed; this is common in multipath routing such as ECMP. When the hash algorithm and parameters are unknown, ATPG cannot properly model such rules. However, if there are known packet patterns that can iterate through all possible outputs, ATPG can generate packets to traverse every output. **3) Invisible rules**: A failed rule can make a backup rule active, and as a result no changes may be observed by the test packets. This can happen when, despite a failure, a test packet is routed to the expected destination by other rules. In addition, an error in a backup rule cannot be detected in normal operation. Another example is when two drop rules appear in a row:

the failure of one rule is undetectable since the effect will be masked by the other rule. **4) Transient network states**: ATPG cannot uncover errors whose lifetime is shorter than the time between each round of tests. For example, congestion may disappear before an available bandwidth probing test concludes. Finer-grained test agents are needed to capture abnormalities of short duration. **5) Sampling**: ATPG uses sampling when generating test packets, with the assumption that a rule will treat all traversing packets equally. As a result, if an error is not necessarily uniform across all matching headers, ATPG can miss it. In the worst case (when only one header is in error), to capture such error, exhaustive testing is needed.

2.7 Related Work

We are unaware of earlier techniques that automatically generate test packets from configurations. The closest related work we know of are offline tools that check invariants in networks. In the control plane, NICE [13] attempts to exhaustively cover the code paths symbolically in controller applications with the help of simplified switch/host models. In the data plane, Anteater [49] models invariants as boolean satisfiability problems and checks them against configurations with a SAT solver. Header Space Analysis [38] uses a geometric model to check reachability, detect loops, and verify slicing. Recently, SOFT [42] was proposed to verify consistency between different OpenFlow agent implementations that are responsible for bridging control and data planes in the SDN context. ATPG complements these checkers by directly *testing* the data plane and covering a significant set of dynamic or performance errors that cannot otherwise be captured.

End-to-end probes have long been used in network fault diagnosis in work such as [22, 23, 25, 40, 47, 48, 50]. Recently, mining low-quality, unstructured data, such as router configurations and network tickets has attracted interest [28, 45, 74]. 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 [6, 9]. ATPG improves the detection granuality to the rule level by employing router configuration and data plane information. Further, 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 [24, 46, 58, 77]. 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 [12] 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.

2.8 Summary

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 [62]. It suffices to find a minimal set of end-to-end packets that traverse each link. But doing this requires a way of abstracting across device specific configuration files (e.g., header space), generating headers and the links they reach (e.g., all-pairs reachability), and finally determining a minimum set of test packets (Min-Set-Cover). Even the fundamental problem of automatically generating test packets for efficient liveness testing requires techniques akin to ATPG.

ATPG, however, goes much further than liveness testing with the same framework. ATPG can test for reachability policy (by testing all rules including drop rules) and performance health (by associating performance measures such as latency and loss with test packets). Our implementation also augments testing with a simple fault localization scheme also constructed using the header space framework. As in software testing, the formal model helps maximize test coverage while minimizing test packets. Our results show that all forwarding rules in Stanford backbone or Internet2 can be exercised by a surprisingly small number of test packets ($< 4,000$ for Stanford, and $< 40,000$ for Internet2).

Network managers today use primitive tools such as `ping` and `traceroute`. Our survey results indicate that they are eager for more sophisticated tools. Other fields of engineering indicate that these desires are not unreasonable: for example, both the ASIC and software design industries are buttressed by billion dollar tool businesses that supply techniques for both static (e.g., design rule) and dynamic (e.g., timing) verification. In fact, many months after we built and named our system, we discovered to our surprise that ATPG was a well-known acronym in hardware chip testing, where it stands for Automatic Test *Pattern* Generation [3]. We hope network ATPG will be equally useful for automated dynamic testing of production networks.

Chapter 3

Gray-box Testing in Large Networks

There are things known and there are things unknown, and in between are the doors of perception.

Aldous Huxley (1894-1963)

THE previous chapter demonstrates that ATPG works well in networks like Stanford's backbone network and the Internet2. In this chapter, we aim to solve a *bigger* problem – large networks with hundreds of routers and links. In these networks, faults that lead to packet loss or high latency are not only commonplace but also hard to localize. Nowadays, operators often rely on low-level SNMP counters (e.g., link load and drops) from routers to pinpoint such faults, but these counters are very noisy and lead to many false alerts (Section 3.5.3). Consequently, operators frequently ignore them, which leads to high delay in fault detection and localization. In the operational network that we study, we have seen many faults that took several hours to detect and localize.

Given the shortcomings of SNMP counters, operators have used active testing to detect and localize faults [62]. In such testing, test agents (at the network boundary) send end-to-end probes (e.g., pings) along network paths to detect faults. A simple active

testing technique is *all-pairs*, which sends probes between each pair of test agents. Borrowing software testing terminology, we call this technique *black box* because it requires no knowledge of the network. It shares the shortcomings of any black box tester—lack of scalability and inability to reason about coverage. To effectively test a large network, many test agents are needed, which leads to an intractably large number of probes per test interval. Even then, it is difficult to determine what fraction of links and routers were covered by the probes.

To overcome these shortcomings, researchers have proposed the equivalent of *white box* testing [8, 23, 70]. These techniques need detailed information about the network such as the entire FIB (forwarding information base) from each router, which describes how a router processes a given packet. Using this information, they compute a subset of all possible probes that can cover all the routers and links in the network.

Based on what is demonstrated in Chapter 2, we may naturally raise this question:

> Can we use white box testers, such as ATPG, to test large networks?

Unfortunately, for most of modern large networks, the answer is "no". This is because of the amount of information a white box tester needs to gather. Fundamentally, the techniques assume that FIB state is a consistent snapshot across all routers, which is hard to obtain in a large network with high churn. Furthermore, today's router CPUs tend to be underpowered, and it is generally not possible to frequently read the entire FIB state. This is further complicated by multipath routing (e.g., ECMP), which none of the white box techniques handle. With ECMP, the FIB does not fully describe how a given packet will be forwarded; the outgoing link of a packet is decided by hashing packet headers (usually the 5-tuple). Today, the hash function is often not known to the network operator.

We thus argue that testing in large networks must be *gray box*, based on easily obtainable information about the network. It should then use this information to efficiently cover all links and routers. Based on the information available for a given network, different gray box testing techniques may be developed.

We develop NetSonar, one such gray box network tester. NetSonar operates in three phases. In the first phase, it computes a comprehensive test plan, the set of probes

needed to cover all links/routers, assuming that the set of paths between a source and destination is known (but not which exact path will be taken by a given packet). To handle uncertainty due to multipath routing, the plan is computed using probabilistic path covers – it includes probes with different packet headers (5-tuple) such that a sufficient number of paths between a source and destination are covered with high probability. To aid fault localization, plan computation also uses diagnosable link covers – each link is covered by a configurable number of probes.

In the second phase, NetSonar executes the plan. Each probe involves a series of pings with the same 5-tuple; multiple pings are needed to reliably detect latency spike. In addition, we also conduct traceroutes with the same 5-tuples as the pings so that we can map a ping to the actual path taken. Finally, in the third phase, we process the probing results to detect and localize faults, after filtering out noise due to factors such as overloaded test agents.

We deploy NetSonar to troubleshoot faults in a global inter-datacenter (DC) network of a large online service provider, with test agents located in the DCs. Our results are encouraging. In Feb-March 2013, NetSonar detected 66 faults involving high latency and localized them to a router or link in the core of the network. Of these, 56 (84.8%) faults can be cross-verified with other monitoring data sources. Several of the faults that we detected were deemed high priority and, after investigation by operators, it was confirmed that NetSonar's localization was correct. We also find that, compared to NetSonar, fault alerting based on SNMP counters would have led to 87 times the number of alerts, the vast majority of which would have been false positives.

3.1 Motivation

Consider the simple topology in Figure 3.1 with four routers and four hosts that can act as test agents. Assume that our goal is to test all 3 inter-router links for gray faults (i.e., those that go undetected by low-level monitoring such as keep alive messages) that cause packet loss or delay. A black box approach such as all-pairs can achieve this goal, but with 12 (4×3) probes. Such quadratic behavior cannot scale to large networks.

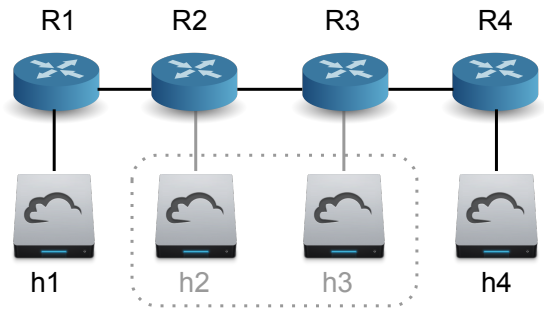


Figure 3.1: Probing based on knowledge of the network can reduce the number probes by strategically selecting probes. All-pairs testing will generate 12 probes, while only 2 probes ($h1 \rightarrow h4$, $h4 \rightarrow h1$) can cover all links.

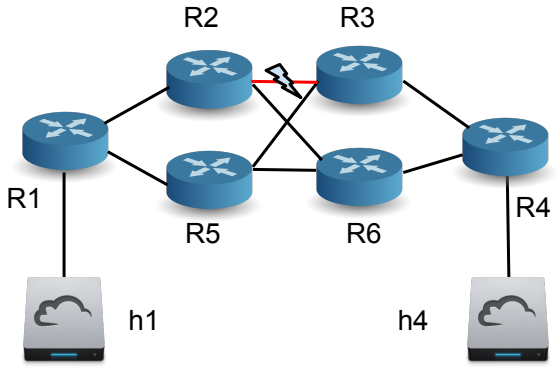


Figure 3.2: A link with performance problems is difficult to detect in the presence of multipath routing.

A white box approach can compute that the same test coverage can be achieved using only 2 probes: $h1 \rightarrow h4$ and $h4 \rightarrow h1$. Both are needed because we must test the links in both directions. Such minimal probe sets can be computed by the Min-Set-Cover algorithm, as in [5, 8, 78]. While a white box technique can compute a small set of probes, as mentioned before, it needs highly detailed information (e.g., the entire FIB state) that is difficult to acquire in large networks.

Further, current white box approaches do not consider multipath routing, which is prevalent in large networks. Figure 3.2 illustrates the challenge posed by multipath routing. The network employs equal-cost multipath (ECMP): four shortest paths between $R1$

and $R4$ are used in parallel ($R1-R2-R3-R4$, $R1-R2-R6-R4$, etc.). Assume that the link $(R2,R3)$ is faulty. There is no guarantee that a probe sent between $h1$ and $h4$ will traverse the faulty link. Whether the faulty link is covered depends on packet headers in probes and the (unknown) hash function used by $R1$ and $R2$.

We develop NetSonar based on the observations above. We call it gray box because it relies on some knowledge of network routing but not highly precise information on how individual routers forward traffic. In particular, we assume that the *set* of available paths (four paths in Figure 3.2) between each pair of ingress-egress routers is known, without knowing the precise path a given packet header traverses.

3.2 Related Work

NetSonar builds on the long, rich line of work in network fault localization. While it is almost impossible to cover all prior techniques, we classify prior work into five categories and describe how we relate to each.

Unplanned tomography: NetSonar's goal is to identify links with faults such as congestion, which falls squarely under Boolean Network Tomography, pioneered by Duffield et al. [23], followed by many others (e.g., [19, 22, 55, 56, 80, 81]). Besides, more general inference algorithms include SCORE [40] and Sherlock [4]. Unplanned tomography assumes that the set of input measurements is fixed (and uncontrollable). It thus cannot provide guarantees on failure detection and coverage (Section 3.6.2). NetSonar leverages the inference algorithms developed in this line of work, but provides them as input the results of its planned testing.

Planned testing: Many researchers have used optimization techniques to plan tests. Bejerano and Rastogi [8] consider the problem of minimizing test agents and probes to cover every link. Nguyen and Thiran [54] extend these results to diagnose multiple failures. Kumar and Kaur [41] further explore test agent placement in the face of routing dynamics. Barford et al. [5] propose a time-variant weighted based cover algorithm. ATPG [78] extends traditional link covers to more general forwarding and access control rule covers. Huang et al. [32] evaluate the Nguyen and Thiran's approach in a controlled testbed; they point out several practical issues such as scalability and errors caused by a

lack of consistent snapshots.

NetSonar improves existing planned testing methods in two ways. First, all these works cannot handle multipath networks. As we show in Section 3.3.3, handling multipath networks is non-trivial, as it requires techniques such as probabilistic path covers. Second, minimizing the number of test agents or probes may lead to *identifiability* problem, which means bad links are not uniquely identifiable. Existing approaches usually require iterative active probing [5] after problem detection, or direct probing of router interfaces [8, 41]. Our diagnosable link covers provide a simple, practical way to balance probing efficiency and identifiability.

Scalable end-to-end monitoring: Scalable end-to-end monitoring techniques infer end-to-end measures (e.g., latency) between *all* test agents using measurements taken between *some* test agents. Techniques include rank-based [16], SVD [17], and Bayesian experiment design [70]. In principle, the results of such methods could be fed into a network tomography solution to isolate high delay links. However, this is an indirect and unnecessarily complex way to do fault diagnosis. Further, the evaluations of these techniques only demonstrate their effectiveness for inferring end-to-end measures, not for fault localization.

Multipath measurement: Recent studies [2, 20, 61] point out that traditional host-based diagnosis tools such as ping and traceroute are inadequate for multipath networks. While Paris Traceroute [2] also varies packet headers like NetSonar in order to cover paths, it outputs a topology, not the tuples required for coverage. It also uses a *dynamic* scheme based on hypothesis testing to discover a multipath topology. By contrast, NetSonar uses *static* analysis of the topology to compute the number of random tuples needed.

Using other data sources: NetDiagnoser [22] takes control plane messages into account in Binary Network Tomography. Others [35, 36, 47, 48] focus on temporal correlation between different network events using statistical mining. These tools complement NetSonar by exploring several different dimensions.

In summary, the first contribution of NetSonar is test plan computation for large multipath networks; this is different from other work that computes test plans but cannot handle multipathing [8, 41, 56, 78] and require complete network knowledge. The

second contribution of NetSonar is deployment experience in IDN, a global inter-DC network, together with comparison with SNMP counter methods.

3.3 NetSonar

We designed NetSonar to cover all links and routers in a global backbone network, mainly targeting performance problems such as packet loss and latency spikes. Today, global backbone networks typically use MPLS to establish many parallel label switched paths (LSPs) between two sites. NetSonar examines the network topology and LSP configuration to generate test plans with high link coverage. The plans are then executed by corresponding test agents—currently, ping and traceroute agents. Once the test data is collected, post-processing cleans the data, triangulates the problem and provides reports to human operators.

3.3.1 Overview

NetSonar works in three phases: computing covers, executing test plan, and localizing faults.

In the first offline bootstrapping phase, NetSonar computes a cover by reading the set of LSPs in the network.[1] It then uses its knowledge of routes and available test agents to compute a set of test probes that forms two covers: First, for each pair of test agents connected by N LSPs, it generates a *probabilistic path cover* by choosing k random TCP port-pairs to send test packets between them. If we assume that the (unknown) hash function in the edge router maps every 5-tuple (IP source/destination addresses, TCP source/destination ports, IP protocol type) to a route with equal probability, the relationship of N and k follows the analysis of the "coupon collector's problem" (Problem 3). Second, across the entire network, we generate a *diagnosable link cover* to cover all links – in other words, if a single link exhibits a performance fault, there is sufficient information for the tester to localize that link without further probing. This is done by selecting pairs of test agents so that each link is covered multiple times.

[1] In an IP (non-MPLS) network, NetSonar can use shortest paths or link weights to bootstrap.

In the second online testing phase, for all these 5-tuples, NetSonar sends low-frequency traceroute and high-frequency ping *simultaneously*. Why is traceroute needed when we already know the paths through LSPs? First, LSP information may be outdated when the actual probes are sent out. More fundamentally, the LSP information is insufficient. Assume there are two paths $p1$ and $p2$ between S and D. Recall that a probabilistic path cover only guarantees a certain probability of covering both $p1$ and $p2$. However, because of the non-determinism, *for a specific packet*, the test agent does not know whether it is mapped to $p1$ or $p2$. Hence, its ping results cannot be mapped to the correct links. Traceroute maps each chosen 5-tuple (and ultimately associated pings) to a specific path. Note that traceroute triggers ICMP responses from routers; thus it cannot run very often, currently once every five minutes for each 5-tuple. At the same time, NetSonar sends ping probes for each 5-tuple chosen in the diagnosable link cover much more frequently, currently every 3 seconds between any pair of agents. We call this combined ping-traceroute approach *traceable probes*. Thanks to the traceroute done earlier, NetSonar knows the path taken by each ping packet.

Finally, in the offline analysis phase, NetSonar collects ping and traceroute results, and uses a fault localization algorithm to pinpoint the faulty spot.

The entire NetSonar workflow is an *open loop*. This is to ensure that even when forwarding information changes *between* phases, each phase can still capture relatively accurate information. For example, in the offline phase, the cover calculation does not depend on the traceroute results in the online phase. Similarly, in the final analysis phase, NetSonar does not need to issue more probes (back to the online phase) since each link has already been covered multiple times.

3.3.2 Components

NetSonar contains three loosely coupled main components: test agents, the controller, and the data sink, as depicted in Figure 3.3.

Test agents: Test agents running on DC servers execute the test plan. Each test agent contains two test clients: a TCP traceroute client and a TCP ping client. Traceroute client collects only path information; ping client collects latency information.

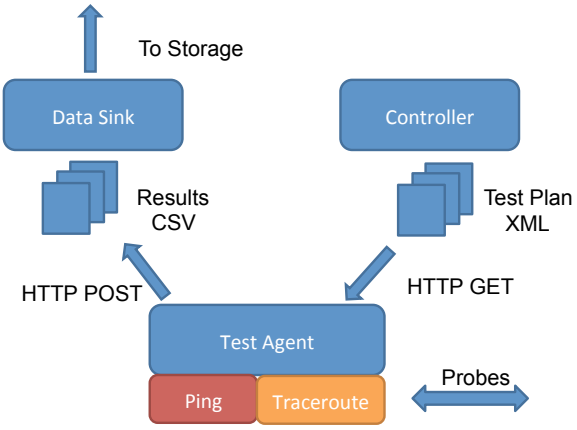


Figure 3.3: NetSonar Components

Conventional ICMP based ping and traceroute are usually processed via a "slow-path" through router CPUs. We choose TCP ping and traceroute because TCP is the dominant traffic type, hence the test results closely reflect application-perceived performance. Each ping probe is a TCP SYN packet sent to an open remote TCP port. We measure the time gap between this SYN packet and the returned ACK packet as the latency. A TCP traceroute probe is a set of SYN packets with small TTLs, so that routers along the path can return ICMP Time-Exceeded messages.

Each test agent downloads its own test plan – a list of commands – periodically. The commands are executed by both traceroute and ping clients. Traceroutes and pings in the a single test use *the same* 5-tuples, so that we know the exact path for each ping result in a multipath network. Notice that these traceable probes also provide certain level of robustness against path changes - even the LSP data is slightly outdated, we are still able to map ping results to correct paths at the time of testing. NetSonar uses raw TCP sockets so that the TCP source ports are configurable. Both ping and traceroute intervals are configurable. The frequency of traceroute is chosen to avoid overwhelming routers.

Controller and data sink: The logically centralized controller periodically reads the network topology, LSP data, as well as the health of test agents, and generates test plans for individual test agents that cover all links and routers. The controller ensures that traceroute-triggered ICMP responses from any router and the number of commands

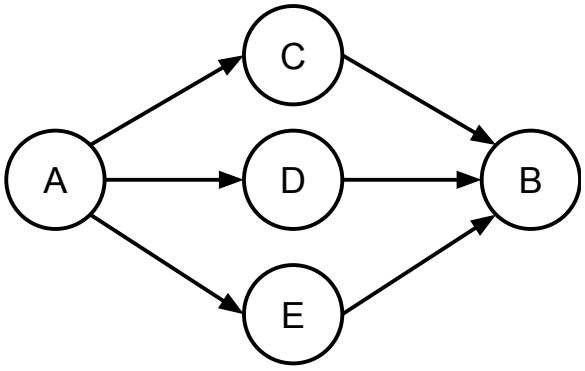


Figure 3.4: Source multipath. Source router A can select one of three paths ($N = 3$).

assigned to any individual test agent are below certain safe thresholds. Moreover, the loss of individual test agents will not result in a large drop in coverage. The data sink collects the results in comma-separated values (CSV) from test agents, and uploads the aggregated data to data storage for analysis.

3.3.3 Probabilistic Path Covers

Following Paris traceroute [2], we disambiguate different load balancing paths by varying the 5-tuple headers of ping and traceroute probes between the same pair of test agents. A fundamental question is: without knowledge of the hash functions used by routers, how many different 5-tuples are needed to cover all links and interfaces between a pair of test agents?

Source Multipath

We start by examining the *source multipath* network, where load balancing only happens at the source router. A example source multipath network is depicted in Figure 3.4: router A is the source router that chooses path among $A-C-B$, $A-D-B$, and $A-E-B$. This load balancing approach is popular in LSP-based wide-area networks such as IDN, where multiple paths are set up between two edge routers, and only the edge router can decide the exact path of a particular class of packets.

Consider a router that has N next-hops for a class of outgoing packets. In the worst

case, we need all possible 5-tuples assuming an unknown, adversarial hash function. Instead, we assume: (1) Each 5-tuple is hashed to a next-hop with a uniform probability of $1/N$; (2) Each tuple is treated by the hash function independently. The first assumption is reasonable because of the prevalent use of ECMP to spread the traffic uniformly; the second is reasonable because most router hash functions treat tuples statelessly. With these assumptions, the number of tuples needed can be derived from the Coupon Collector's problem [26]:

Problem 3 (Coupon Collector's Problem) *Suppose that there are N different coupons, equally likely, from which coupons are being collected with replacement. What is the probability of collecting all N coupons in less than k trials?*

The analogy is easily seen: each next-hop is a different coupon; each sending of a probe with a specific 5-tuple is equivalent to drawing a coupon from the pool. Let T be the number of trials needed, we can calculate the expectation as follows:

$$\mathbb{E}(T) = N\sum_{i=1}^{N}\frac{1}{i} \sim O(N\log(N))$$

The expected value of T does not guarantee 100% coverage. One can calculate the exact probability:

$$\Pr(T \leq k) = \sum_{i=1}^{N}(-1)^{(i+1)}\binom{N}{i}\left(1-\frac{i}{N}\right)^{k}, \text{ where } k \geq N$$

Figure 3.5 shows the number of 5-tuples needed to exercise different next-hops. We define the "inflation factor" as T divided by N. For instance, 75 randomly selected 5-tuples with inflation factor 4.7 can exercise 16 next-hops with 90% confidence. To achieve 99% confidence, we need approximately 110 5-tuples with inflation factor 6.9. Note that the percentage here represents the confidence to cover *all* next-hops, not the percentage of next-hops covered. The analysis above guides NetSonar in creating a test plan to cover all links and routers in a multipath network at a given confidence level. We call each set of 5-tuples between a pair of test agents a *probabilistic cover*.

We will show in Section 3.6.1 that when test agents are deployed at all sites, we can cover all links and routers with a smaller inflation factor (e.g., 2 in IDN). This is because

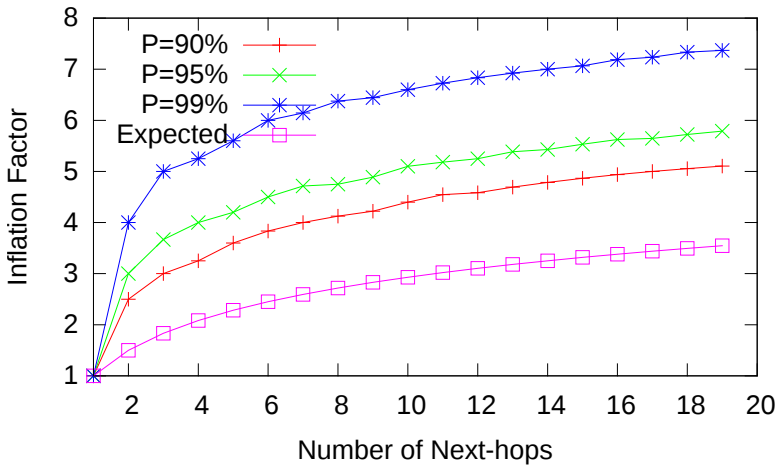


Figure 3.5: The number of 5-tuples needed to exercise N next-hops, with different confidence.

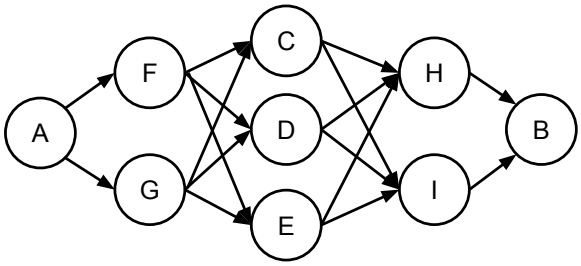


Figure 3.6: Multilevel multipath with 3 levels of edge, aggregation and core. N is determined by the maximum number of links between any two levels ($N = 6$).

even if some paths are missed between a pair of test agents, the links on these paths can still be covered by other test agent pairs. By contrast, a sparse deployment requires larger inflation factor.

Multilevel Multipath

Today's data center networks (such as a fat-tree network) often employ a more general load balancing scheme than source multipath, which we refer to as *multilevel multipath*. In multilevel multipath, routers are categorized into levels. A set of "symmetric" links connecting two levels share the traffic load *equally*. Figure 3.4 shows a full-mesh, 3-level network: A and B are two edge routers; F, G and H, I are two aggregation levels;

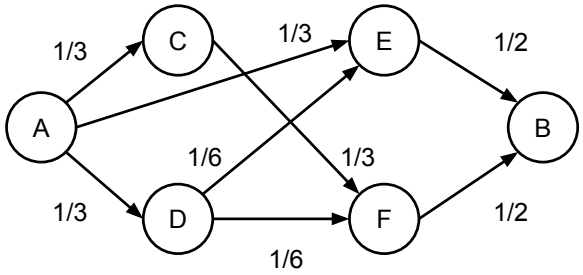


Figure 3.7: General multipath. N can be determined by the minimum traversal probability ($N = 6$).

and C, D, E form the core level. All three levels can independently make load balancing decisions. For example, router F can choose C, D, or E as its next hop.

Observe that the number of unique paths grows exponentially – in this example, there are $2 \times 3 \times 2 = 12$ paths between A and B. However, this does *not* imply that we should plug $N = 12$ into Problem 3. This is because a packet sent from A to B has a probability of $1/2$ to traverse any particular link between edge and aggregation levels such as (A, F), and a probability of $1/6$ to traverse any link between aggregation and core such as (F, C). Hence, we need approximately $2\log(2)$ and $6\log(6)$ packets respectively to cover these links. Since the same packet will traverse all levels, we only need the maximum (not the sum or product) of these two numbers, which is equivalent to $N = 6$.

More generally, in a multilevel multipath network, the results in Problem 3 can apply where N equal to the *maximum* number of links between any two levels.

General Multipath

A natural way to generalize multilevel multipath is to remove the constraints of levels and symmetry. Such general multipath routing is common in large ISP networks. In general, the set of paths between a source and destination forms a single-source, single-sink directed acyclic graph, where each node represents a router, and each edge marks a (possible) forwarding direction. Figure 3.7 shows such an example. For simplicity, assume that each node sends packet to all its next hops with equal probability as in ECMP.

We first calculate the probability of a random packet sent from A to B traversing a

particular link, called *traversal probability*:

1) Sort the nodes topologically so that any prior hop of a node X occurs before X in the sorted order.

2) Initialize the traversal probability of all links and nodes to 0 except for the source node which is assigned probability 1.

3) Start from the source node in topological order: for each node with probability of p and m outgoing links, assign p/k to each outgoing link and add p/m to each of its next-hop node's probability. Note that we can generalize to unequal load balancing (as in weighted cost multipath) by having each node add a weighted portion of its probability to its next-hop. We can add these probabilities because the events are disjoint: a particular packet must arrive from exactly one prior hop.

In Figure 3.7, each link is annotated with its traversal probability. For example, a random packet sent from A to B has probability of $1/3$ traversing link (C,F).

Next, we find the link l with minimum traversal probability p. In our example, l is either (C,E) and (C,F), both with $p = 1/6$. We can now use the coupon collector formula of Problem 3 with $N = \lceil 1/p \rceil$. This is because with $O(N \log N)$ packets we will (almost for sure) traverse l. This also implies that we will traverse *all other links* that have higher traversal probability. Note that this is a coarse upper bound; finding the exact number is known as the non-uniform coupon collector's problem [66].

3.3.4 Diagnosable Link Covers

A probabilistic path cover only covers all paths between *a pair* of test agents. NetSonar's goal is to generate a list of test targets for each test agent so that the overall set of tests forms what we call a *diagnosable link cover*. More formally, assume the network topology is a directed graph $G = (V,E)$, where V are nodes and E are links. Besides the topology, we also know a set of *paths*. Each path is a set of links that connects two nodes. We can treat each path as a subset of E, and choose a *minimum set of paths* to cover all links in E. This is known as a NP-Hard "Min-Set-Cover" problem. A well-known $O(N^2)$ approximation (hereafter referred to as MSC) can solve this problem, where N is the number of paths. MSC first initializes the uncovered edge set U to E. In each iteration,

```
U ← E
# Initialize counters
for link ∈ U do
    visited[link] ← 0
repeat
    path, score ← FIND_MAX(U, paths)
    paths ← paths − path
    if score > 0 then
        for link ∈ path do
            visited[link]++
            if visited[link] == α then
                U ← U − link
until U == ∅ or score == 0 or paths == ∅
```

Figure 3.8: MSC-α algorithm. A link is only removed when it is covered at least α times.

MSC greedily chooses the path p that covers the maximum number of uncovered links (maximize $|p \cap U|$), and removes links in p from U. The algorithm terminates when there are no links left in U.

NetSonar modifies the basic MSC algorithm to handle multipath and to enable open-loop diagnosis. To handle multipath, we group the paths by source/destination pair. In each step, *either* all links in the probabilistic cover P for a source-destination pair are chosen, *or* none of them are chosen. This is because we cannot control the coverage of an individual path in a probabilistic cover. Furthermore, recall that the number of probes in the probabilistic cover between a pair of test agents depends on the number of paths between them. Hence, when choosing a probabilistic cover P in each iteration, instead of using $|P \cap U|$ as the scoring function, we normalize $|P \cap U|$ by the number of probes to favor a P that covers new edges most economically.

We need one more twist for open-loop diagnosability. The original MSC aims to minimize the overall number of probes. However, this can introduce ambiguity in fault localization. For example, if two links l_1 and l_2 are *only* exercised by path p but no other paths, in the original MSC, both links are considered "covered" and no new path will try to exercise these links. If p encounters latency problems, it is not clear whether l_1 or l_2 is the culprit. Traditionally, such ambiguity is removed by sending additional probes after performance problems are discovered [70, 78]. However, in a large network like

```
<Testlist server="A" majorVersion="1" minorVersion="0">
  <Peer uri="test://B:1100:50000" physicalIP="10.0.0.1"
          interval="300000" level="global"/>
  <Peer uri="test://C:1100:50000" physicalIP="10.0.0.2"
          interval="300000" level="global" />
</Testlist>
```

Figure 3.9: Example XML file for a test agent.

IDN, such close-loop, multi-iteration diagnosis can be difficult to implement due to the additional latency in the measurement data processing pipeline.

NetSonar takes a different approach. We introduce a parameter α. NetSonar uses MSC-α (Figure 3.8) which requires each link to be exercised by *at least* α times, except when less than α paths can exercise a link, in which case the link is exercised maximum possible times ($\leq \alpha$). This modification can be easily done by adding a counter to each link initialized to 0. As shown in Figure 3.8, we increment the counter every time a path is chosen that covers the link, and only remove the link when the counter reaches α. Note that MSC-1 is equivalent to the original MSC, and MSC-∞ is "all-pairs" where all paths are exercised.

The intuition behind MSC-α is that by requiring each link to be covered α times, it is more likely to be covered in different combinations of links as part of different paths. Thus, we can distinguish a culprit link from other healthy links (during fault localization) by observing the health of different paths. Although MSC-α is very simple, it is a practical way to improve what is referred to as *identifiability* in the tomography literature [14], leveraging the fact that NetSonar can control the generation of test probes. As we show in Section 3.6.3, MSC-α also improves resilience to path changes.

3.4 Implementation

Each NetSonar component is kept simple for reliability and to minimize load on the servers. Every 30 minutes, the controller generates test plans in XML files and serve them via a web server. Every test agent periodically reloads its own XML file from the

controller and runs the set of tests. Results are uploaded to the data sink, which in turn uploads them to a distributed file system on which the analysis code is run directly.

Controller: The NetSonar controller, implemented in C#, contains two parts: a file generator and a web server. The file generator periodically reads network topology and LSP information. It then uses the test plan generation algorithms described before to select paths that can cover all links. Finally, these paths are clustered by the source test agents; each test agent has a dedicated XML file containing a list of test targets.

Figure 3.9 shows an example XML file that is run by Host A's test agent. In this example, Host A should send traceroute every 300 seconds to Host B with remote port 1100 and local port 50000. The "physicalIP" field is used to avoid unnecessary DNS queries from a large number of agents. The "level" field identifies a test level. For example, a test agent can participate in both "global" and "north_america" tests. Data reported to the data sink is tagged with the "level" field. The web server hosts all the XML files so they can be downloaded by test agents; it does not participate in test plan generation.

Test agent: Each test agent, implemented in C++, contains a management module and several test clients. Periodically, the management module downloads its XML file from the controller, and dispatches the tests to the two test clients used by NetSonar: TCP ping and TCP traceroute. The module is also responsible for uploading test results to the data sink.

Data sink: The NetSonar data sink is also a web server. The test agents `POST` 2KByte test results each time in the form of comma-separated values, including timestamp, source/destination IP and port, latency, path, and other metadata. The results are uploaded to a distributed file system for post-analysis.

Practical considerations: Being part of a global DC infrastructure, NetSonar is implemented for scalability and reliability. First, both the controller and the data sink are implemented as simple, distributed web services. For example, each logical controller is actually a cluster of machines that hosts the same set of XML files. The HTTP `GET` requests from test agents are automatically load balanced across different machines, to avoid the failure of a single machine bringing down the entire system. A watchdog program monitors the health of machines and removes bad ones from the system.

Second, the test agent is fault tolerant. To avoid a malfunctioning controller disrupting the entire network, test agents have a local hard limit on the number of tests they can generate per minute. If an agent loses its connection to the controller or the data sink, it uses the cached XML file to continue testing and buffers the results locally, but will stop testing after a certain period of time. Following conventional fault-tolerant design practice, test agents, controllers, and data sinks are distributed across different availability zones, and do not share top-of-rack switches or power supplies.

Finally, the HTTP interface between different components uses plain-text and is standard-compliant, facilitating debugging and profiling. It also allows components to evolve independently. For example, we have developed new test plan generation algorithms in the controller without touching the data sink or test agents.

3.5 Deployment and Evaluation

We deployed NetSonar as part of IDN's DC management system. As a result, test agents are automatically deployed to *all* hosts in a DC and co-exist with other production services. This approach is unlike some other monitoring systems, which only deploy a small number of dedicated vantage points. It helps to cover more routers and links. However, these machines may be also running other services such as web indexing, which means that their test results can be inaccurate when the servers are busy.

Four machines, acting as both the controller and the data sink, manage the entire NetSonar system. Every 30 minutes, the controller extracts the network topology and LSP information from various data sources and re-generates XML files for test agents. Our current deployment is limited to a modest number (8) of DCs.

We report on incidents captured by NetSonar during a one-month period spanning Feb and March, 2013.

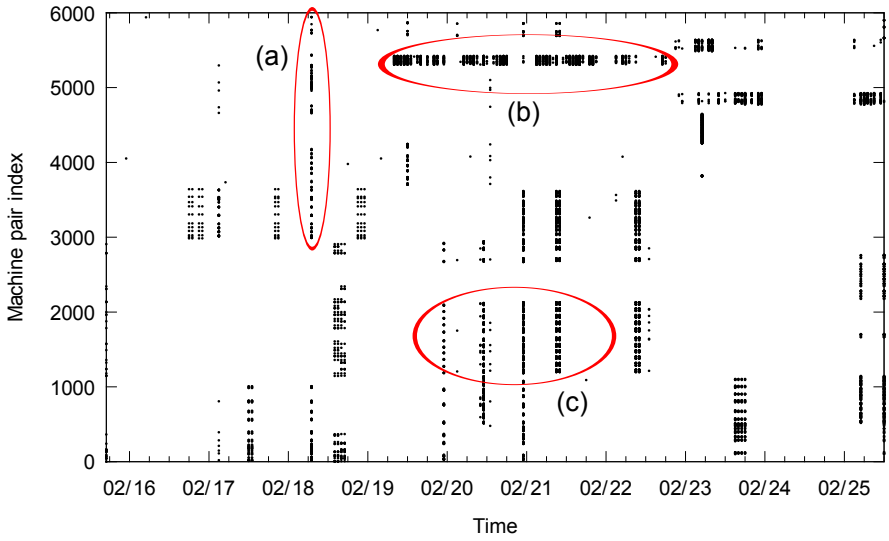


Figure 3.10: Spikes detected by NetSonar, showing (a) vertical strips, (b) horizontal strips, and (c) periodic spikes. Neighboring devices are given consecutive indexes.

3.5.1 Failure Characteristics

Part of the incident heatmap generated by NetSonar is shown in Figure 3.10. Each black dot indicates the time and the machine pair that experiences a latency spike. The y-axis is machine-pair index number ordered first by the source host name and then by the destination host name. Due to this naming scheme, hosts in the same DC will be adjacent to each other. In the graph, vertical strips indicate an incident affecting many machine pairs at the same time, and the horizontal strips indicate an incident affecting a small number of machine pairs but for a long time.

We observed the following using NetSonar's output:

Frequent spikes: To reduce false positives, we use a conservative spike detection threshold of 50 ms, which means that each dot represents at least a 50 ms latency jump from the average of past 3 data points. Each data point is the 99th percentile latency during a one-hour ping aggregation window. Even with such a conservative threshold, over the entire one-month period, there are over 40,000 machine-pair spikes. Each spike is not a unique incident, as an incident can cause many spikes (across time or machine-pairs).

Bad agents: To increase coverage, NetSonar agents co-locate with production machines. Even if we used dedicated machines as test agents, performance impacting incidents such as software updates, hardware failures, and power cycling are inevitable. Hence, some test agents experience sustained periods of high latency. The horizontal strip (b) in Figure 3.10 indicates one such "bad test agent" during Feb 19 - 22. The width of the strip indicates that this test agent experienced high latency when communicating with all other test agents.

Periodicity: There are some periodic, vertical strips (c) in the lower half of Figure 3.10. The interval between the spikes are roughly 12 hours, at 12AM and 12PM respectively. Moreover, the spike incident was DC wide. This is likely to be caused by periodic application level activities, such as website indexing. Some of these activities affect test agents directly, while others generate cross-traffic affecting test agents.

3.5.2 Validation

To validate the root cause of spike incidents inferred by NetSonar, we manually cross-verify the top 5 suspects of each incident using other data sources such as SNMP counters. NetSonar captured 140 incidents during Feb-March, 2013. Of these incidents, 84 were localized to edge devices. That means either the top suspect was the machine itself or there were local issues *inside* the DCs. We could not validate these incidents due to the lack of reliable, independent information for cross-verification. The remaining 66 incidents were localized to core devices. The most common problem was troubled link, e.g., due to large, bursty inter-DC transfers. 56 out of 66 incidents can be validated using SNMP counters and other data sources; that is, these other data sources confirmed that the router interface inferred as culprit by NetSonar was indeed overloaded or dropping packets (without being overloaded) at the time of the incident.

We cannot validate the remaining 10 core incidents with the data sources we have. This does not necessarily mean that these problems were incorrectly localized. Due to the lack of the ground truth, we can only validate NetSonar alerts that correlate well with other available data sources.

To provide insight into the nature of faults, we report on 3 typical incidents in detail.

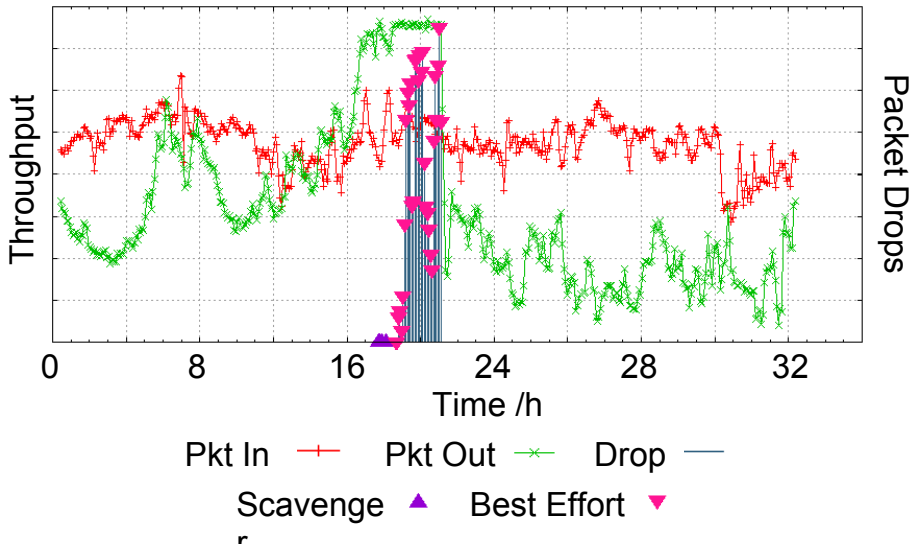


Figure 3.11: A congestion incident captured by NetSonar. SNMP counters show packet drops. This is due to a sudden increase of outgoing packets. Only relative scales are shown.

Case 1: Congestion Figure 3.11 illustrates the router counters in a typical troubled link incident reported by NetSonar. The outgoing traffic increases by 2x in just a few minutes and packets started being dropped. Noticing the increased latency due to the congestion, NetSonar successfully triangulated to this interface. This is likely due to an unplanned inter-DC activity initiated by the application. The traffic went back to normal after about 3 hours.

Case 2: Packet drops without obvious reasons Figure 3.12 shows a packet drop incident without obvious reasons. NetSonar found that the latency between various DCs had increased. It narrowed down the suspect to the link between two DCs. Validation showed a short-lived packet loss of hundreds of packets per second for around 5-10min, which triggered the 99th percentile latency spikes. However, the root cause was not clear from SNMP counters alone, because the traffic was relatively stable during this period. In NetSonar we do not directly measure packet loss. Instead, packet loss is captured as higher latency due to TCP retries.

Case 3: Fiber cut Due to a maintenance-related activity by a circuit provider, the link connecting two sites B and C (Figure 3.13) lost 20 Gbps of bandwidth. Because there

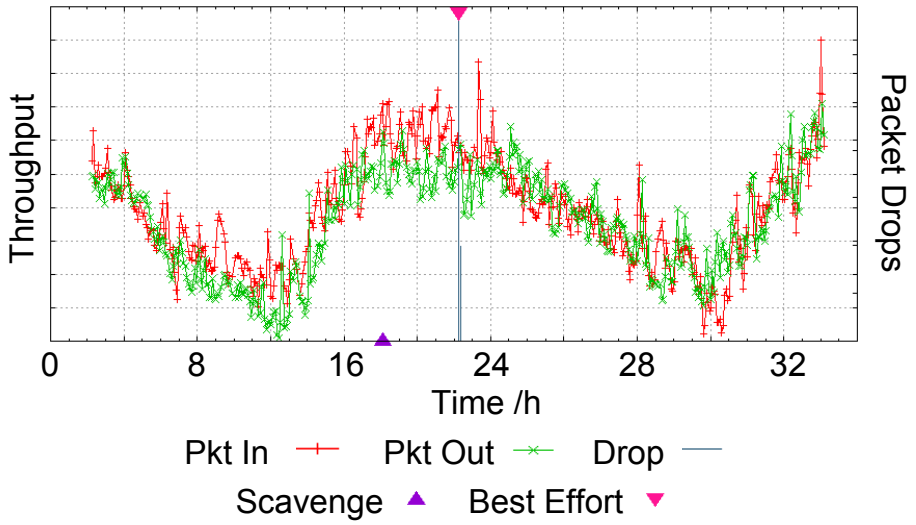


Figure 3.12: A sudden packet drops within just a few minutes. However, the root cause is unclear since the incoming and outgoing traffic are stable according to SNMP counters.

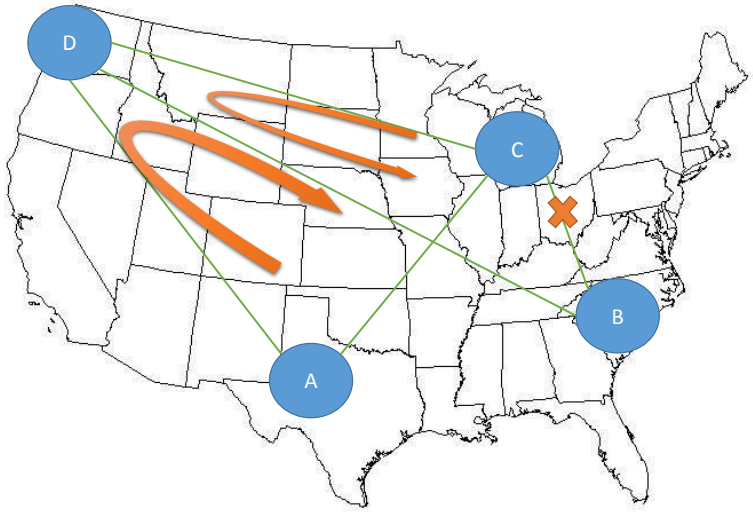


Figure 3.13: A fiber cut between two DCs C and B causes the traffic to reroute to west coast. Figure does not represent the actual DC locations

are multiple links between B and C, and only a portion of the links were cut, one might not correctly localize the problem with just ping and traceroute. In fact, NetSonar captured the latency increase between these two DCs using the troubled link detector and pinpointed the culprit as devices in D since they appeared in several spikes. However, at the same time, NetSonar's latency inflation detector captured the route changes from B-C to B-D-C, and A-C-B to A-D-B. By combining both results, we were able to correctly understand what was happening.

3.5.3 Comparison with SNMP counters

Today, many large network operators still rely on SNMP counters as a primary tool for network monitoring, which can be surprising given the significant progress in network tomography and other monitoring techniques. In our evaluation, SNMP counters are also an important data source to cross-validate the findings of NetSonar. However, we find that *solely* using SNMP counters for monitoring, without additional testing, is inefficient and often misleading.

To illustrate the shortcomings of SNMP counters, we compared alerts from April, 2013 reported by SNMP counters and NetSonar. SNMP counters are polled every 5 minutes, while NetSonar's ping aggregation window is 60 minutes. To enable fair comparison, we aggregated all SNMP alerts in a 60-minute window as *one* alert. The SNMP alerts were generated based on two thresholds: 1) when link utilization exceeds 90% of capacity; 2) when the error or drop rate is higher than 1,000 packets per minute.

We found 36 NetSonar alerts during this period, where 30 of them could be verified by SNMP counters; we could not identify the root cause of the remaining 6 latency incidents. On the other hand, we saw 1,052 high utilization alerts, and 2,091 error/drop alerts from SNMP counters, most of which do not have any impact on end-to-end latency as measured by NetSonar. The total number of SNMP alerts is almost 87x the number of NetSonar alerts. It equates to 104.8 alerts per day, which is well beyond what operators can handle.

The key reason for such a high false positive rate is that SNMP alerts are generated based on 5-minute aggregate data and hence does not always reflect packet-level

end-to-end performance. For example, a 90%+ link utilization may look high but if the traffic is smooth, there could be very little queuing or congestion. Similarly, 1,000 error/dropped packets per minute may look high. But consider a 10 Gbps link rate and 1,500 bytes packet size; the average error/drop rate over the 5-minute interval is merely 0.002% which is again barely noticeable from the perspective of an individual flow. Raising the two thresholds would reduce false positives, but it would also dramatically increase the false negatives and cause us to miss most of the NetSonar alerts. In fact, there simply does not exist an ideal "threshold" that can achieve both low false positive rate and low false negative rate. This problem was also echoed in our conversations with IDN operators.

3.6 Simulation

In this section, we evaluate NetSonar's ability to locate troubled links using simulations. We first evaluate the effectiveness of probabilistic path covers and diagnosable link covers. We then study the impact of latency spike threshold, measurement noise, ping aggregation window, and traceroute frequency.

The topology and MPLS configuration used in our simulations are based on IDN. In all tables below, the "$\leq N$" columns mean the fraction of trials in which NetSonar captures the culprit in the top N suspects. All results are based on 1,000 trials. The "Total" and "Max" columns denote the total number of probes in the network and the maximum number of probes traversing a single router *in one "round"* (during which all links are tested at least once). Unless stated otherwise, we pick an inflation factor of 2 for probabilistic covers, use MSC-2 in diagnosable covers, and simulate a single faulty link.

3.6.1 Accuracy and Overhead

Probabilistic path covers: A large inflation factor can increase the path coverage, but it also adds probing overhead. Table 3.1 shows how inflation factor in probabilistic cover affects performance. The total and maximum number of probes grows almost linearly

Inflat. factor	Accuracy (%)			Probes (#/round)	
	≤ 1	≤ 2	≤ 3	Total	Max
0.5	76.9	82.4	84.7	2,401	280
1	89.8	94.1	96.1	4,813	582
2	92.1	96.0	98.8	9,626	1,100
3	95.7	97.6	99.8	14,439	1,653

Table 3.1: Accuracy and overhead of probabilistic covers. "Inflation factor" indicates the number of 5-tuples chosen per site pair, normalized by the number of LSPs between two sites.

Algorithm	Accuracy (%)			Probes (#/round)	
	≤ 1	≤ 2	≤ 3	Total	Max
MSC-1	74.6	87.1	94.9	8,073	800
MSC-2	92.1	96.0	98.8	9,626	1,100
MSC-3	96.0	98.8	99.9	18,957	2,206
All-pairs	98.5	99.3	99.9	51,621	7,054

Table 3.2: Accuracy and overhead of diagnosable covers with different α values.

with the inflation factor. NetSonar can achieve high accuracy when inflation factor > 1. This is because test agents are present in all sites. Even when some paths are not covered by one test agent pair (because a relatively small inflation factor is chosen), MSC-2 ensures that there are some other test agent pairs which will exercise the links on those uncovered paths.

Diagnosable link covers: α determines how redundant a diagnosable link cover is. Table 3.2 compares the diagnosable link covers under different α values.

Overall, MSC-2 attains a good balance between overhead and accuracy. Compared to MSC-1, MSC-2 increases the accuracy to over 90% with only 1.4x probing overhead. Further increase in α has little effect on accuracy and only results in larger probing overhead.

Multiple faulty links: Multiple simultaneous faulty links, while less common, can occur in a large network. We evaluate how NetSonar performs with multiple faulty links by selecting top N or top X% suspects inferred by Sherlock. Table 3.3 shows that, with 2 or 3 faulty links, NetSonar may not be able to accurately localize the culprits to the

N	Accuracy (%)			
	$\leq N$	0.5%	1%	2%
1	92.1	100	100	100
2	46.4	85.1	94.5	97.6
3	19.1	56.2	79.6	92.9

Table 3.3: Accuracy with multiple failures. N denotes the number of simultaneous failures. The percentage in "Accuracy" column denotes the failures fall in top X% suspects.

Algorithm	Accuracy (%)			Coverage (%)
	≤ 1	≤ 2	≤ 3	
NetSonar	92.1	96.0	98.8	100
All-pairs+Single Path	60.9	68.4	73.8	78.2
Random pairs+Prob. Cover	71.9	79.7	82.8	84.5

Table 3.4: NetSonar provides higher accuracy and coverage compared to either ignoring multipathing or using unplanned tomography.

top 2 or 3 suspects. However, in over 90% of the cases, NetSonar can still capture the culprits within the top 2% candidates. In other words, NetSonar can still successfully help operators eliminate a vast majority of the links from the suspect set.

3.6.2 Multipathing and Planned Tomography

We compare NetSonar's coverage techniques to two simple alternatives suggested by the existing literature. First, most existing planned tomography schemes (e.g., [8, 78]) assume a single forwarding path between sites, and do not consider multipathing. To simulate this, we pick one random path from the set of paths between two sites for measurement. Table 3.4 shows that, even with these all-pairs measurements, ignoring multipath causes diagnosis accuracy to drop by more than 30% compared to NetSonar. The lack of probabilistic path covers and the single path assumption results in only 78.2% of links being covered.

Second, we assume a probabilistic cover for multipathing and consider using unplanned tomography, where the measurements are not designed using a test plan (Section 2.7). We simulate unplanned tomography by randomly picking the same number of

False negatives (%)	Accuracy (%)		
	≤ 1	≤ 2	≤ 3
0	92.1	96.0	98.8
5	91.5	95.6	97.9
10	91.0	95.2	97.6
20	90.6	94.5	96.6

Table 3.5: Accuracy as false negative rate varies due to threshold selection.

pairs of test agents as NetSonar's diagnosable link cover. The random choice simulates the lack of planning. The last row in Table 3.4 shows that accuracy is 20% lower than NetSonar with the same testing overhead, again due to reduced link coverage. These experiments suggest that we need to incorporate *both* multipathing and planning to achieve better coverage.

3.6.3 Robustness and Parameters

False Negatives in Measurement: To detect a latency spike, we use a threshold to compare the 99th percentile latency in the current aggregation window to the average of the past three data points. To prevent raising too many *false positives*, we pick a fairly conservative threshold (50 ms) to filter out most small latency variations. However, such a conservative threshold may generate *false negatives*, i.e., "bad" probes that traverse a faulty link are mistakenly labeled as "good". In the following simulations, we randomly and deliberately relabel a certain percentage of "bad" probes as "good".

Table 3.5 shows how NetSonar performs under different percentage of false negatives. Surprisingly, even with 20% of false negatives, the ≤ 3 accuracy is still above 95%. This is because with MSC-2, one faulty link will be covered by multiple probes between different test agent pairs. Thus, a small percentage of false negatives will have little impact on fault localization accuracy, as long as the faulty link is captured by a sufficient number of other probes. This result also indicates that a conservative latency spike threshold works well in practice.

Bad agents: False positives can arise from bad agents, which would nudge Sherlock away from the actual culprit. In this simulation, we assume a certain fraction of bad

Bad agents (%)	**Accuracy (%)**		
	≤ 1	≤ 2	≤ 3
0	92.1	96.0	98.8
0.25	90.0	94.6	95.4
0.5	79.3	88.0	92.7
1	44.7	59.2	72.9

Table 3.6: Accuracy as agent noise varies.

Path changes	≤ 2 **Accuracy (%)**		
	MSC-1	MSC-2	MSC-3
0	87.1	96.0	98.8
1	74.6	88.6	95.3
2	72.6	84.0	93.7
3	65.2	81.3	84.8

Table 3.7: Accuracy as the number of path changes per aggregation window varies for various values of α in diagnosable link cover. Only ≤ 2 accuracy is shown.

agents which will distort latency measurements, with a probability varying from 10% to 90%. Table 3.6 shows that 0.5% of bad agents will cause noticeable drop in localization accuracy. Because of the use of 99th percentile latency in spike detection, even 10% of distorted latency measurements will trigger false positives.

We identify bad agents by examining all latency measurements from the same agent. If an agent reports latency spikes from all of its probes (to different targets), we will discard all of its latency measurements. By applying this simple trick, we can restore the localization accuracy to the level when there is no false negative (first row in Table 3.6).

Path changes within aggregation windows: Path changes may occur during an aggregation window. In other words, the latency measurements may actually correspond to multiple paths. In the following simulations, we introduce path changes by randomly picking a new LSP within the same pair of test agents. We then vary the number of path changes in an aggregation window. When a latency spike is detected, NetSonar will blame *all* paths appearing in the same window.

TR Period (min)	Accuracy (%)			Overhead (pps)	
	≤ 1	≤ 2	≤ 3	Total	Max
0	92.1	96.0	98.8	∞	∞
5	89.3	95.8	98.1	32.1	3.7
10	87.5	94.1	96.3	16.0	1.8
20	82.4	90.3	94.0	8.0	0.9

Table 3.8: Accuracy as traceroute frequency varies

Table 3.7 shows how path changes affect accuracy and overhead. The first row represents the ideal case where we can exactly map each latency measurement to the correct path. When the number of changes grows, accuracy drops. We see that the diagnosable link cover provides extra protection against path changes. For example, in the last row, MSC-2 has 16.1% higher accuracy compared with MSC-1 when there are 3 path changes in the same aggregation window. In a diagnosable link cover, the same link is tested independently by multiple probes. Even if probes between certain pairs of test agents fail to map to a correct link, other probes traversing the same link can still provide accurate identification of a troubled link.

Path changes between traceroutes: If a path's lifetime (from the time when a new path is set up to the time when the path is torn down) is shorter than the traceroute probing period, such a path will not be captured by traceroute and some latency measurements may be mapped to incorrect paths. In the following simulations, we again introduce path changes as in the last experiment but with a median life time of 15 minutes. In Table 3.8, the first row (0 min TR period) denotes the ideal case where we can associate each ping with the correct path at the cost of traceroutes sent at an infinite frequency. We can see a traceroute period between 5 and 10 minutes strikes a good balance between overhead and accuracy.

3.7 Limitations

Matching measurements: NetSonar combines ping and traceroute but ping measures round-trip latency, while traceroute only reports forward path information. This mismatch can lower the diagnosis accuracy - the long ping latency caused by bad reverse paths may be incorrectly attributed to the forward path reported by traceroute. This problem can be solved by applying One-Way-Ping [68], or probing reverse path information with a reversed 5-tuple (swapping source/destination IP and TCP port) by the receiving test agent.

Single administrative domain: NetSonar assumes the entire network belongs to the same administrative domain and that the network operator knows the topology as well as basic forwarding information (such as LSPs) in advance. Otherwise, the host selection algorithm cannot perform path and link covers, which may reduce the coverage and diagnosis accuracy. In this case, the system has to fallback to all-pairs probing, where no attempt is made to reduce probing overhead.

IP level probing: NetSonar relies on IP level probing (traceroute) to map pings to physical paths. For some devices, such probing may not be as useful. For example, the Ethernet level link aggregation groups (LAGs) prevent traceroute from discovering the physical links used, since the entire group is a single IP link.

Uneven hashing: In Section 3.3.3, we assume that hash functions always distribute traffic to all next-hops uniformly, or at least with a known distribution as in weighted cost multipath (WCMP). This may not be true in some cases. For example, a broken hash function can create unknown, non-uniform traffic distribution, which may lower NetSonar's path/link coverage.

Intra-DC diagnosis: NetSonar performs well in troubleshooting inter-DC performance problems, and in theory it can also be used in intra-DC fault localization. However, our trial deployment experience reveals that while the cover algorithms work, there are still several practical difficulties. First, intra-DC latency is extremely small (usually much less than 1ms), hence the measurements are much more sensitive to machine "hiccups". By comparison, inter-DC latency is usually more than 20 ms. Second, intra-DC distributed applications, such as web indexing, can simultaneously overload many

devices. Such behavior is likely to confuse our fault localization algorithm.

3.8 Summary

NetSonar is a large-scale network tester: it treats the whole network as a device-under-test, and automatically generates high-coverage test plans. NetSonar handles the complexities of real networks including changing paths and multipath routing. NetSonar is also a *gray box* tester: it utilizes partial forwarding information and deals with the remaining uncertainty by computing efficient probabilistic and diagnosable covers that allow diagnosis in a single pass. Both probabilistic and diagnosable covers are general ideas that apply to networks that belong to a single administrative domain. Beyond these ideas, to the best of our knowledge, NetSonar is the first tester to report deployment experience in a large production inter-DC network beyond simulations [5, 41] and testbeds [32].

Many aspects of NetSonar's design are driven by imperfect knowledge of the data plane today. We do not know the hash functions used by multipath routing; so we need to use traceroute to map pings to paths. Router CPUs are underpowered and responsible for other critical tasks such as route computation; so we can only infrequently harvest snapshots of the data plane to bootstrap testing. One might argue that these problems are temporary, and will disappear with newer router hardware and the stronger consistency between data and control planes that SDN provides.

However, even in a future network with perfect technology, we believe some uncertainty is inevitable, at least for non-technical reasons such as policies and organizational boundaries. Thus, gray box testing will still be needed to achieve trustworthy results. Moreover, our experience with NetSonar shows that, contrary to what one may think at first, accounting for uncertainty is practical. It can be done with minimum overhead while achieving high accuracy. We hope that NetSonar is a first step towards building tools that balance what can be known about networks with their unknowns.

Chapter 4

Static Checking in Large Networks

Divide and Conquer.

Philip II of Macedon (359 – 336 BCE)

ATPG and NetSonar are both dynamic checkers. While these checkers can discover many run-time problems such as congestion or a fiber cut, they have the overhead of test agent deployment and test plan execution. By contrast, static checkers treat the network as "read-only": they only collect forwarding rules and topology, without injecting test packets into the network. In small networks, the implementation of static checkers can be straightforward. However, as the network grows larger, static checkers face two problems:

1. How to take a stable snapshot of forwarding rules across numerous devices, so that the snapshot faithfully reflects the actual state of the network?

2. How to check the data plane properties, such as loop freedom, within a reasonable time frame?

Data center networks are the perfect examples to verify the scalability of static checkers. Modern data centers can employ 10,000 switches or more, each with its own forwarding table. In such a large network, failures are frequent: links go down, switches reboot, and routers may hold incorrect prefix entries. Whether routing entries are written by a distributed routing protocol (such as OSPF) or by a remote route server(s), the routing state is so large and complex that errors are inevitable. We have seen logs from a production data center reporting many thousands of routing changes per day, creating plenty of opportunity for error.

Data centers withstand failures using the principles of scale-out and redundant design. However, the underlying assumption is that the system reacts correctly to failures. Dormant bugs in the routing system triggered by rare boundary conditions are particularly difficult to find. Common routing failures include routing loops and black-holes (where traffic to one part of the network disappears). Some errors only become visible when an otherwise benign change is made. For example, when a routing prefix is removed it can suddenly expose a mistake with a less specific prefix.

Routing errors should be caught quickly before too much traffic is lost or security is breached. We therefore need a fast and scalable approach to verify correctness of the entire forwarding state. A number of tools have been proposed for analyzing networks including HSA [38], Anteater [49], NetPlumber [37] and Veriflow [39]. These systems take a snapshot of forwarding tables, then analyze them for errors. We first tried to adopt these tools for our purposes, but ran into two problems. First, they assume the snapshot is consistent. In large networks with frequent changes to routing state, the snapshot might be inconsistent because the network state changes while the snapshot is being taken. Second, none of the tools are sufficiently fast to meet the performance requirements of modern data center networks. For example, Anteater [49] takes more than 5 minutes to check for loops in a 178-router topology.

Hence, we set out to create Libra, a fast, scalable tool to quickly detect loops, black-holes, and other reachability failures in networks with tens of thousands of switches. Libra is much faster than any previous system for verifying forwarding correctness in a large-scale network. Our benchmark goal is to verify all forwarding entries in a 10,000 switch network with millions of rules in minutes.

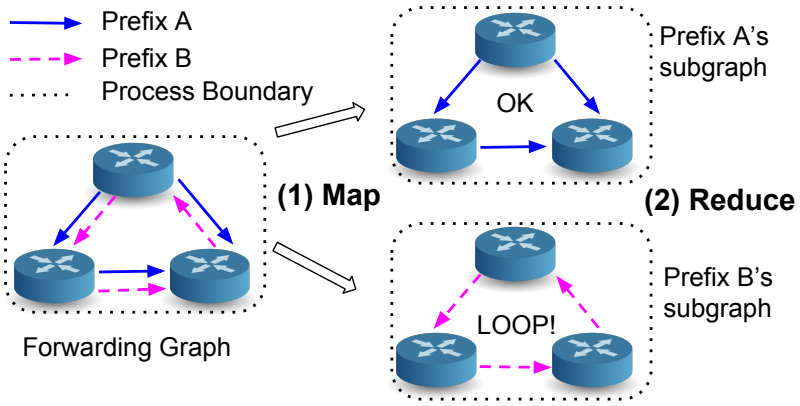


Figure 4.1: Libra divides the network into multiple forwarding graphs in mapping phase, and checks graph properties in reducing phase.

We make two main contributions in Libra. First, Libra capture stable and consistent snapshots across large network deployments, using the event stream from routing processes (Section 4.2). Second, in contrast to prior tools that deal with arbitrarily structured forwarding tables, we substantially improve scalability by assuming packet forwarding based on longest prefix matching.

Libra uses MapReduce for verification. It starts with the full graph of switches, each with its own prefix table. As depicted in Figure 4.1, Libra completes verification in two phases. In the *map* phase, it breaks the graph into a number of slices, one for each prefix. The slice consists of only those forwarding rules used to route packets to the destination. In the *reduce* phase, Libra indepedently analyzes each slice, represented as a forwarding graph, *in parallel* for routing failures.

We evaluate Libra on the forwarding tables from three different networks. First, "DCN" is an emulated data center network with 2 million rules and 10,000 switches. Second, "DCN-G" is made from 100 replicas of DCN connected together; i.e., 1 million switches. Third, "INET" is a network with 300 IPv4 routers each contains the full BGP table with half a million rules. The results are encouraging. Libra takes one minute to check for loops and black-holes in DCN, 15 minutes for DCN-G and 1.5 minutes for INET.

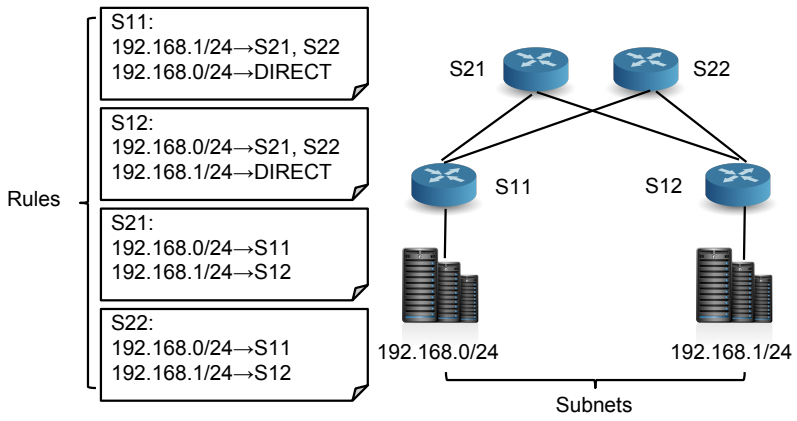


Figure 4.2: Small network example for describing the types of forwarding error found by Libra.

4.1 Forwarding Errors

A small toy network can illustrate three common types of error found in forwarding tables. In the two-level tree network in Figure 4.2 two top-of-rack (ToR) switches (S11, S12) are connected to two spine switches (S21, S22). The downlinks from S11 and S12 connect to up to 254 servers on the same /24 subnet. The figure shows a "correct" set of forwarding tables. Note that our example network uses multipath routing. Packets arriving at S12 on the right and destined to subnet 192.168.0/24 on the left are load-balanced over switches S21 and S22. Our toy network has 8 rules, and 2 subnets.

A *forwarding graph* is a directed graph that defines the network behavior for each subnet. It contains a list of (local_switch, remote_switch) pairs. For example, in Figure 4.3(a), an arrow from S12 to S21 means the packets of subnet 192.168.0/24 can be forwarded from S12 to S21. Multipath routing can be represented by a node that has more than one outgoing edge. Figure 4.3(b)-(d) illustrates three types of forwarding error in our simple network, depicted in forwarding graphs.

Loops: Figure 4.3(b) shows how an error in S11's forwarding tables causes a *loop*. Instead of forwarding 192.168.0/24 down to the servers, S11 forwards packets up, i.e., to S21 and S22. S11's forwarding table is now:

```
192.168.0/24 → S21, S22
192.168.1/24 → S21, S22
```

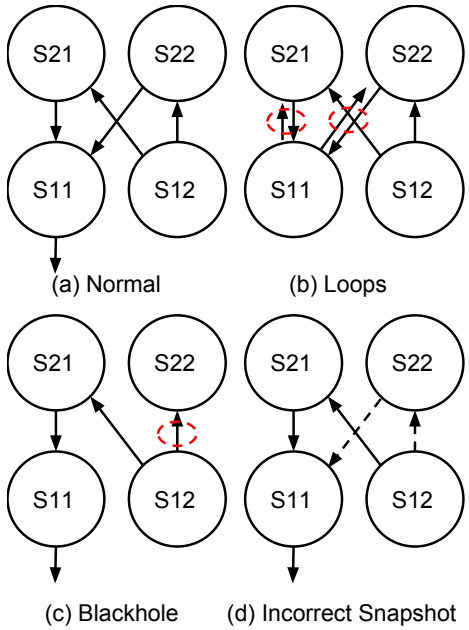


Figure 4.3: Forwarding graphs for 192.168.0/24 as in Figure 4.2, and potential abnormalities.

The network has two loops: S21-S11-S21 and S22-S11-S22, and packets addressed to 192.168.0/24 will never reach their destination.

Black-holes: Figure 4.3(c) shows what happens if S22 loses one of its forwarding entries: `192.168.0/24` → `S11`. In this case, if S12 spreads packets destined to 192.168.0/24 over both S21 and S22, packets arriving to S22 will be dropped.

Incorrect Snapshot: Figure 4.3(d) shows a subtle problem that can lead to *false positives* when verifying forwarding tables. Suppose the link between S11-S22 goes down. Two events take place (shown as dashed arrows in the figure): $e1$: S22 deletes `192.168.0/24` → `S11`, and $e2$: S12 stops forwarding packets to S22. Because of the asynchronous nature of routing updates, the two events could take place in either order $(e1, e2)$ or $(e2, e1)$. A snapshot may capture one event, but not the other, or might detect them happening in the reverse order.

The sequence $(e1, e2)$ creates a temporary blackhole as in Figure 4.3(c), whereas the desired sequence $(e2, e1)$ does not. To avoid raising an unnecessary alarm (by detecting $(e1, e2)$ even though it did not happen), or missing an error altogether (by incorrectly

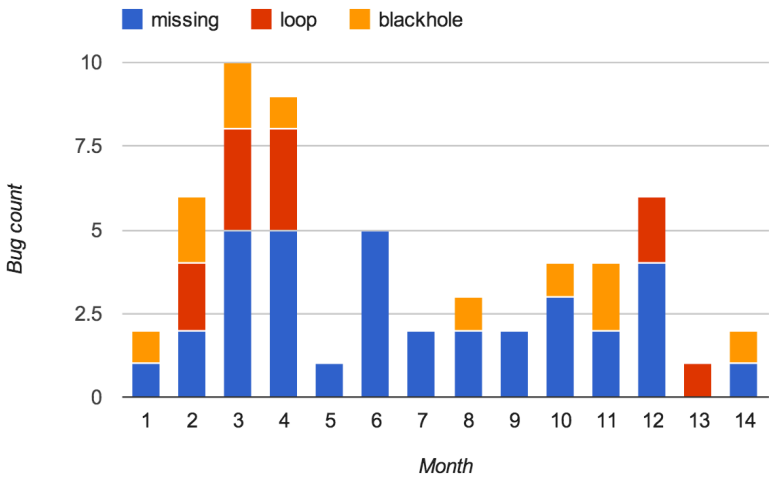


Figure 4.4: Routing related tickets by month and type.

assuming that $(e2, e1)$ happened), Libra must detect the correct state of the network.

4.1.1 Real-world Failure Examples

To understand how often forwarding errors take place, we examined a log of "bug tickets" from 14 months of operation in a large Google data center. Figure 4.4 categorizes 35 tickets for missing forwarding entries, 11 for loops, and 11 for black-holes. On average, four issues are reported per month.

Today, forwarding errors are tracked down by hand which - given the size of the network and the number of entries - often takes many hours. And because the diagnosis is done after the error occurred, the sequence of events causing the error has usually long-since disappeared before the diagnosis starts. This makes it hard to reproduce the error.

Case 1: Detecting Loops. One type of loop is caused by prefix aggregation. Prefixes are aggregated to compact the forwarding tables: a cluster E can advertise a *single* prefix to reach all of the servers connected "below" it to the core C, which usually includes the addresses of servers that have not yet been deployed. However, packets destined to these non-deployed addresses (e.g., due to machine maintenance) can get stuck in loops. This is because C believes these packets are destined to E, while E lacks the forwarding rules

to digest these packets due to the incomplete deployment, instead, *E*'s default rules lead packets back to *C*.

This failure does not cause a service to fail (because the service will use other servers instead), but it does degrade the network causing unnecessary congestion. In the past, these errors were ignored because of the prohibitive cost of performing a full cluster check. Libra can finish checking in less a minute, and identify and report the specific switch and prefix entry that are at risk.

Case 2: Discovering Black-holes. In one incident, traffic was interrupted to hundreds of servers. Initial investigation showed that some prefixes had high packet loss rate, but packets seemed to be discarded randomly. It took several days to finally uncover the root cause: A subset of routing information was lost during BGP updates between domains, likely due to a bug in the routing software, leading to black-holes.

Libra will detect missing forwarding entries quickly, reducing the outage time. Libra's stable snapshots also allow it to disambiguate temporary states during updates from long-term back-holes.

Case 3: Identifying Inconsistencies. Network control runs across several instances, which may fail from time to time. When a secondary becomes the primary, it results in a flurry of changes to the forwarding tables. The changes may temporarily or permanently conflict with the previous forwarding state, particularly if the changeover itself fails before completing. The network can be left in an inconsistent state, leading to packet loss, black-holes and loops.

4.1.2 Lessons Learned

Simple things go wrong: Routing errors occur even in networks using relatively simple IP forwarding. They also occur due to firmware upgrades, controller failure and software bugs. It is essential to check the forwarding state *independently*, outside the control software.

Multiple moving parts: The network consists of multiple interacting subsystems. For example, in case 1 above, Intra-DC routing is handled locally, but routing is a global property. This can create loops that are hard to detect locally within a subsystem. There

are also multiple network controllers. Inconsistent state makes it hard for the control plane to detect failures on its own.

Scale matters: Large data center networks use multipath routing, which means there are many forwarding paths to check. As the number of switches, N, grows the number of paths and prefix tables grow, and the complexity of checking all routes grows with N^2. It is essential for a static checker to scale linearly with the network.

4.2 Stable Snapshots

It is not easy to take an accurate snapshot of the forwarding state of a large, constantly changing network. But if Libra runs its static checks on a snapshot of the state that never actually occurred, it will raise false alarms and miss real errors. We therefore need to capture - and check - a snapshot of the global forwarding state that actually existed at one instant in time. We call these *stable snapshots*.[1]

When is the state stable? A large network is usually controlled by multiple *routing processes*,[2] each responsible for one or more switches. Each process sends timestamped updates, which we call *routing events*, to add, modify and delete forwarding entries in the switches it is responsible for. Libra monitors the stream of routing events to learn the global network state.

Finding the stable state of a *single* switch is easy: each table is only written by one routing process using a single clock, and all events are processed in order. Hence, Libra can reconstruct a stable state simply by replaying events in timestamp order.

By contrast, it is not obvious how to take a *globally* stable snapshot of the state when different routing processes update their switches using different, unsynchronized clocks. Because the clocks are different, and events may be delayed in the network, simply replaying the events in timestamp order can result in a state that did not actually occur in practice, leading to false positives or missed errors (Section 4.1).

[1] Note that a stable snapshot is not the same as a *consistent* snapshot [15], which is only one *possible state* of a distributed system that might not actually have occurred in practice.

[2] Libra only considers processes that can directly modify tables. While multiple high-level protocols can co-exist (e.g., OSPF and BGP), there is usually one common low-level table manipulation API.

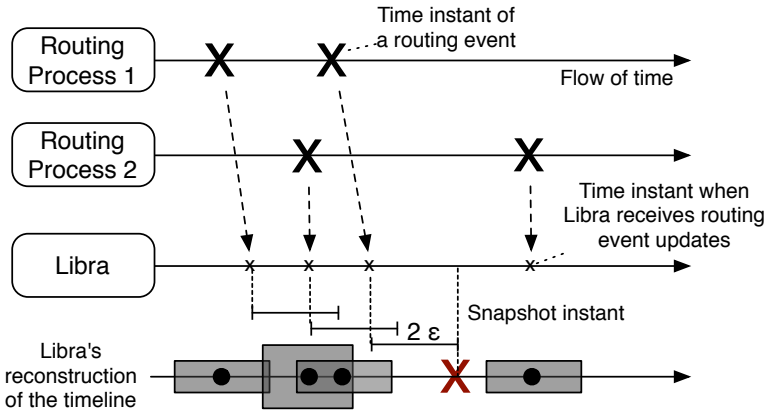


Figure 4.5: Libra's reconstruction of the timeline of routing events, taking into account bounded timestamp uncertainty ϵ. Libra waits for twice the uncertainty to ensure there are no outstanding events, which is sufficient to deduce that routing has stabilized.

However, even if we can not precisely synchronize clocks, we can *bound* the difference between any pair of clocks with high confidence using NTP [53]. And we can bound how out-of-date an event packet is, by prioritizing event packets in the network. Thus, every timestamp t can be treated as lying in an interval $(t-\epsilon, t+\epsilon)$, where ϵ bounds the uncertainty of when the event took place.[3] The interval represents the notion that network state changes atomically at some unknown time instant within the interval.

Figure 4.5 shows an example of finding a stable snapshot instant. It is easy to see that if no routing events are recorded during a 2ϵ period we can be confident that no routing changes actually took place. Therefore, the snapshot of the current state is stable (i.e., accurate).[4]

The order of any two past events from *different* processes is irrelevant to the current state, since they are applied to different tables without interfering with each other (recall that each table is controlled by only one process). So Libra only needs to replay all events in timestamp order (to ensure events for the *same* table are played in order) to accurately reconstruct the current state.

This observation suggests a simple way to create a stable snapshot by simply waiting

[3]The positive and negative uncertainties can be different, but here we assume they are the same for simplicity.

[4]A formal proof can be found in [51, § 3.3].

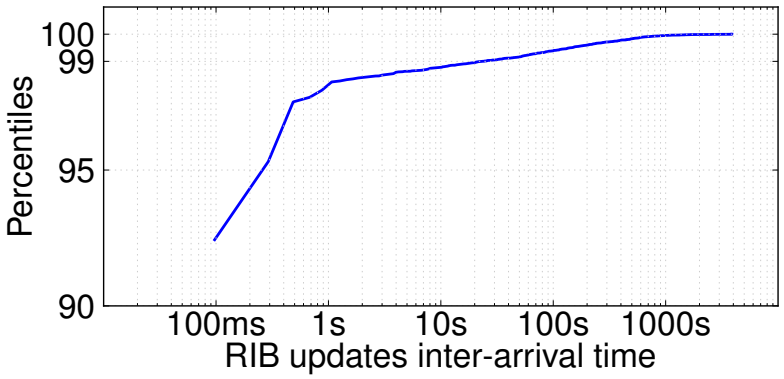


Figure 4.6: CDF of inter-arrival times of routing events from a large production data center. Routing events are very bursty: over 95% of events happen within 400ms of another event.

ϵ/ms	# of stable states	time in stable state/%
0	28,445	100.00
1	16,957	99.97
100	2,137	99.90
1,000	456	99.75
10,000	298	99.60

Table 4.1: As the uncertainty in routing event timestamps (ϵ) increases, the number of stable states decreases. However, since routing events are bursty, the state is stable most of the time.

for a quiet 2ϵ period with no routing update events.

Feasibility: The scheme only works if there are frequent windows of size 2ϵ in which no routing events take place. Luckily, we found that these quiet periods happen frequently: we analyzed a day of logs from all routing processes in a large Google data center with a few thousand switches. Figure 4.6 shows the CDF of the inter-arrival times for the 28,445 routing events reported by the routing processes during the day. The first thing to notice is the burstiness — over 95% of events occur within 400ms of another event, which means there are long periods when the state is stable. Table 4.1 shows the fraction of time the network is stable, for different values of ϵ. As expected, larger ϵ leads to fewer stable states and smaller percentage of stable time. For example, when ϵ=100ms, only 2,137 out of all 28,445 states are stable. However, because the event

stream is so bursty, the unstable states are extremely short-lived, occupying *in total* only 0.1% (~1.5min) of the entire day. Put another way, for 99.9% of the time, snapshots are stable and the static analysis result is trustworthy.

Taking stable snapshots: The stable snapshot instant provides a reference point to reconstruct the global state. Libra's stable snapshot process works as follows:

1) Take an initial snapshot S_0 as the combination of all switches' forwarding tables. At this stage, each table can be recorded at a slightly different time.

2) Subscribe to timestamped event streams from all routing processes, and apply each event e_i, in the order of their timestamps, to update the state from S_{i-1} to S_i.

3) After applying e_j, if no event is received for 2ϵ time, declare the current snapshot S_j stable. In other words, S_0 and all past events e_i form a stable state that actually existed at this time instant.

4.3 Divide and Conquer

After Libra has taken a stable snapshot of the forwarding state, it sets out to statically check its correctness. Given our goal of checking networks with over 10,000 switches and millions of forwarding rules, we will need to break down the task into smaller, parallel computations. There are two natural ways to consider partitioning the problem:

Partition based on switches: Each server could hold the forwarding state for a cluster of switches, partitioning the network into a number of clusters. We found this approach does not scale well because checking a forwarding rule means checking the rules in many (or all) partitions - the computation is quickly bogged down by communication between servers. Also, it is hard to balance the computation among servers because some switches have very different numbers of forwarding rules (e.g. spine and leaf switches).

Partition based on subnets: Each server could hold the forwarding state to reach a set of subnets. The server computes the forwarding graph to reach each subnet, then checks the graph for abnormalities. The difficulty with this approach is that each server must hold the entire set of forwarding tables in memory, and any update to the forwarding rules affects all servers.

Libra partitions the network based on subnets, for reasons that will become clear. We observe that the route checker's task can be divided into two steps. First, Libra *associates* forwarding rules with subnets, by finding the set of forwarding rules relevant to a subnet (i.e., they are associated if the subnet is included in the rule's prefix). Second, Libra builds a forwarding graph to reach each subnet, by assembling all forwarding rules for the subnet. Both steps are embarrassingly parallel: matching is done per (subnet, forwarding rule) pair; and each subnet's forwarding graph can be analyzed independently.

Libra therefore proceeds in three steps using N servers:

Step 1 - Matching: Each server is initialized with the *entire* list of subnets, and each server is assigned $1/N$ of all forwarding rules. The server considers each forwarding rule in turn to see if it belongs to the forwarding graph to a subnet (i.e. the forwarding rule is a prefix of the subnet).[5] If there is a match, the server outputs the (subnet, rule) pair. Note that a rule may match more than one subnet.

Step 2 - Slicing: The (subnet, rule) pairs are grouped by subnet. We call each group a *slice*, because it contains all the rules and switches related to this subnet.

Step 3 - Graph Computing: The slices are distributed to N servers. Each server constructs a forwarding graph based on the rules contained in the slice. Standard graph algorithms are used to detect network abnormalities, such as loops and black-holes.

Figure 4.7 shows the steps to check the network in Figure 4.2. After the slicing stage, the forwarding rules are organized into two slices, corresponding to the two subnets 192.168.0/24 and 192.168.1/24. The forwarding graph for each slice is calculated and checked in parallel.

If a routing error occurs and the second rule in S11 becomes `192.168.0/24 → S21, S22`, the loop will show up in the forwarding graph for 192.168.0/24. S11 will point back to S21 and S22, which will be caught in graph loop detection algorithm.

Our three-step process is easily mapped to MapReduce, which we describe in the next section.

[5] Otherwise, a subnet will be fragmented by a more specific rule, leading to a complex forwarding graph. See the last paragraph in Section 4.8 for detailed discussion.

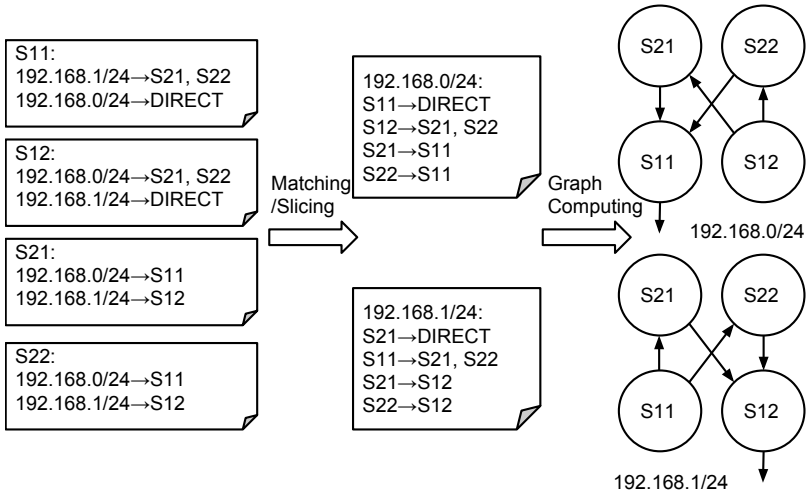


Figure 4.7: Steps to check the routing correctness in Figure 4.2.

4.4 Libra

Libra consists of two main components: a *route dumper* and a MapReduce-based *route checker*. Figure 4.8 shows Libra's workflow.

The route dumper takes stable snapshots from switches or controllers, and stores them in a distributed file system. Next, the snapshot is processed by a MapReduce-based checker.

A quick review of MapReduce: MapReduce [21] divides computation into two phases: *mapping* and *reducing*. In the mapping phase, the input is partitioned into small "shards". Each of them is processed by a mapper in parallel. The mapper reads in the shard line by line and outputs a list of `<key, value>` pairs. After the mapping phase, the MapReduce system *shuffles* outputs from different mappers by sorting by the key. After shuffling, each reducer receives a `<key, values>` pair, where `values=[value1, value2, ...]` is a list of all values corresponding to the `key`. The reducer processes this list and outputs the final result. The MapReduce system also handles checkpointing and failure recovery.

In Libra, the set of forwarding rules is partitioned into small shards and delivered to mappers. Each mapper also takes a full set of subnets to check, which by default contains all subnets in the cluster, but alternatively can be subsets selected by user. Mappers generate intermediate keys and values, which are shuffled by MapReduce. The reducers

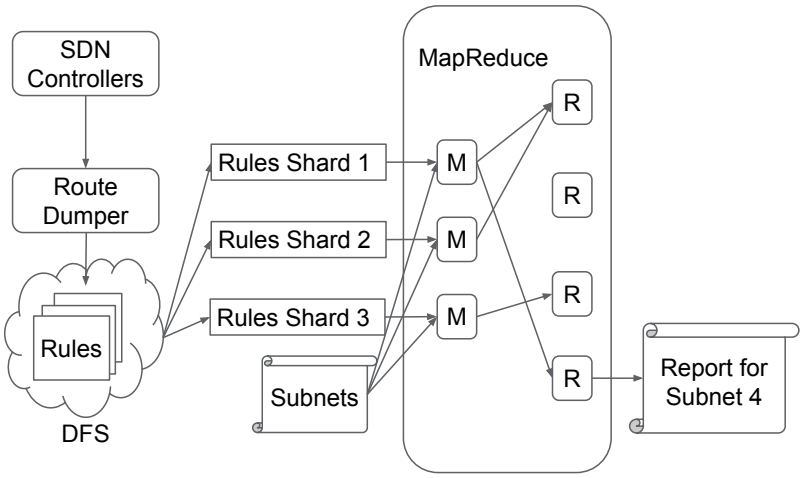


Figure 4.8: Libra workflow.

compile the values that belong to the same subnet and generate final reports.

4.4.1 Mapper

Mappers are responsible for slicing networks by subnet. Each mapper reads one forwarding rule at a time. If a subnet matches the rule, the mapper outputs the subnet prefix as the intermediate key, along with the value <rule_mask_len, local_switch, remote_switches, priority>. The following is an example (local_switch, remote_switches, priority is omitted):

```
Subnets:  192.168.1.1/32
          192.168.1.2/32
Rules:    192.168.1.0/28
          192.168.0.0/16
Outputs:  <192.168.1.1/32, 28>
          <192.168.1.1/32, 16>
          <192.168.1.2/32, 16>
```

Since each mapper only sees a portion of the forwarding rules, there may be a longer and more specific—but unseen—matching prefix for the subnet in the same forwarding table. We *defer* finding the longest matching to the reducers, which see all matching rules.

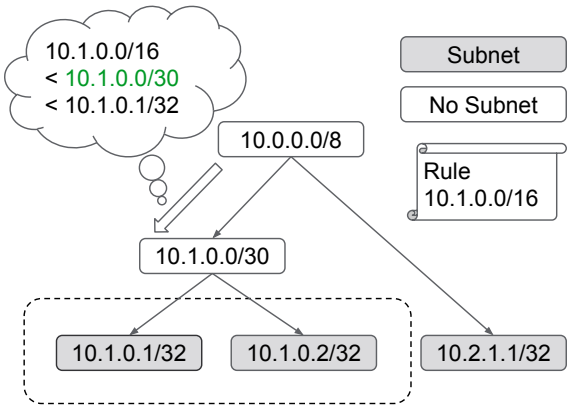


Figure 4.9: Find all matching subnets in the trie. `10.1.0.0/30` (X) is the smallest matching trie node bigger than the rule `10.1.0.0/16` (A). Hence, its children with subnets `10.1.0.1/32` and `10.1.0.2/32` match the rule.

Mappers are first initialized with a full list of subnets, which are stored in an in-memory binary trie for fast prefix matching. After initialization, each mapper takes a shard of the routing table, and matches the rules against the subnet trie. This process is different from the conventional longest prefix matching: First, in conventional packet matching, rules are placed in a trie and packets are matched one by one. In Libra, we build the trie with *subnets*. Second, the goal is different. In conventional packet matching, one looks for the longest matching rule. Here, mappers simply output *all* matching subnets in the trie. Here, matching has the same meaning—the subnet's prefix must fully fall within the rule's prefix.

We use a trie to efficiently find "all matching prefixes," by searching for the *smallest matching trie node* (called node X) that is *bigger or equal to* the rule prefix (called node A). Here, "small" and "big" refer to the lexicographical order (not address space size), where for each bit in an IP address, $wildcard < 0 < 1$. X may or may not contain a subnet. If X exists, we enumerate all its non-empty decedents (including X itself). Otherwise, we declare that there exist no matching subnets in the trie. Figure 4.9 shows an example. `10.1.0.0/30` (X) is the smallest matching trie node bigger than the rule `10.1.0.0/16` (A). Hence, its children with subnets `10.1.0.1/32` and `10.1.0.2/32` match the rule.

Proof: We briefly prove why this algorithm is correct. In an uncompressed trie, each

bit in the IP address is represented by one level, and so the algorithm is correct by definition: if there exist matching subnets in the trie, A must exist in the trie and its descendants contain all matching prefixes, which means $A = X$.

In a compressed trie, nearby nodes may be combined. A may or may not exist in the trie. If it exists, the problem reduces to the uncompressed trie scenario. If A does not exist in the trie, X (if it exists) contains all matching subnets in its descendants. This is because:

a) Any node Y smaller than X does not match A. Because there is no node bigger than A and smaller than X (otherwise X is not the smallest matching node), $Y < X$ also means $Y < A$. As a result, Y cannot fall within A's range. This is because for Y to fall within A, all A's non-wildcard bits should appear in Y, which implies $Y \geq A$.

b) Any node Y bigger than the biggest descendants of X does not match A. Otherwise, X and Y must have a common ancestor Z, where Z matches A because both X and Y match A, and $Z < X$ because Z is the ancestor of X (a node is always smaller than its descendants). This contradicts the assumption that X is smallest matching node of A.

Time complexity: We can break down the time consumed by the mapping phase into two parts. The time to construct the subnet trie is $O(T)$, where T is the number of subnets, because inserting an IP prefix into a trie takes constant time (≤ length of IP address). If we consider a single thread, it takes $O(R)$ time to match R rules against the trie. So the total time complexity is $O(R+T)$. If N mappers share the same trie, we can reduce the time to $O(R + \frac{T}{N})$. Here, we assume $R \gg T$. If $T \gg R$, one may want to construct a trie with rules rather than subnets (as in conventional longest-prefix-matching).

4.4.2 Reducer

The outputs from the mapping phase are shuffled by intermediate keys, which are the subnets. When shuffling finishes, a reducer will receive a subnet, along with an unordered set of values, each containing `rule_mask_len`, `local_switch`, `remote_switches`, and `priority`. The reducer first selects the highest priority rule per `local_switch`: For the same `local_switch`, the rule with higher `priority` is selected; if two rules have the

same priority, the one with larger `mask_len` is chosen. The reducer then constructs a directed forwarding graph using the selected rules. Once the graph is constructed, the reducer uses graph library to verify the properties of the graph, for example, to check if the graph is loop-free.

Time complexity: In most networks we have seen, a subnet matches at most 2-4 rules in the routing table. Hence, selecting the highest priority rule and constructing the graph takes $O(E)$ time, where E is the number of physical links in the network. However, the total runtime depends on the verification task, as we will discuss in Section 4.5.

4.4.3 Incremental Updates

Until now, we have assumed Libra checks the forwarding correctness from scratch each time it runs. Libra also supports incremental updates of subnets and forwarding rules, allowing it to be used as an independent "correctness checking service" similar to NetPlumber [37] and Veriflow [39]. In this way, Libra could be used to check forwarding rules quickly, *before* they are added to the forwarding tables in the switches. Here, a in-memory, "streaming" MapReduce runtime (such as [18]) is needed to speed up the event processing.

Subnet updates. Each time we add a subnet for verification, we need to rerun the whole MapReduce pipeline. The mappers takes $O(\frac{R}{N})$ time to find the relevant rules. And a single reducer takes $O(E)$ time to construct the directed graph slice for the new subnet. If one has several subnets to add, it is faster to run them in a batch, which takes $O(T + \frac{R}{N})$ instead of $O(\frac{RT}{N})$ to map.

Removing subnets is trivial. All results related to the subnets are simply discarded.

Forwarding rule updates. Figure 4.10 shows the workflow to add new forwarding rules. To support incremental updates of rules, reducers need to store the forwarding graph for each slice it is responsible for. The reducer could keep the graph in memory or disk—the trade-off is a larger memory footprint.[6] If the graphs are in disk, a fixed number of idle reducer processes live in the memory and fetch graphs upon request. Similarly, the mappers need to keep the subnet trie.

[6]At any time instance, only a small fraction of graphs will be updated, and so keeping all states in-memory can be quite inefficient.

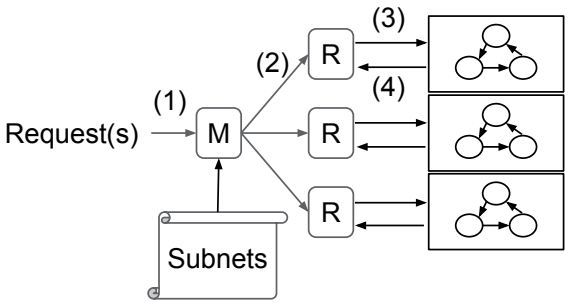


Figure 4.10: Incremental rule updates in Libra. Mappers dispatch matching <subnet, rule> pair to reducers, indexed by subnet. Reducers update the forwarding graph and recompute graph properties.

To add a rule, a mapper is spawned just as it sees another line of input (Step 1). Matching subnets from the trie are shuffled to multiple reducers (Step 2). Each reducer reads the previous slice graph (Step 3), and recalculates it with the new rule (Step 4).

Deleting a rule is similar. The mapper tags the rule as "to be deleted" and pass it to reducers for updating the slice graph. However, in the graph's adjacency list, the reducer not only needs to store the highest priority rule, but also *all* matching rules. This is because if a highest priority rule is deleted, the reducer must use the second highest priority rule to update the graph.

Besides updating graphs, in certain cases, graph properties can also be checked incrementally, since the update only affects a small part of graph. For example, in loop-detection, adding an edge only requires a Depth-First-Search (DFS) starting from the new edge's destination node, which normally will not traverse the entire graph.

Unlike NetPlumber and Veriflow, Libra does not need to explicitly remember the dependency between rules. This is because the dependency is already encoded in the matching and shuffling phases.

4.4.4 Route Dumper

The route dumper records each rule using five fields: `<switch, ip, mask_len, nexthops, priority>`. `switch` is the unique ID of the switch; `ip` and `mask_len` is the prefix. `nexthops` is a list of port names because of multipath. `priority` is an integer field serving as a

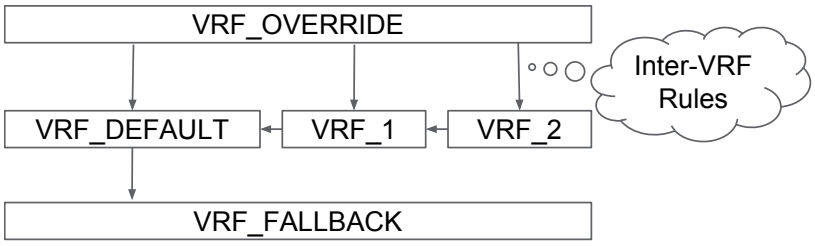


Figure 4.11: Virtual Routing and Forwarding (VRFs) are multiple tables within the same physical switch. The tables have dependency (inter-VRF rules) between them.

tie-breaker in longest prefix matching. By storing the egress ports in `nexthops`, Libra encodes the topology information in the forwarding table.

Although the forwarding table format is mostly straightforward, two cases need special handling:

Ingress port dependent rules. Some forwarding rules depend on particular ingress ports. For example, a rule may only be in effect for packets entering the switch from port `xe1/1`. In reducers we want to construct a simple directed graph that can be represented by an adjacency list. Passing this ingress port dependency to the route checker will complicate the reducer design, since the next hop in the graph depends not only on the current hop, but also the *previous hop*.

We use the notion of *logical switches* to solve this problem. First, if a switch has rules that depend on the ingress port, we split the switch into multiple logical switches. Each logical switch is given a new name and contains the rules depending on one ingress port, so that the port is "owned" by the new logical switch. We copy rules from the original switch to the logical switch. Second, we update the rules in upstream switches to forward to the logical switch.

Multiple tables. Modern switches can have multiple forwarding tables that are chained together by arbitrary matching rules, usually called "Virtual Routing and Forwarding" (VRF). Figure 4.11 depicts an example VRF set up: incoming packets are matched against VRF_OVERRIDE. If no rule is matched, they enter VRF_1 to VRF_16 according to some "triggering" rules. If all matching fails, the packet enters VRF_DEFAULT.

The route dumper maps multiple tables in a physical switch into multiple logical switches, each containing one forwarding table. Each logical switch connects to other

logical switches directly. The rules chaining these VRFs are added as lowest priority rules in the logical switch's table. Hence, if no rule is matched, the packet will continue to the next logical switch in the chain.

4.5 Use cases

In Libra, the directed graph constructed by the reducer contains *all* data plane information for a particular subnet. In this graph, each vertex corresponds to a forwarding table the subnet matched, and each edge represents a possible link the packet can traverse. This graph also encodes multipath information. Therefore, routing correctness directly corresponds to graph properties.

Reachability: A reachability check ensures the subnet can be reached from any switch in the network. This property can be verified by doing a (reverse) DFS from the subnet switch, and checking if the resulting vertex set contains all switches in the network. The verification takes $O(V+E)$ time where V is the number of switches and E the number of links.

Loop detection: A loop in the graph is equivalent to at least one *strongly connected component* in the directed graph. Two vertices `s1` and `s2` belong to a strongly connected component, if there is a path from `s1` to `s2` and a path from `s2` to `s1`. We find strongly connected components using Tarjan's Algorithm [73] whose time complexity is $O(V+E)$.

Black-holes: A switch is a black-hole for a subnet if the switch does not have a matching route entry for the subnet. Some black-holes are legitimate: if the switch is the last hop for the subnet, or there is an explicit drop rule. Implicit drop rules need to be checked if that is by design. Black-holes map to vertices with zero out-degree, which can therefore be enumerated in $O(V)$ time.

Waypoint routing: Network operators may require traffic destined to certain subnets to go through a "waypoint," such as a firewall or a middlebox. Such behavior can be verified in the forwarding graph by checking if the waypoint exists on all the forwarding paths. Specifically, one can remove the waypoint and the associated links, and verify that no edge switches appear any more in a DFS originated from the subnet's first hop switch, with the runtime complexity of $O(V+E)$.

4.6 Implementation

We have implemented Libra for checking the correctness of Software-Defined Network (SDN) clusters. Each cluster is divided into several domains where each domain is controlled by a controller. Controllers exchange routing information and build the routing tables for each switch.

Our Libra prototype has two software components. The route dumper, implemented in Python, connects to each controller and downloads routing events, forwarding tables and VRF configurations in Protocol Buffers [63] format in parallel. It also consults the topology database to identify the peer of each switch link. Once the routing information is downloaded, we preprocess the data as described in Section 4.4.4 and store it in a distributed file system.

The route checker is implemented in C++ as a MapReduce application in about 500 lines of code. We use a Trie library for storing subnets, and use Boost Graph Library [10] for all graph computation. The same binary can run at different levels of parallelism—on a single machine with multiple processes, or on a cluster with multiple machines, simply by changing command line flags.

Although Libra's design supports incremental updates, our current prototype only does batch processing. We use micro-benchmarks to evaluate the specific costs for incremental processing in Section 4.7.5, on a simplified prototype with one mapper and one reducer.

4.7 Evaluation

To evaluate Libra's performance, we first measure start-to-finish runtime on a single machine with multithreading, as well as on multiple machines in a cluster. We also demonstrate Libra's linear scalability as well as its incremental update capability.

4.7.1 Data Sets

We use three data sets to evaluate the performance of Libra. The detailed statistics are shown in Table 4.2.

Data set	Switches	Rules	Subnets
DCN	11,260	2,657,422	11,136
DCN-G	1,126,001	265,742,626	1,113,600
INET	316	151,649,486	482,966

Table 4.2: Data sets used for evaluating Libra.

DCN: DCN is an SDN testbed used to evaluate the scalability of the controller software. Switches are emulated by OpenFlow agents running on commodity machines and connected together through a virtual network fabric. The network is partitioned among controllers, which exchange routing information to compute the forwarding state for switches in their partition. DCN contains about 10 thousand switches and 2.6 million IPv4 forwarding rules. VRF is used throughout the network.

DCN-G: To stress test Libra, we replicate DCN 100 times by shifting the address space in DCN such that each DCN-part has a unique IP address space. A single top-level switch interconnects all the DCN pieces together. DCN-G has 1 million switches and 265 million forwarding rules.

INET: INET is a synthetic wide area backbone network. First, we use the Sprint network topology discovered by RocketFuel project [71], which contains roughly 300 routers. Then, we create an interface for each prefix found in a full BGP table from Route Views [65] (~500k entries as of July 2013), and spread them randomly and uniformly to each router as "local prefixes." Finally, we compute forwarding tables using shortest path routing.

4.7.2 Single Machine Performance

We start our evaluation of Libra by running loop detection locally on a desktop with Intel 6-core CPU and 32GB memory. Table 4.3 summarizes the results. We have learned several aspects of Libra from single machine evaluation:

I/O bottlenecks: Standard MapReduce is disk-based: Inputs are piped into the system from disks, which can create I/O bottlenecks. For example, on INET, reading from disk take 15 times longer than graph computation. On DCN, the I/O time is much shorter due to the smaller number of forwarding rules. In fact, in both cases, the I/O

Threads	1	2	4	6	8
Read/s	13.7				
Shuffle/s	7.4				
Reduce/s	46.3	25.8	15.6	12.1	11.1
Speedup	1.00	1.79	2.96	3.82	4.17

a) DCN with 2000 subnets

Threads	1	2	4	6	8
Read/s	170				
Shuffle/s	3.8				
Reduce/s	11.3	5.9	3.2	2.7	2.1
Speedup	1.00	1.91	3.53	4.18	5.38

b) INET with 10,000 subnets

Table 4.3: Runtime of loop detection on DCN and INET data sets on single machine. The number of subnets is reduced compared to Table 4.2 so that all intermediate states can fit in the memory. Read and shuffle phases are single-threaded due to the framework limitation.

is the bottleneck and the CPU is not fully utilized. The runtime remains the same with or without mapping. Hence, the mapping phase is omitted in Table 4.3.

Memory consumption: In standard MapReduce, intermediate results are flushed to disk after the mapping phase before shuffling, which is very slow on a single machine. We avoid this by keeping all intermediate states in-memory. However, it limits the number of subnets that can be verified at a time—intermediate results are all matching (subnet, rule) pairs. On a single machine, we have to limit the number of subnets to 2000 in DCN and 10,000 in INET to avoid running out of memory.

Graph size dominates reducing phase: The reducing phase on DCN is significantly slower than on INET, despite INET having 75 times more forwarding rules. With a single thread, Libra can only process 43.2 subnets per second on DCN, compared with 885.0 subnets per second on INET (20.5 times faster). Note that DCN has 35.6 times more nodes. This explains the faster running time on INET, since the time to detect loops grows linearly with the number of edges and nodes in the graph.

Multi-thread: Libra is multi-threaded, but the multi-thread speedup is not linear. For example, on DCN, using 8 threads only resulted in a 4.17 speedup. This effect is

	DCN	DCN-G	INET
Machines	50	20,000	50
Map Input/Byte	844M	52.41G	12.04G
Shuffle Input/Byte	1.61G	16.95T	5.72G
Reduce Input/Byte	15.65G	132T	15.71G
Map Time/s	31	258	76.8
Shuffle Time/s	32	768	76.2
Reduce Time/s	25	672	16
Total Time/s	**57**	**906**	**93**

Table 4.4: Running time summary of the three data sets. Shuffle input is compressed, while map and reduce inputs are uncompressed. DCN-G results are extrapolated from processing 1% of subnets with 200 machines as a single job.

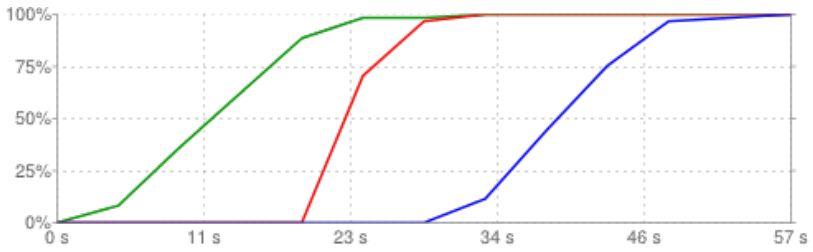


Figure 4.12: Example progress percentage (in Bytes) of Libra on DCN. The three curves represent (from left to right) Mapping, Shuffling, and Reducing phases, which are partially overlapping. The whole process ends in 57 seconds.

likely due to inefficiencies in the threading implementation in the underlying MapReduce framework, although theoretically, all reducer threads should run in parallel without state sharing.

4.7.3 Cluster

We use Libra to check for loops against our three data sets on a computing cluster. Table 4.4 summarizes the results. Libra spends 57 seconds on DCN, 15 minutes on DCN-G, and 93 seconds on INET. To avoid overloading the cluster, the DCN-G result is extrapolated from the runtime of 1% of DCN-G subnets with 200 machines. We assume 100 such jobs running in parallel—each job processes 1% of subnets against all rules. All the jobs are *independent* of each other.

We make the following observations from our cluster-based evaluation.

Runtime in different phases: In all data sets, the sum of the runtime in the phases is larger than the start-to-end runtime. This is because the phases can overlap each other. There is no dependency between different mapping shards. A shard that finishes the mapping phase can enter the shuffling phase without waiting for other shards. However, there is a global barrier between mapping and reducing phases since MapReduce requires a reducer to receive all intermediate values before starting. Hence, the sum of runtime of mapping and reducing phases roughly equals the total runtime. Table 4.4 shows the overlapping progress (in bytes) of all three phases in an analysis of DCN.

Shared-cluster overhead: These numbers are a lower bound of what Libra can achieve for two reasons: First, the cluster is shared with other processes and lacks performance isolation. In all experiments, Libra uses 8 threads. However, the CPU utilization is between 100% and 350% on 12-core machines, whereas it can achieve 800% on a dedicated machine. Second, the machines start processing at different times—each machine may need different times for initialization. Hence, all the machines are not running at full-speed from the start.

Parallelism amortizes I/O overhead: Through detailed counters, we found that unlike in the single machine case (where I/O is the bottleneck), the mapping and reducing time dominates the total runtime. We have seen 75%–80% of time spent on mapping/reducing. This is because the aggregated I/O bandwidth of all machines in a cluster is much higher than a single machine. The I/O is faster than the computation throughput, which means threads will not starve.

4.7.4 Linear scalability

Figure 4.13 shows how Libra scales with the size of the network. We change the number of devices in the DCN network, effectively changing both the size of the forwarding graph and the number of rules. We do not change the number of subnets. Our experiments run the loop detection on 50 machines, as in the previous section. The figure shows that the Libra runtime scales linearly with the number of rules. The reduce phase grows more erratically than the mapping time, because it is affected by both nodes and

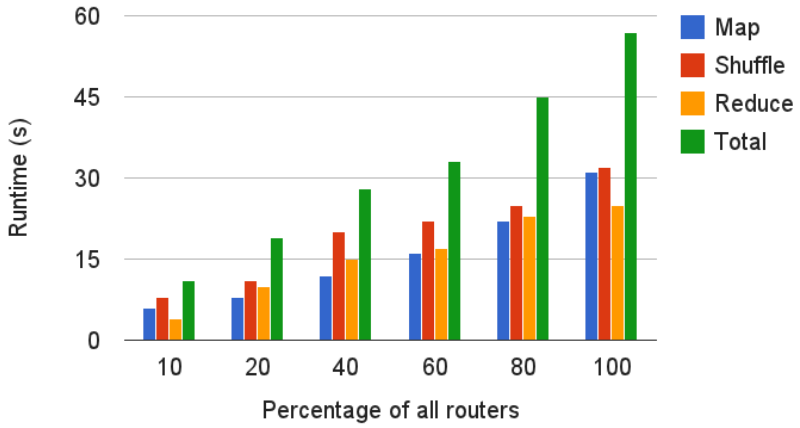


Figure 4.13: Libra runtime increases linearly with network size.

edges in the network, while mapping only depends on the number of rules.

Libra's runtime is not necessarily inversely proportional to the number of machines used. The linear scalability only applies when mapping and reducing time dominate. In fact, we observe that more machines can take *longer* to finish a job, because the overhead of the MapReduce system can slow down Libra. For example, if we have more mapping/reducing shards, we need to spend an additional overhead on disk shuffling. We omit the detailed discussion as it depends on the specifics of the underlying MapReduce framework.

4.7.5 Incremental Updates

Libra can update forwarding graphs incrementally as we add and delete rules in the network, as shown in Section 4.4.3. To understand its performance, we can breakdown Libra's incremental computation into two steps: (1) time spent in prefix matching (map phase) to find which subnets are affected, and (2) time to do an incremental DFS starting from the node whose routing entries have changed (reduce phase). We also report the total heap memory allocated.

We measured the time for each of the components as follows: (1) for prefix matching, we randomly select rules and find out all matching subnets using the algorithm

	Map (μs)	Reduce (ms)	Memory (MB)
DCN	0.133	0.62	12
DCN-G	0.156	1.76	412
INET	0.158	<0.01	7

Table 4.5: Breakdown of runtime for incremental loop checks. The unit for map phase is microsecond and the unit for reduce phase is millisecond.

described in Section 4.4.1, and (2) for incremental DFS, we started a new DFS from randomly chosen nodes in the graph. Both results are averaged across 1000 tests. The results are shown in Table 4.5.

First, we verified that no matter how large the subnet trie is, prefix matching takes almost constant time: DCN-G's subnet trie is 100 times larger than DCN-G's but takes almost the same time. Second, the results also show that the runtime for incremental DFS is likely to be dominated by I/O rather than compute, because the size of the forwarding graph does not exceed the size of the physical network. Even the largest dataset, DCN-G, has only about a million nodes and 10 million edges, which fits into 412MBytes of memory. This millisecond runtime is comparable to results reported in [37] and [39], but now on much bigger networks.

4.8 Limitations of Libra

Libra is designed for static headers: Libra is faster and more scalable than existing tools because it solves a narrower problem; it assumes packets are only forwarded based on IP prefixes, and that headers are not modified along the way. Unlike, say HSA, Libra cannot process a graph that forwards on an arbitrary mix of headers, since it is not obvious how to carry matching information from mappers to reducers, or how to partition the problem.

As with other static checkers, Libra cannot handle non-deterministic network behavior or dynamic forwarding state (e.g., NAT). It requires a complete, static snapshot of forwarding state to verify correctness. Moreover, Libra cannot tell *why* a forwarding state is incorrect or how it will evolve as it does not interpret control logic.

Libra is designed to slice the network by IP subnet: If headers are transformed in a

deterministic way (e.g., static NAT and IP tunnels), Libra can be extended by combining results from multiple forwarding graphs at the end. For example, 192.168.0/24 in the Intra-DC network may be translated to 10.0.0/24 in the Inter-DC network. Libra can construct forwarding graphs for both 192.168.0/24 and 10.0.0/24. When analyzing the two subgraphs we can add an edge to connect them.

Forwarding graph too big for a single server: Libra scales linearly with both subnets and rules. However, a single reducer still computes the entire forwarding graph, which might still be too large for a single server. Since the reduce speed depends on the size of the graph, we could use distributed graph libraries [60] in the reduce phase to accelerate Libra.

Subnets must be contained by a forwarding rule: In order to break the network into one forwarding graph per subnet, Libra examines all the forwarding rules to decide which rules *contain* the subnet. This is a practical assumption because, in most networks, the rule is a prefix that *aggregates* many subnets. However, if the rule has a longer, more specific prefix (e.g., it is for routing to a specific end-host or router console) than the subnet's, the forwarding graph would be complicated since the rule, represented as an edge in the graph, does not apply to all addresses of the subnet. In this case, one can use Veriflow [39]'s notion of equivalence classes to acquire subnets directly from the rules themselves. This technique may serve as an alternative way to find all matching (subnet, rule) pairs. We leave this for future work.

4.9 Related Work

Static data plane checkers: Xie et. al introduced algorithms to analyze reachability in IP networks [76]. Anteater [49] makes them practical by converting the checking problem into a Boolean satisfiability problem and solving it with SAT solvers. Header space analysis [38] tackles general protocol-independent static checking using a geometric model and functional simulation. Recently, NetPlumber [37] and Veriflow [39] show that, for small networks (compared to the ones we consider here) static checking can be done in milliseconds by tracking the dependency between rules. Specifically, Veriflow slices the network into *equivalence classes* and builds a *forwarding graph* for each class, in a

similar fashion to Libra.

However, with the exception of NetPlumber, all of these tools and algorithms assume centralized computing. NetPlumber introduces a "rule clustering" technique for scalabilty, observing that rule dependencies can be separated into several relatively *independent* clusters. Each cluster is assigned to a process so that rule updates can be handled individually. However, the benefits of parallelism diminish when the number of workers exceeds the number of natural clusters in the ruleset. In contrast, Libra scales linearly with both rules and subnets. Specifically, even two rules have dependency, Libra can still place them into different map shards, and allow reducers to resolve the conflicts.

Other network troubleshooting techniques: Existing network troubleshooting tools focus on a variety of network components. Specifically, the explicitly layered design of SDN facilitates systematic troubleshooting [31]. Efforts in formal language foundations [27] and model-checking control programs [13] reduce the probability of buggy control planes. This effort has been recently extended to the embedded software on switches [42]. However, based on our experience, multiple simultaneous writers in a dynamic environment make developing a bug-free control plane extremely difficult.

Active testing tools [78] reveal the inconsistency between the forwarding table and the actual forwarding state by sending out specially designed probes. They can discover runtime properties such as congestion, packet loss, or faulty hardware, which cannot be detected by static checking tools. Libra is orthogonal to these tools since we focus on forwarding table *correctness*.

Researchers have proposed systems to extract abnormalities from event histories. STS [67] extracts "minimal causal sequences" from control plane event history to explain a particular crash or other abnormalities. NDB [29] compiles packet histories and reasons about data plane correctness. These methods avoid taking a stable snapshot from the network.

4.10 Summary

Today's networks require way too much human intervention to keep them working. As networks get larger and larger there is huge interest in automating the control, error-reporting, troubleshooting and debugging. Until now, there has been no way to automatically verify all the forwarding behavior in a network with tens of thousands of switches. Libra is fast because it focuses on checking the IP-only fabric commonly used in data centers. Libra is scalable because it can be implemented using MapReduce allowing it to harness large numbers of servers. In our experiments, Libra can meet the benchmark goal we set out to achieve: it can verify the correctness of a 10,000-node network in 1 minute using 50 servers. In future, we expect tools like Libra to check the correctness of even larger networks in real-time.

Modern large networks have gone far beyond what human operators can debug with their wisdom and intuition. Our experience shows that it also goes beyond what single machine can comfortably handle. We hope that Libra is just the beginning of bringing distributed computing into the network verification world.

Chapter 5

Conclusion

The future is already here – it's just not very evenly distributed.

William Gibson (1948 -)

As stated in Chapter 1, an automatic data plane tester can be classified with three metrics: (1) **Method**: either static analysis based on the stable snapshot of the data plane state or dynamic analysis by sending out test packets to exercise rules, interfaces, and links; (2) **Knowledge**: black box testing without any knowledge on the data plane, white box testing assuming the data plane can be expressed unambiguously, or gray box testing with partial knowledge; (3) **Coverage**: deterministic cover versus probabilistic cover. The actual design point fundamentally depends on the scale of the network (enterprise, backbone, or data center network), the types of errors to catch (compile-time or run-time), and management constraints (whether we can deploy test agents). With this taxonomy, the three data plane testers described previous chapters fit naturally into different categories:

1. **ATPG** is a dynamic, white box tester with rule and link cover. In the offline stage, ATPG reads forwarding tables from all routers, constructs a device-independent

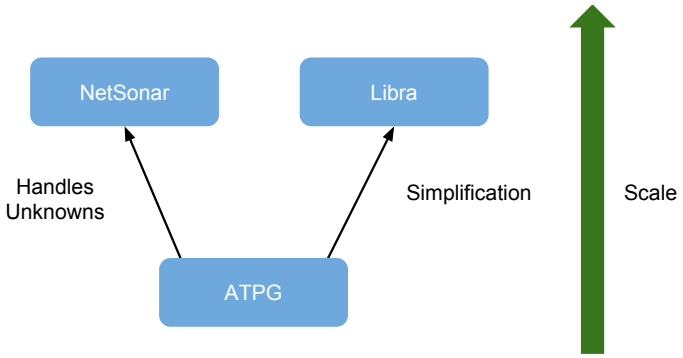


Figure 5.1: Relationship among ATPG, NetSonar, and Libra.

model based on Header Space Analysis, and generates a *minimal* test suite of packets with different headers to exercise all forwarding rules and links. Subsequently, the online testing stage sends out these packets and analyzes the results.

2. **NetSonar** is a dynamic, gray box tester with probabilistic path cover and diagnosable link cover. It is designed for large inter-DC networks with frequent changes and extensive multipath routing. NetSonar only relies on coarse, less-frequently changed information such as network topology and LSPs. The fine-grained information is collected in parallel to the testing itself.

3. **Libra** is a static, white box tester with the rule cover designed for data center networks. Compared to other white box testers, Libra solves two challenges, both related to the scale of data centers: how to take a stable snapshot of the data plane state across numerous switches, and how to complete computation within a reasonable time frame in a distributed fashion.

In the following sections, I will situate these three systems into a broader perspective, summarize the contributions made in this dissertation, and present my views of how data plane testers can move forward.

5.1 The Big Picture

Figure 5.1 shows the relationship among ATPG, NetSonar, and Libra, the three systems described in previous chapters. ATPG sets the foundation of data plane testing by generating an exhaustive all-pairs reachability table against the data plane model. Its Min-Set-Cover algorithm minimizes the test suite while maintaining the test coverage. ATPG works well in enterprise-size networks such as Stanford backbone. However, the challenges of scaling ATPG to larger networks prove to be non-trivial.

NetSonar handles the challenge of scaling ATPG to an inter-DC backbone: the *unknowns* in the data plane. These unknowns come from the imperfect engineering in today's network devices (e.g., unknown hash functions), as well as intrinsic size of networks (e.g., frequent changes in data plane). NetSonar's gray-box testing approaches, including probabilistic path cover, diagnosable link cover, and traceable probes help eliminate as many unknowns as practical.

Libra takes an alternative approach to scaling ATPG through *simplification*. First, Libra is a static rather than a dynamic tester, concerned only about the logical correctness of forwarding tables rather than network performance. This is because in data center networks, the protocol convergence is slow and less predictable due to the size of the network, and thus checking logical correctness to avoid having the network caught in a bad state is crucial. Second, instead of general <match, action> rules, Libra assumes homogeneous, pure IP forwarding throughout the network, and can consequently employ the abstraction of forwarding graphs. This simplification is also reasonable since data center networks do not have the complex forwarding schemes seen in enterprise and backbone networks. The benefit of this simplification is that testing can achieve a very large scale.

5.2 Contributions

This dissertation made the following contributions:

1. Designed ATPG, an automatic tester that generates a minimal test suite against the data plane with the Min-Set-Cover algorithm, which achieves 100% coverage

of rules and links.

2. Designed a fault localization algorithm in ATPG for synthesizing the results collected by test agents, pinpointing the rule or link that causes failures.

3. Implemented ATPG as an open-source tool, with Cisco and Juniper device parsers.

4. Deployed ATPG to three Stanford buildings and captured real-world failures.

5. Designed NetSonar, a tester for large networks that handles dynamic changes and unknown multipath hash functions by using only stable information and probing fast-changing states *while* testing.

6. Designed an offline cover algorithm for NetSonar, with probabilistic path cover and diagnosable link cover.

7. Implemented NetSonar as part of Autopilot data center management framework.

8. Deployed and evaluated NetSonar on a large production inter-DC network, compressing the conventional SNMP alerts by 80x.

9. Designed Libra, a distributed static tester for data center networks, checking correctness of forwarding tables with a divide-and-conquer algorithm.

10. Designed a stable snapshot algorithm for Libra to capture the data plane state across numerous switches.

11. Implemented Libra on MapReduce, a distributed computing framework.

12. Evaluated Libra on several large datasets, proving scalability by checking loops in a 10,000-node network in one minute.

5.3 Future Directions

Graybox Static Tester If we place the three systems described in this dissertation in a method-knowledge matrix, as shown in Table 5.1, there is an obvious gap in the graybox-static slot. So far, almost all static network testers assume full knowledge of the data

	White-box	Gray-box
Static	Libra	?
Dynamic	ATPG	NetSonar

Table 5.1: Graybox static tester is an obvious gap among ATPG, NetSonar, and Libra.

plane. HSA [38] cannot model boxes if their internal states can be changed by test packets, for example, an NAT that dynamically assigns TCP ports to outgoing packets confuses the online monitor, because the same test packet can give different results.

However, certain types of unknowns offer an easy extension that provides at least useful and possibly even solid conclusions. The forwarding graph abstraction in Libra is such an example. In Figure 4.3 (a), although we cannot determine whether a packet will flow from S12 to S21 or S22, the forwarding graph simply iterates *all possible* output links. If the forwarding graph is loop-free, we can conclude that the network is loop-free, no matter which path a packet takes in the run-time.

Complexity Traditionally, the network data plane is kept simple - classic IP forwarding is purely destination-based and has no header manipulation. Our work on Libra shows that such a simple model provides the fast testing that can easily implemented in a distributed fashion. However, more and more complexity is added into the data plane to achieve particular goals: MPLS is introduced to trunk traffic in the backbone, VLAN to provide logical isolation; and recently NVGRE [57] and VXLAN [75] to facilitate network visualization in data centers.

All these movements in the data plane pose challenges for automatic data plane testing. The testers need a general data plane model such as Header Space Analysis [38] that can handle various header manipulations. They need to properly define coverage, i.e., what and how data plane operations are tested. Finally and most importantly, all the testing, whether static or dynamic, should finish in a reasonable time frame, especially when the reachability problem with header manipulation is NP-Complete [49].

Partition States All data plane testers are built upon the understanding of the data plane state. The change of state, which happens more frequently in large networks, must be captured by testers to avoid producing false positives. NetSonar and Libra tackle this problem in different ways: NetSonar probes the state *in parallel to* testing, while

Libra utilizes a stable snapshot algorithm that wait until the network is "silent" before checking.

The difficulty of knowing the data plane state is largely determined by the size of the network. A small network will have fewer changes, making a snapshot of the data plane state easy to achieve. This suggests that we can partition the network and classify changes into "intra-partition" changes and "inter-partition" changes, where each partition can be verified separately. The challenge here is partitioning the network properly under different environments and carefully defining the interactions between partitions. This approach is different from the divide-and-conquer algorithm in Libra, which assumes a stable snapshot across the *entire* network.

Device Design In all testers described in this dissertation, the networking device is assumed to be passive or only minimally cooperative (e.g., responds `traceroute` with ICMP packets). How can we improve the devices themselves to facilitate automatic data plane testing? Two possibilities come to my mind: First, the device should be logically transparent. A device model without ambiguity will minimize the guess work done by gray box testers and thus increase the accuracy. For example, with the knowledge of hash function, NetSonar can reduce the number of probes needed to cover all the paths.

Second, packets should be allowed the option of carrying debug information in the data plane. At present, NetSonar has to send `traceroute` along with `ping` simply because ping itself cannot carry path information. Most of the debugging information is processed in the control plane, which is less realistic in the data plane testing context, especially in performance testing. A hardware-supported information tagging function, similar to IPv4's Record Route (RR), would ease the troubleshooting significantly. The information could include router and interface name, matching rule index, etc.

Reconfigurable Data Plane Conventionally, the data plane is thought to be static - once the ASIC is produced, its functionality is fixed. One can update the entry in the table, but cannot change the order of parsing, the supported packet headers, or the size and width of tables. Recently, researchers have shown that it is possible to design a switching ASIC that is reconfigurable. [11] Given a parse graph and a table dependency graph, a "compiler" can map the design goal (e.g., the support of a new wire protocol) into the hardware resources available on the ASIC.

A reconfigurable data plane creates another opportunity for data plane testers. On a single device scope, the tester can check the consistency between the design goal and the final hardware placement, essentially checking the correctness of the compiler. On a wider network scope, each device's design goal is usually part of a higher level functionality goal, which again can be tested either statically with device configuration or dynamically with test packets.

5.4 Closing Remarks

The last few decades have witnessed rapid improvement in network performance: Ethernet speed has increased from 10Mbps (IEEE 802.3i, 1990) to 100Gbps (IEEE 802.3ba, 2010) in 20 years. Networks have also grown in scale: The Internet used to be filled with webpages served on a single server. Nowadays, online service providers maintain dozens of data centers, each with tens of thousands of switches, and these numbers will only continue to grow in the future. By contrast, the way networks are tested and verified today has not changed much: `ping` was first authored in 1983, `traceroute` in 1987, and SNMP in 1988. As a result, network engineers have to devote large amounts of resources on manual debugging in order to keep networks working.

In any complex system, accidents and failures are the norm rather than the exception. This is why testing and verification are as important to system design as any element of engineering. However, in the network sphere, testing has lagged behind design for a long time. The tools and methods described in this dissertation, along with other recent efforts from the network research community, demonstrate the power of rigorous, systematic, and automatic network testing. They reduce the chance of outages, save the precious human resources for more important tasks, and ultimately lower the cost of network operation. I am confident that the era of automatic testing and verification of networks will arrive in the near future.

Bibliography

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

[2] Brice Augustin, Xavier Cuvellier, Benjamin Orgogozo, Fabien Viger, Timur Friedman, Matthieu Latapy, Clémence Magnien, and Renata Teixeira. Avoiding traceroute anomalies with paris traceroute. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 153–158, New York, NY, USA, 2006. ACM.

[3] Automatic Test Pattern Generation. `http://en.wikipedia.org/wiki/Automatic_test_pattern_generation`.

[4] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 13–24, New York, NY, USA, 2007. ACM.

[5] P. Barford, N. Duffield, A. Ron, and J. Sommers. Network performance anomaly detection and localization. In *INFOCOM 2009, IEEE*, pages 1377–1385, 2009.

[6] P. Barford, N. Duffield, A. Ron, and J. Sommers. Network performance anomaly detection and localization. In *INFOCOM 2009, IEEE*, pages 1377–1385, April.

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

[8] Y. Bejerano and R. Rastogi. Robust monitoring of link delays and faults in ip networks. *Networking, IEEE/ACM Transactions on*, 14(5):1092–1103, 2006.

[9] Yigal Bejerano and Rajeev Rastogi. Robust monitoring of link delays and faults in ip networks. *IEEE/ACM Trans. Netw.*, 14(5):1092–1103, October 2006.

[10] Boost Graph Library. `http://www.boost.org/libs/graph`.

[11] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI'08*, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[13] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, and Jennifer Rexford. A NICE way to test openflow applications. *Proceedings of NSDI'12*, 2012.

[14] Rui Castro, Mark Coates, Gang Liang, Robert Nowak, and Bin Yu. Network tomography: recent developments. *Statistical Science*, 19:499–517, 2004.

[15] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[16] Yan Chen, David Bindel, Hanhee Song, and Randy H. Katz. An algebraic approach to practical and scalable overlay network monitoring. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 55–66, New York, NY, USA, 2004. ACM.

[17] David Chua, Eric D. Kolaczyk, and Mark Crovella. A statistical framework for efficient monitoring of end-to-end network properties. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 390–391, New York, NY, USA, 2005. ACM.

[18] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. *NSDI*, 2010.

[19] Ítalo Cunha, Renata Teixeira, Nick Feamster, and Christophe Diot. Measurement methods for fast and accurate blackhole identification with binary tomography. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 254–266, New York, NY, USA, 2009. ACM.

[20] Italo Cunha, Renata Teixeira, Darryl Veitch, and Christophe Diot. Predicting and tracking internet path changes. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 122–133, New York, NY, USA, 2011. ACM.

[21] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[22] Amogh Dhamdhere, Renata Teixeira, Constantine Dovrolis, and Christophe Diot. Netdiagnoser: troubleshooting network unreachabilities using end-to-end probes and routing data. In *Proceedings of the 2007 ACM CoNEXT conference*, pages 18:1–18:12, New York, NY, USA, 2007. ACM.

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

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

[25] N.G. Duffield, F. Lo Presti, V. Paxson, and D. Towsley. Inferring link loss using striped unicast probes. In *Proceedings IEEE INFOCOM 2001*, volume 2, pages 915 –923 vol.2, 2001.

[26] William Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, Inc., 1968.

[27] N. Foster, A. Guha, M. Reitblatt, A. Story, M.J. Freedman, N.P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for software-defined networks. *Communications Magazine, IEEE*, 51(2):128–134, 2013.

[28] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 350–361, New York, NY, USA, 2011. ACM.

[29] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. Where is the debugger for my software-defined network? In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 55–60, New York, NY, USA, 2012. ACM.

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

[31] Brandon Heller, Colin Scott, Nick McKeown, Scott Shenker, Andreas Wundsam, Hongyi Zeng, Sam Whitlock, Vimalkumar Jeyakumar, Nikhil Handigol, James McCauley, Kyriakos Zarifis, and Peyman Kazemian. Leveraging sdn layering to systematically troubleshoot networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, HotSDN '13, pages 37–42, New York, NY, USA, 2013. ACM.

[32] Yiyi Huang, Nick Feamster, and Renata Teixeira. Practical issues with using network tomography for fault diagnosis. *SIGCOMM Comput. Commun. Rev.*, 38(5):53–58, September 2008.

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

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

[35] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 243–254, New York, NY, USA, 2009. ACM.

[36] Ethan Katz-Bassett, Harsha V. Madhyastha, John P. John, Arvind Krishnamurthy, David Wetherall, and Thomas Anderson. Studying black holes in the internet with hubble. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 247–262, Berkeley, CA, USA, 2008. USENIX Association.

[37] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 99–112, Berkeley, CA, USA, 2013. USENIX Association.

[38] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: static checking for networks. *Proceedings of NSDI'12*, 2012.

[39] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 15–28, Berkeley, CA, USA, 2013. USENIX Association.

[40] Ramana Rao Kompella, Jennifer Yates, Albert Greenberg, and Alex C. Snoeren. Ip fault localization via risk modeling. In *Proceedings of NSDI'05 - Volume 2*, pages 57–70, Berkeley, CA, USA, 2005. USENIX Association.

[41] R. Kumar and J. Kaur. Practical beacon placement for link monitoring using network tomography. *Selected Areas in Communications, IEEE Journal on*, 24(12):2196–2209, 2006.

[42] Maciej Kuzniar, Peter Peresini, Marco Canini, Daniele Venzano, and Dejan Kostic. A SOFT way for openflow switch interoperability testing. In *Proceedings of the 2012 ACM CoNEXT Conference*, 2012.

[43] Kevin Lai and Mary Baker. Nettimer: a tool for measuring bottleneck link, bandwidth. In *Proceedings of USITS'01 - Volume 3*, pages 11–11, Berkeley, CA, USA, 2001. USENIX Association.

[44] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of Hotnets '10*, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[45] Franck Le, Sihyung Lee, Tina Wong, Hyong S. Kim, and Darrell Newcomb. Detecting network-wide and router-specific misconfigurations through data mining. *IEEE/ACM Trans. Netw.*, 17(1):66–79, February 2009.

[46] Harsha V. Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iplane: an information plane for distributed services. In *Proceedings of OSDI'06*, pages 367–380, Berkeley, CA, USA, 2006. USENIX Association.

[47] Ajay Mahimkar, Zihui Ge, Jia Wang, Jennifer Yates, Yin Zhang, Joanne Emmons, Brian Huntley, and Mark Stockert. Rapid detection of maintenance induced changes in service performance. In *Proceedings of the 2011 ACM CoNEXT Conference*, pages 13:1–13:12, New York, NY, USA, 2011. ACM.

[48] Ajay Mahimkar, Jennifer Yates, Yin Zhang, Aman Shaikh, Jia Wang, Zihui Ge, and Cheng Tien Ee. Troubleshooting chronic conditions in large ip networks. In *Proceedings of the 2008 ACM CoNEXT Conference*, pages 2:1–2:12, New York, NY, USA, 2008. ACM.

[49] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. *SIGCOMM Comput. Commun. Rev.*, 41(4):290–301, August 2011.

[50] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, Yashar Ganjali, and Christophe Diot. Characterization of failures in an operational ip backbone network. *IEEE/ACM Trans. Netw.*, 16(4):749–762, August 2008.

[51] Keith Marzullo and Gil Neiger. *Detection of global state predicates*. Springer, 1992.

[52] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, March 2008.

[53] David L Mills. Internet time synchronization: the network time protocol. *Communications, IEEE Transactions on*, 39(10):1482–1493, 1991.

[54] Hung X. Nguyen and Patrick Thiran. Active measurement for multiple link failures diagnosis in ip networks. In *in Proceedings of Passive and Active Measurement Workshop (PAM*, pages 185–194, 2004.

[55] H.X. Nguyen, R. Teixeira, P. Thiran, and C. Diot. Minimizing probing cost for detecting interface failures: Algorithms and scalability analysis. In *INFOCOM 2009, IEEE*, pages 1386–1394, 2009.

[56] H.X. Nguyen and P. Thiran. The boolean solution to the congested ip link location problem: Theory and practice. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 2117–2125, 2007.

[57] NVGRE: Network Virtualization using Generic Routing Encapsulation. `http://tools.ietf.org/html/draft-sridharan-virtualization-nvgre-03`.

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

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

[60] The Parallel Boost Graph Library. `http://osl.iu.edu/research/pbgl/`.

[61] Cristel Pelsser, Luca Cittadini, Stefano Vissicchio, and Randy Bush. From paris to tokyo: On the suitability of ping to measure latency. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 427–432, New York, NY, USA, 2013. ACM.

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

[63] Protocol Buffers. `https://code.google.com/p/protobuf/`.

[64] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference*. ACM, 2012.

[65] Route Views. `http://www.routeviews.org/`.

[66] Herman Von Schelling. Coupon collecting for uneqal probabilities. *The American Mathematical Monthly*, 61(5):pp. 306–311, 1954.

[67] Colin Scott, Andreas Wundsam, Sam Whitlock, Andrew Or, Eugene Huang, Kyriakos Zarifis, and Scott Shenker. How Did We Get Into This Mess? Isolating Fault-Inducing Inputs to SDN Control Software. Technical Report UCB/EECS-2013-8, EECS Department, University of California, Berkeley, 2013.

[68] S. Shalunov, B. Teitelbaum, A. Karp, J. Boote, and M. Zekauskas. A One-way Active Measurement Protocol (OWAMP). RFC 4656 (Proposed Standard), September 2006.

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

[70] Han Hee Song, Lili Qiu, and Yin Zhang. Netquest: a flexible framework for large-scale network measurement. *IEEE/ACM Trans. Netw.*, 17(1):106–119, February 2009.

[71] Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, February 2004.

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

[73] Robert Tarjan. Depth-first search and linear graph algorithms. *12th Annual Symposium on Switching and Automata Theory*, 1971.

[74] Daniel Turner, Kirill Levchenko, Alex C. Snoeren, and Stefan Savage. California fault lines: understanding the causes and impact of network failures. *SIGCOMM Comput. Commun. Rev.*, 41(4):–, August 2010.

[75] VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. `http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-04`.

[76] G.G. Xie, Jibin Zhan, D.A. Maltz, Hui Zhang, Albert Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 2170–2183 vol. 3, 2005.

[77] Praveen Yalagandula, Puneet Sharma, Sujata Banerjee, Sujoy Basu, and Sung-Ju Lee. S3: a scalable sensing service for monitoring large networked systems. In *Proceedings of INM '06*, pages 71–76, New York, NY, USA, 2006. ACM.

[78] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 241–252, New York, NY, USA, 2012. ACM.

[79] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Presented as part of the 11th USENIX Symposium on Networked Systems Design and Implementation*, Berkeley, CA, 2014. USENIX.

[80] Ming Zhang, Chi Zhang, Vivek Pai, Larry Peterson, and Randy Wang. Planetseer: internet path failure monitoring and characterization in wide-area services. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 12–12, Berkeley, CA, USA, 2004. USENIX Association.

[81] Yao Zhao, Yan Chen, and David Bindel. Towards unbiased end-to-end network diagnosis. *IEEE/ACM Trans. Netw.*, 17(6):1724–1737, December 2009.