

HEADER SPACE ANALYSIS

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

Peyman Kazemian

June 2013

© 2013 by Peyman Kazemian. All Rights Reserved.
Re-distributed by Stanford University under license with the author.



This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.

<http://creativecommons.org/licenses/by-nc/3.0/us/>

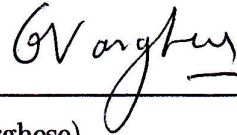
This dissertation is online at: <http://purl.stanford.edu/sc635rt6094>

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) Principal 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.



(George Varghese)

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.



(Balaji Prabhakar)

Approved for the University Committee on Graduate Studies

Abstract

In many engineering disciplines, such as digital design or software engineering, there is an abundance of theoretical foundations and practical tools for verification and debugging. In sharp contrast, the field of networking mostly relies on rudimentary tools such as `ping` and `traceroute`, together with the accrued wisdom and intuition of network administrators, for verification and debugging of networks. Debugging networks is becoming harder as networks are getting *bigger* and *more complicated*.

The verification and debugging of networks is difficult because (1) the forwarding state—the set of rules that determines how an incoming packet is processed and forwarded by network boxes—is distributed across multiple boxes, expressed in vendor dependent command line interface (CLI) formats, and is defined by the forwarding tables, filter rules, and other configuration parameters. As a result it is hard to observe and analyze the forwarding state and understand the overall system behavior. (2) The forwarding state is written by multiple independent programs, protocols, and humans. This may result in complex and unpredictable interactions between these independently generated forwarding states.

Therefore, the first step in making tools for network verification and debugging is to create a simple model for the forwarding functionality of the network that abstracts away the complexities of understanding the forwarding state. One observation is that packet headers, despite carrying multiple protocols, are just sequences of bits, and networking boxes, despite all their complexities, simply rewrite and forward packet headers. Therefore, in our analytical framework, called *the Header Space Analysis (HSA)*, a packet header is viewed as a flat sequence of bits and is modeled as a point in a $\{0, 1\}^L$ space, called *the Header Space*, where L is the length of the header. Each

dimension in the header space corresponds to one bit in the packet header. Also, networking boxes are modeled as *Transfer Functions*, transforming packets from one point in the header space to another point or set of points.

This easy-to-use formalism abstracts away the complexity of the protocols and vendor-specific semantics of network boxes and gives us a model to analytically prove properties about networks that are otherwise hard to ensure. For example, in a typical network, it is hard to find the reachability of two ports, that is, whether any packet can reach from port A to B and if so, which packets? However, using HSA, one can make a transfer function for each box in the network by reading and parsing the forwarding states, and use those transfer functions for answering reachability questions.

HSA is a useful foundation for building techniques and tools for network verification, testing, and debugging. In this dissertation I will describe three set of techniques and tools for network verification and testing based on HSA.

1. In Chapter 3, I use HSA to develop algorithms and build a tool, called *Hassel*, for static analysis of networks. *Hassel* can answer questions that are critical for the operational correctness of networks, such as determining the reachability between end hosts, detecting forwarding loops, counting the repetition of loops, and checking the isolation of network slices.
2. In Chapter 4, I describe the design and implementation of a tool based on HSA, called *NetPlumber*, which can verify—in real time—a wide range of policies including header and path predicates on flows, lack of forwarding loops, and black hole freedom.
3. In Chapter 5, I introduce a framework called *Automatic Test Packet Generation (ATPG)* which uses HSA to automatically generate a minimal set of test packets to maximally test all forwarding rules, queues, or links in the network. By periodically sending these test packets, ATPG can test the liveness of the underlying network topology and the congruence between the data plane state and the observed behavior of the network at all times.

All of these tools were tested on real-world networks. *Hassel* found some loops in Stanford University’s backbone network, *NetPlumber* verified—in real time—the

all-pair connectivity of Google's data centers via its wide area network, and ATPG detected and localized a fault as a result of a configuration mistake in the network of the computer science and electrical engineering department at Stanford University.

This dissertation shows that by making the right model for the forwarding functionality of networks, we can create verification, testing, and debugging tools that are practical, systematic, and provably correct. Tools similar to the ones introduced in this dissertation can replace the ad hoc methods used for network troubleshooting today, and as a result, can greatly simplify the job of network administrators and reduce the operational costs of networks.

Acknowledgements

My advisor, Prof. Nick McKeown, has shaped my professional life more than anyone else in the world. Over the years, Nick taught me to think big, to strive for impact, and to never get discouraged. Nick was not only a great source of wisdom, fresh ideas, and encouragement for my PhD research, but he was a great friend and caring advisor. I could not ask for a better advisor than Nick, and I am very grateful to him.

I was very fortunate to meet and work with Prof. George Varghese during his visit to Stanford in 2010; It has been a prolific relationship that continues to this day and hopefully will continue for many years to come. Over the years, I have enjoyed many energetic discussions with George, been inspired by his enthusiasm, and benefited from his brilliant new ideas. I am very thankful to him.

Guru Parulkar has been a caring advisor who, despite being very busy, always had his door open for me. I enjoyed many great conversations with Guru and benefited from his advice, ideas, and feedback. I truly appreciate his help.

I would like to thank the other members of my oral committee, Prof. David Dill, Prof. Balaji Prabhakar, Prof. Sachin Katti, and Prof. Scott Shenker for their hard questions, which helped refine the written version of this dissertation. I also acknowledge my reading committee for their valuable comments and feedback on the written version of this dissertation.

I enjoyed my time at the McKeown Group and learned a lot from everyone in the group. My colleagues at the McKeown Group helped me refine my ideas, write better code, publish better papers, and give more effective presentations. This hard-working, brilliant, and friendly group has always been a great source of inspiration for

me, and I am honored to know them and grateful to work with them. I want to more specifically thank James Hongyi Zeng, who was a great collaborator and without whose contributions many parts of this dissertation would not have been possible; Michael Chang, who helped with the C implementation of Hassel; Kok Kiong Yap, who carefully reviewed an early draft of this dissertation and provided invaluable comments and feedback; Brandon Heller, whose special ability to give constructive feedback greatly improved the quality of the work that went into this dissertation; Masayoshi Kobayashi, who was always available to help me with anything in my projects; Neda Beheshti, who was a great help and support for me during my early days at the McKeown group; and Ali Al-Shabibi, Adam Covington, Saurav Das, Jonathan Ellithorpe, David Erickson, Glen Gibb, Nikhil Handigol, Te-Yuan Huang, Jad Naous, Alireza Sharafat, Rob Sherwood, Tatsuya Yabe and Yiannis Yiakoumis with whom I had very informative conversations over the years.

I am grateful to our network administrators at Stanford, Johan van Reijendam, Charlie Orgish, Miles Davis, and Joe Little, for sharing their experiences with me and giving me access to Stanford’s network data.

My parents have always given me unconditional love and unrelenting support in every possible way. I would like to dedicate this—as a small form of appreciation—to my parents, who always cherished higher education and wanted me to become a “doctor.” I also want to thank my brother, Pooyan, my “American parents”, Connie and Edward Barthold, and my new parents, brothers and sisters—a.k.a. my in-laws—for supporting me emotionally and always being there for me.

Last, but absolutely not least, I want to express my greatest gratitude to the sweetest and loveliest life companion I could wish for; my “dream,” Roya. Her love, care, and support made it possible for me to make it through tough times and harsh deadlines, and I am also dedicating this dissertation to her.

To my parents, Azar and Mahmoud,

&

To my wife, Roya.

Contents

Abstract	v
Acknowledgements	ix
1 Introduction	1
1.1 Problem Statement	1
1.2 Motivations	1
1.3 Network Troubleshooting Today	3
1.4 My Thesis Goal	5
1.5 Header Space Analysis: A Brief Overview	6
1.6 Thesis Organization	7
2 Header Space Analysis	9
2.1 Terminology	9
2.1.1 Header Space, \mathcal{H}	10
2.1.2 Network Space, \mathcal{N}	10
2.1.3 Switch Transfer Function, $T()$	11
2.1.4 Network Transfer Function, $\Psi()$	11
2.1.5 Topology Transfer Function, $\Gamma()$	12
2.2 Modeling Networking Boxes	13
2.2.1 IPv4 Router	14
2.2.2 Firewall	16
2.2.3 Tunneling End Points	16
2.2.4 Network Address Translator	17

2.2.5	Load Balancer	19
2.3	Header Space Algebra	19
2.3.1	Set Operations on \mathcal{H}	20
2.3.2	Domain, Range and Inverse of Transfer Function	22
2.4	Limitations	22
2.5	Related Work	23
2.6	Summary	23
3	Network Verification with Header Space Analysis	25
3.1	Finding Reachability	25
3.1.1	Reachability Algorithm	26
3.1.2	Checking Path Predicates	31
3.1.3	Complexity of Finding Reachability	31
3.2	Detecting Forwarding Loops	33
3.2.1	Finding Generic loops	34
3.2.2	Finding Repetition of Forwarding Loops	36
3.2.3	Dealing with Tortuous Forwarding Loops	38
3.2.4	Complexity of Finding Loops	39
3.3	Checking Isolation of Network Slices	40
3.3.1	Checking Disjointness of Slice Definitions	41
3.3.2	Detecting Packet Leakage	41
3.3.3	Complexity of Checking Slice Isolation	42
3.4	Hassel: the Header Space Library	43
3.4.1	Algorithmic Optimizations	44
3.4.2	Implementation Optimizations	48
3.5	Evaluation	49
3.5.1	Verification of an Enterprise Network	50
3.5.2	Checking Slice Isolation	53
3.5.3	Debugging a Protocol Design	55
3.6	Limitations	59
3.7	Related Work	59

3.8	Summary	60
4	Real-Time Policy Checking using Rule Dependency Graph	61
4.1	NetPlumber Architecture	62
4.1.1	Overview of NetPlumber	62
4.1.2	The Rule Dependency Graph	65
4.1.3	Source and Sink Nodes	67
4.1.4	Probe Nodes	68
4.1.5	Updating NetPlumber State	69
4.1.6	Complexity Analysis	71
4.2	Checking Policies and Invariants	72
4.2.1	Flowexp Language	73
4.2.2	Checking Loops and Black Holes	75
4.2.3	Checking Policies on Path Predicates	75
4.2.4	Policy Translator	77
4.3	Distributed NetPlumber	78
4.4	Implementation	81
4.5	Evaluation	82
4.5.1	Our Data Set	83
4.5.2	All-pair Connectivity of Google WAN	84
4.5.3	Checking Policy in Stanford network	87
4.5.4	Performance Benchmarking	87
4.6	Discussion	88
4.6.1	Other Use Cases	89
4.6.2	Limitations	90
4.7	Related Work	90
4.8	Summary	92
5	Automatic Test Packet Generation	93
5.1	ATPG System	95
5.1.1	Test Packet Generation	96
5.1.2	Fault Localization	100

5.2	Use Cases	103
5.2.1	Functional Testing	103
5.2.2	Performance Testing	104
5.3	Implementation	105
5.3.1	Test Packet Generator	105
5.3.2	Network Monitor	106
5.3.3	Alternate Implementations	106
5.4	Evaluation	107
5.4.1	Test Packet Generation	107
5.4.2	Testing in an Emulated Network	109
5.4.3	Testing in a Production Network	112
5.5	Discussion	115
5.6	Related Work	116
5.7	Summary	117
6	Conclusion	119
6.1	Summary of Dissertation	119
6.2	Contributions	121
6.3	Future Directions	122
6.4	Closing Remarks	123
	Bibliography	125

List of Tables

1.1	Rankings of various error symptoms in networks by administrators. . .	4
1.2	Rankings of various causes of network errors by administrators. . . .	4
1.3	Rankings of tools usage by administrators.	5
2.1	Example: Extracting Transfer Function from an IPv4 Routing Table.	15
3.1	Impact of optimization techniques on the run time of the reachability and loop detection algorithms.	45
3.2	Run time of the loop detection and reachability tests on Stanford backbone network, using implementations of Hassel in Python and C. . .	52
4.1	Average and median run time of the distributed NetPlumber, checking all-pair connectivity policy on Google WAN.	86
4.2	Average and median run time of NetPlumber for a single rule and link update when only one source node is connected to NetPlumber. . . .	88
5.1	All-pairs reachability table: all possible headers from every terminal to every other terminal, along with the rules they exercise.	98
5.2	Test packets for the example network depicted in Figure 5.3. p_6 is stored as a reserved packet.	98
5.3	Test packet generation results for Stanford backbone (top) and Internet2 (bottom), against the number of ports selected for deploying test terminals.	108
5.4	Test packets used in the functional testing example.	111

List of Figures

1.1	Geometric representation of packets in header space.	6
2.1	Multi-hop traversal of packets in a network by applying $\Phi(.) = \Gamma(\Psi(.))$	12
3.1	DFS algorithm for finding the range of the reachability function between a and b	27
3.2	A toy network topology used for computing reachability from a to b	29
3.3	Computing reachability from a to b in a toy example network.	29
3.4	Propagation graph when finding reachability from a to b in the network of figure 3.2.	30
3.5	Fragmentation of the header space as it passes through the network.	33
3.6	Total number of wildcard expressions generated in the process of computing reachability, counted at each propagation hop.	34
3.7	An example of running the complete loop detection check.	35
3.8	Example of a propagation graph for a test packet injected from port A_1 in the network of figure 3.7.	36
3.9	Relative position of h_{ret} and h_{orig} and their effect on loop repetition.	37
3.10	A recursive algorithm for finding loop repetition of each header involved in a loop.	39
3.11	Example of a tortuous loop consisting of multiple loops.	39
3.12	Detecting slice leakage by finding the image of slice a under network transformation and intersecting it with slice b	42
3.13	Architecture of Hassel—the Header Space Library.	44
3.14	Compressing IP forwarding table using a binary tree representation.	45

3.15	Lazy evaluation of transfer function rules: by switching the order of applying transfer function rules, the overall computation complexity is reduced.	48
3.16	Topology of the backbone network of Stanford university and three types of loops detected using Hassel. Overall, we found 26 loops on 14 loop paths, 10 of which are infinite.	50
3.17	Run time of the slice isolation check on Stanford backbone network for randomly generated slices. The check uses the Python based implementation of Hassel.	54
3.18	Header format and stack rotation of IP* protocol.	55
3.19	Example of an IP* network with routing through middleboxes.. . . .	56
3.20	Alice’s second network topology can cause infinite loops.	57
3.21	Running time of the loop detection test on Alice’s IP* network using the Python based Hassel.	57
3.22	Loop detection code	58
4.1	Deploying NetPlumber as a policy checker in SDNs.	63
4.2	Example of finding next hop dependency in NetPlumber. Here, the switches are connected by a link and the range of $R1$ and the domain of $R2$ has some intersection: 01x.	64
4.3	Rule dependency graph of a simple network.	66
4.4	Finding reachability between S and P using the rule dependency graph.	67
4.5	Updating the rule dependency graph after rule insertion.	70
4.6	Updating the rule dependency graph after rule deletion.	71
4.7	Flowexp language grammar.	74
4.8	Clustering the rule dependency graph to enable parallelization.	79
4.9	NetPlumber software block diagram and its dependency on Hassel.	83
4.10	Google inter-datacenter WAN network—July 2012.	84
4.11	CDF of the run time of NetPlumber per update, when checking the all-pair reachability constraint in Google WAN with 1-5 instances and in Stanford backbone with a single instance.	86

5.1	Condition for correctness of a network: policy = forwarding state = actual behavior.	94
5.2	ATPG system block diagram.	96
5.3	Example topology with three switches used for finding test packets. .	99
5.4	Generating packets to test drop rules by “flipping” the drop rule to a broadcast rule in the analysis.	104
5.5	The cumulative distribution function of rule repetition, ignoring different action fields.	109
5.6	A portion of the Stanford backbone network showing the test packets used for functional and performance testing examples as described in Section 5.4.2.	110
5.7	Priority testing: Latency measured by test agents when (a) low-priority or (b) high-priority slice is congested.	113
5.8	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).	114
6.1	Relation between Hassel, NetPlumber, ATPG and the Header Space Analysis.	120

Chapter 1

Introduction

1.1 Problem Statement

It is notoriously hard to debug networks. It requires analyzing the state of multiple devices from different vendors running numerous protocols and distributed across multiple tables. Yet, there are ver few methods and tools to help network administrators to troubleshoot networks. Instead network administrators must rely on their wisdom and experience of network administrators to hunt down root causes of problems using simple tools such as `ping` and `traceroute`. My goal in this dissertation is to develop a rigorous foundation for analyzing the behavior of networks and use it to build methods and tools to help network admins with testing and debugging networks.

1.2 Motivations

In the beginning, a switch or router was breathtakingly simple. About all the device needed to do was index into a forwarding table using a destination address and decide where to send the packet next. Over time, forwarding grew more complicated. Middleboxes (e.g., NAT and firewalls) and encapsulation mechanisms (e.g., VLAN and MPLS) appeared to overcome IP's limitations: e.g., NAT bypasses address limits and MPLS allows flexible routing. Further, new protocols for specific domains, such

as data centers, WANs and wireless, have greatly increased the complexity of packet forwarding. Today, there are over 6,000 Internet RFCs, and it is not unusual for a switch or router to handle ten or more encapsulation formats simultaneously. At the same time, networks are growing in size: modern data centers may contain 10,000 switches, a campus network may serve 50,000 users, and a 100 Gb/s long haul link may carry 100,000 flows.

Despite this added complexity, managing a network is still mostly a manual process. When a network administrator adds a new route to the network, he or she must login to configure each switch or router along the path. The process is cumbersome and error-prone; in a recent survey [57] network administrators reported configuration errors as one of the most common problems in their network. Also the tools for debugging networks are very rudimentary: network engineers hunt down bugs using the simple tools (e.g., `ping`, `traceroute`, SNMP, and `tcpdump`), and track down root causes using a combination of accrued wisdom and intuition, as evidenced in [57]. No wonder that network engineers have been labeled “masters of complexity” [47].

Troubleshooting a network is difficult for two reasons. First, the forwarding state is distributed across multiple routers, firewalls and other boxes and is defined by their forwarding tables, filter rules and other configuration parameters. As a result, the forwarding state is hard to observe and parse, because it typically requires manually logging into every box in the network and understanding every protocol and CLI output format. Second, there are many different programs, protocols and humans updating the forwarding state simultaneously, which interact in complex ways. The forwarding state of networks is not generated in a way that lends itself well to verification. As a result, the best practice for network debugging today is to use ad hoc tools like `ping` and `traceroute` to indirectly probe and infer the current set of forwarding rules and use that information to debug the network.

As a consequence, even simple questions about a network are hard to answer. For example:

1. Can host *A* talk to host *B*? If so, using which packet headers?
2. If host *A* can't talk to host *B*, where are the packets dropped?

3. What will happen if I remove a forwarding entry from a router or a line from a configuration file?
4. Are two slices of my network completely isolated?
5. Is there a forwarding loop in my network?
6. Why is my network so slow? Where is the congestion happening?

As we will see in Section 1.3, the answers to these questions are directly related to the type of problems most commonly seen by network administrators.

1.3 Network Troubleshooting Today

In this section, I report the result of a survey conducted during May–June 2012 to gather data from 61 subscribers to the NANOG¹ mailing list (full report available in [57]). This survey was conducted to understand the problems administrators face in their networks, the potential causes and how problems are diagnosed today. The respondents included 12 administrators of small ($< 1k$ hosts) networks, 23 of medium ($1k - 10k$ hosts) networks, 11 of large ($10k - 100k$ hosts) and 12 of very large ($> 100k$ hosts) networks. The goal was to understand what network debugging in the real world looks like, and how well the solutions offered in the rest of this dissertation address these challenges.

Network administrators face various “symptoms” and “diseases” in their networks. Table 1.1 demonstrates that “reachability” and “throughput/latency” problems are among those that happen most often. Not only are the faulty behaviors diverse, but the possible causes are also complex. “Hardware failure”, “switch/router software bug” and “misconfigurations” are among the top three causes for network problems (Table 1.2). A long tail of other symptoms/causes exists, which complicates the search space during debugging. The categories are coarse-grained, so there can be more symptoms/causes that are missed.

¹North American Network Operators’ Group

We are also interested in the debugging tools that administrators use today (Table 1.3). `ping`, `traceroute` and SNMP are the most popular tools. When asked what the ideal tool for network debugging would be, 70.7% of respondents thought that automatic test generation to check performance and correctness problems are important. Some of them explicitly write down “long running tests to detect jitter or intermittent issues,” “real-time link capacity monitoring,” “monitoring tools for network state,” etc.

Category	Avg	% of ≥ 4
Reachability Failures	3.67	56.90%
Intermittent Problems	3.38	53.45%
Throughput Degradation/High Latency	3.39	52.54%
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%

Table 1.1: Rankings of various error symptoms in networks by administrators. 5=most often, 1=least often. Average rankings and percentages of respondents who ranked ≥ 4 are shown.

Category	Avg	% of ≥ 4	
Misconfigurations	Protocol Misconfig.	2.29	23.64%
	ACL Misconfig.	2.44	20.00%
	QoS/TE Misconfig.	1.70	7.41%
External Errors	3.06	42.37%	
Hardware Failure	3.07	41.07%	
Switch/Router Software Bug	3.12	40.35%	
Attacks (DOS, Security, etc)	2.67	29.82%	
Software Upgrade	2.35	18.52%	
Host Network Stack Bug	1.98	16.00%	

Table 1.2: Rankings of various causes of network errors by administrators. 5=most often, 1=least often. Average rankings and percentages of respondents who ranked ≥ 4 are shown.

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

Table 1.3: Rankings of tools usage by administrators. 5=most often, 1=least often. Average rankings and percentages of respondents who ranked ≥ 4 are shown.

1.4 My Thesis Goal

As stated in Section 1.2, the forwarding state of networks is not created in a way that lends itself well to verification. As a result, it is hard to formally verify the correctness of networks and pinpoint the errors. However, I believe that if we define the right abstraction model for the forwarding functionality of networks, regardless of where that forwarding state comes from or to which protocol or table it belongs to, then we can use it to design algorithms and build tools to formally verify and debug networks.

In fact, defining the right abstraction model is an important step in analyzing complex systems in the other fields of engineering as well. For example, in the field of communication engineering, a typical system consists of many different components such as filter, amplifier, antenna and communication channel. To analyze these systems, the behavior of each component is modeled using a *transfer function*. Transfer function provides a simple and unified representation of the behavior of various components in the system which greatly simplifies the analysis. Similarly, in digital hardware design, instead of looking at individual transistors that make a circuit, the logical gate abstraction is used to model groups of transistors. The logical gate abstraction enables the use of higher-level methods for analyzing and designing a digital circuits, such as boolean algebra and Karnaugh map.

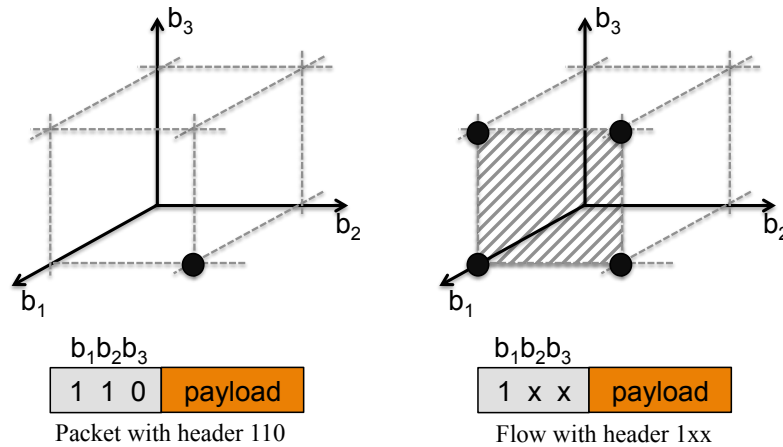


Figure 1.1: Geometric representation of packets in header space.

Inspired by this, I will define a simple, protocol-independent abstraction for the forwarding functionality of networks and use it to build a framework for analyzing networks, called *the Header Space Analysis (HSA)* framework. This framework gives us a unified view of the forwarding state of all boxes in a network in such a way that is suitable for formally checking different properties and invariants of the network. To demonstrate the power of this framework, I will use it to design algorithms for checking important network properties such as reachability between hosts, lack of forwarding loops and isolation of network slices. I will also use HSA to build a tool for real-time policy checking in networks and another framework for monitoring health of network data plane by generating carefully-crafted test packets. I will show how these techniques work on real-world networks such as the Stanford University backbone network, the Internet 2 nationwide network and the Google inter-data center WAN.

1.5 Header Space Analysis: A Brief Overview

Header Space Analysis (HSA) [23] is an abstraction model for the forwarding functionality of networks and a framework for analyzing networks. HSA is the foundation for all the algorithms, techniques and tools introduced in this dissertation. Key to

HSA is a generalization of the geometric approach to packet classification in which classification rules over K packet fields are viewed as subspaces in a K dimensional space [29]. HSA jettisons the notion of pre-specified fields in favor of a header space of L bits where each packet is represented by a point in $\{0, 1\}^L$ space, where L is the header length (see Figure 1.1). This allows us to work with emerging protocols and arbitrary field formats.

HSA models networking boxes using a *Switch Transfer Function* T , which transforms a header h received on input port p to a set of packet headers on some output ports: $T : (h, p) \rightarrow \{(h_1, p_1), (h_2, p_2), \dots\}$. Each transfer function consists of an ordered set of *rules* R . A typical rule consists of a set of physical *input ports*, a *match* which is a wildcard expression, and a set of actions to be performed on packets that match the wildcard expression. Examples of actions include forward to a port, drop, rewrite, encapsulate, and decapsulate. Network topology is modeled using a *Topology Transfer Function*, Γ , which models the physical connection between ports. If port p_{src} is connected to p_{dst} by a physical link, then Γ will have a rule that transfers (h, p_{src}) to (h, p_{dst}) : $\Gamma(h, p_{src}) = (h, p_{dst})$.

One application of the HSA framework is to compute reachability of packets and flows in the network: to find how flow f , represented by its header wildcard expression (h_f) , reaches from port A to B , we need to compose transfer functions as follows:

$$R_{A \rightarrow B} = \bigcup_{A \rightarrow B \text{ paths}} \{T_n(\Gamma(T_{n-1}(\dots(\Gamma(T_1(h_f, A))\dots)))\},$$

i.e., for each possible path between A and B ($A \rightarrow S_1 \rightarrow \dots \rightarrow S_{n-1} \rightarrow S_n \rightarrow B$), we find how the flow along that path is processed by networking boxes. The flow at the destination port is the sum (union) of all the sub-flows received on each of these paths.

1.6 Thesis Organization

I start by introducing the Header Space Analysis (HSA) framework in Chapter 2. I will define header space, network space, transfer function and header space set

algebra. These are the basic building blocks of HSA. I show how we can create transfer function for different boxes in a network.

In Chapter 3, I show how HSA framework can be used for network verification. In particular, I will introduce algorithms for finding reachability, detecting forwarding loops and checking isolation of network slices and will discuss their run times. Then, I will describe our implementation of the header space library (Hassel) in C and Python and discuss the implementation of these verification algorithms on top of Hassel along with some examples. Chapter 2 and 3 of this dissertation were first published in [23].

Next, in Chapter 4 I will show how the HSA framework can be used as the building block for a system to check design policies and invariants of networks in real time and showcase the performance on three real-world networks. This chapter is based on [22].

To detect hardware failures such as link or ASIC failure or performance issues such as congestion, we need to passively monitor or actively test the data plane. In Chapter 5, I will introduce a framework called *Automatic Test Packet Generation (ATPG)*, which uses HSA as its foundation to generate test packets in way that maximizes coverage of network and minimizes the number of test packets required. The content of this chapter was originally published in [56].

Finally, I will conclude this dissertation in Chapter 6. I will summarize my contributions, put them into perspective, and mention some future directions.

Chapter 2

Header Space Analysis

The goal of this chapter is to define a simple, protocol independent abstraction for packets and forwarding functionality of networks that can be used as foundation for systematic verification of networks. To achieve this, I developed a general and protocol-agnostic framework called *Header Space Analysis* in which protocol header fields are not first-class entities; instead, we look at the entire packet header as a concatenation of bits without any associated meaning. Each packet is a point in the $\{0, 1\}^L$ space, where L is the maximum length of a packet header, and networking boxes transform packets from one point in the space to another point or set of points. The concepts defined here will be used as basic building blocks throughout this thesis and will lead to design and implementation of practical tools for network testing and debugging.

2.1 Terminology

The header space framework is built on a geometric model of packets. Packets are represented as points *in* a geometric space, and network boxes are transfer functions *on* the same space. We should first define these spaces.

2.1.1 Header Space, \mathcal{H}

Packets in a network are processed based on their header bits. Therefore, as far as forwarding functionality of networks is concerned, a packet is represented by its header bits. To define a protocol-independent model for packets, we ignore the protocol-specific meanings associated with header bits (i.e., fields) and view a packet header as a flat sequence of ones and zeros. Formally, a packet is a point and a flow is a region in the $\{0, 1\}^L$ space, where L is an upper bound on the header length and each bit corresponds to one dimension in the space. We call this space *Header Space*, \mathcal{H} . Figure 1.1 shows a simple example in which three-bit headers are represented geometrically in the header space.

A wildcard expression is the basic building block used to define objects in \mathcal{H} . Each wildcard expression is a sequence of L bits, where each bit can be either 0, 1 or x. Each wildcard expression corresponds to a hypercube in \mathcal{H} . Every region, or flow, in \mathcal{H} is defined as a *union* of wildcard expressions.

\mathcal{H} abstracts away the data portion of a packet because we assume it does not affect packet processing. If it does, as in an intrusion-detection box, then L must be the length of the entire packet. If the fields are fixed, we can define macros for each field in \mathcal{H} to reduce dimensionality. However, the general notion of header space is critical when dealing with different protocols that interpret the same header bits in different ways.

2.1.2 Network Space, \mathcal{N}

We model the network as a set of boxes called *switches* with external interfaces called *ports*, each of which is modeled as having a unique identifier. We use “switches” to denote routers, bridges and any other middleboxes.

If we take the cross-product of the switch-port space (the space of all ports in the network, \mathcal{S}) with \mathcal{H} , we can represent a packet traversing on a link as a point in $\{0, 1\}^L \times \{1, \dots, P\}$ space, where $\{1, \dots, P\}$ is the list of ports in the network. We call the space of all possible packet headers, localized at all possible input ports in the network, the *Network Space*, \mathcal{N} .

2.1.3 Switch Transfer Function, $T()$

As a packet traverses the network, it is transformed from one point in Network Space to other point(s) in Network Space. For example, a layer 2 switch, that merely forwards a packet from one port to another, without rewriting headers, transforms packets only along the switch-port axis, \mathcal{S} . On the other hand, an IPv4 router that rewrites some fields (e.g., MAC address, TTL, checksum) and then forwards the packet, transforms the packet both in \mathcal{H} and \mathcal{S} .

As these examples suggest, all networking boxes can be modeled as *Transformers*, with a *Transfer Function*, that models their protocol dependent functions. More precisely, a node can be modeled using its transfer function, T , that maps header h arriving on port p :

$$T(h, p) : (h, p) \rightarrow \{(h_1, p_1), (h_2, p_2), \dots\}$$

In general, the transfer function may depend on the input port to model input-port-specific behavior, and the output may be a set of (*header, port*) pairs to allow multicasting.¹

A transfer function consists of an ordered set of *rules*. A typical rule consists of a *match* condition, which determines the packets to be processed by the rule, and an *action*, which is the processing to be done on the matching packets. The match part of the rule may include a set of *input ports* and a *match wildcard expression*. Examples of actions include forward to a port, drop, rewrite, encapsulate and decapsulate. In Section 2.2, we will see how to make a transfer function for some networking boxes used today.

2.1.4 Network Transfer Function, $\Psi()$

A notation that we use heavily is the *network transfer function*, $\Psi(\cdot)$. Given that switch ports are numbered uniquely, we combine all the switch transfer functions

¹It also enables us to model load balancing boxes for which the output port is a pseudo-random function of the header bits.

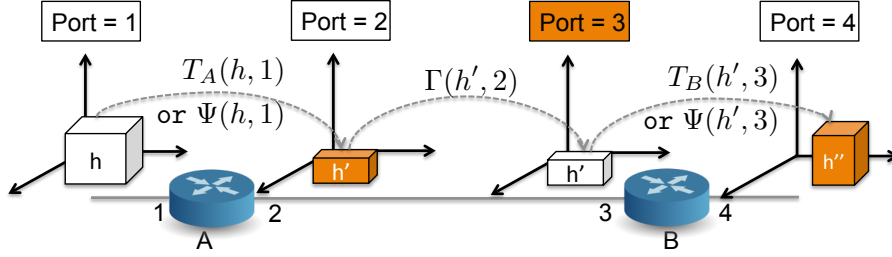


Figure 2.1: Multi-hop traversal of packets in a network by applying $\Phi(\cdot) = \Gamma(\Psi(\cdot))$. Header h at port 1 is transformed to h' at port 2 by transfer function of box A (or equivalently by Ψ). h' is then forwarded to port 3 by Γ , where it is transformed to h'' by T_B (or equivalently by Ψ).

into a composite transfer function describing the overall behavior of the network. Formally, if a network consists of n boxes with transfer functions $T_1(\cdot), \dots, T_n(\cdot)$, then

$$\Psi(h, p) = \begin{cases} T_1(h, p) & \text{if } p \in \text{switch}_1 \\ \dots & \dots \\ T_n(h, p) & \text{if } p \in \text{switch}_n \end{cases}$$

2.1.5 Topology Transfer Function, $\Gamma(\cdot)$

A unidirectional link connects a source port P_{src} to a destination port P_{dst} and delivers packets from P_{src} to P_{dst} . The topology of a network is defined by the set of links in the network, each represented by its source and destination ports. We can model the network topology using a *topology transfer function*, $\Gamma(\cdot)$, defined as:

$$\Gamma(h, p) = \begin{cases} \{(h, p^*)\} & \text{if } p \text{ connected to } p^* \\ \{\} & \text{if } p \text{ is not connected.} \end{cases}$$

Γ models the behavior of links in the network. It accepts a packet at one end of a link and returns the same packet, unchanged, at the other end. Note that links are unidirectional in this model. To model bidirectional links, one rule should be added per direction.

Using the network and topology transfer function, we can model a packet as it traverses the network by applying $\Phi(.) = \Gamma(\Psi(.))$ at each hop. For example, if a packet with header h enters a network on port p , the header after k hops will be $\Gamma(\Psi(\dots(\Gamma(\Psi(h,p))\dots))$, or simply $\Phi^k(h,p)$: each Γ forwards the packet on a link and each Ψ passes the packet through a box.

2.2 Modeling Networking Boxes

As we will see in the next chapters, transfer functions will be used to analyze of networks quite extensively. We wrote parsing scripts that read the configuration state and forwarding tables from network boxes and automatically create transfer functions. To solidify our understanding of transfer functions, this section shows how they can model different boxes, illustrating their power in modeling in a unified way. As we will see in this section, transfer functions can perfectly capture the behavior of stateless devices. However, stateful devices whose behavior change based on external factors such as load condition or history of observed packets cannot be accurately modeled by a transfer function.

To be more clear and concise, we use the following helper macros in this section:

`protocol_field()` : refers to a particular field in a particular protocol. For example, `ip_src(h)` refers to the source IP address bits of header h .

`$\mathcal{R}(h, fields, values)$` : refers to a rewrite action in which the *fields* in h are rewritten with *values*. For example, `$\mathcal{R}(h, mac_dst, d)$` rewrites the MAC destination address to d . Note that the rewrite action can be implemented by simple logical operations. To rewrite some header bits, we need to first zero them out using a *masking AND*, and then rewrite the new value by an *OR* operation. For example, assume we have an 8-bit header, `xx0101xx`. To write the right-most three bits with value 011, we need to first *AND* it with 11111000: `xx0101xx & 11111000 = xx010000`. Then we need to *OR* the result with 00000011: `(xx0101xx & 11111000) | 00000011 = xx010011`.

2.2.1 IPv4 Router

Let's start by modeling an IPv4 router that processes packets as follows: 1) decrement TTL, 2) update checksum, 3) rewrite source and destination MAC addresses and 4) forward to outgoing port. Thus, the transfer function of an IPv4 router composes four functions as follows:

$$T_{IPv4}(\cdot) = T_{fwd}(T_{mac}(T_{chksum}(T_{ttl}(\cdot)))).$$

We examine each function in turn. $T_{fwd}(\cdot)$ looks up $ip_dst(h)$ in a lookup table and returns the output port. If we represent this function as $ip_lookup(h)$, then

$$T_{fwd}(h, p) = \{(h, ip_lookup(ip_dst(h)))\}$$

As a simple example, assume we have an IPv4 router with the forwarding table as in Table 2.1. Then the forwarding transfer function would look like this:

$$T_{fwd}(h, p) = \begin{cases} \{(h, 1)\} & \text{if } ip_dst(h) \in 192.168.1.x \\ \{(h, 2)\} & \text{if } ip_dst(h) \in 192.168.2.x \\ \{(h, 3)\} & \text{if } ip_dst(h) \in 192.168.x.x - (192.168.1.x \cup 192.168.2.x) \\ \{\} & \textit{otherwise.} \end{cases}$$

To construct this transfer function, the routing table and interface configuration information is used. Note that the third rule in the routing table has lower priority over the first two rules; therefore, it only matches on packets that don't match on them. Also, we have a default drop rule in the transfer function for packets with no matching forwarding entries.

Similarly, $T_{mac}(\cdot)$ looks up the next hop MAC address and updates source and destination MAC addresses. If we are interested in including MAC rewriting in the final transfer function, we can apply the appropriate rewrite action to the output

Routing Table			
Network Address	Network Mask	Gateway	Interface
192.168.1.0	255.255.255.0	192.168.1.1	192.168.1.100
192.168.2.0	255.255.255.0	192.168.2.1	192.168.2.100
192.168.0.0	255.255.0.0	192.168.0.1	192.168.0.100

Interface Configuration		
Name (ID)	Interface MAC address	Interface IP address
1	00:00:00:00:00:01	192.168.1.100
2	00:00:00:00:00:02	192.168.2.100
3	00:00:00:00:00:03	192.168.0.100

ARP Table		
IP Address	MAC address	Interface Name(ID)
192.168.1.1	00:00:00:00:00:0A	1
192.168.2.1	00:00:00:00:00:0B	2
192.168.0.1	00:00:00:00:00:0C	3

Table 2.1: Example: Extracting Transfer Function from an IPv4 Routing Table.

header. In the previous example, this could be done as follows:

$$T_{fwd}(T_{mac}(h, p)) = \left\{ \begin{array}{ll} \{(\mathcal{R}(h, [\text{mac_src}, \text{mac_dst}], [:01, :0A]), 1)\} & \text{if } \text{ip_dst}(h) \in 192.168.1.x \\ \{(\mathcal{R}(h, [\text{mac_src}, \text{mac_dst}], [:02, :0B]), 2)\} & \text{if } \text{ip_dst}(h) \in 192.168.2.x \\ \{(\mathcal{R}(h, [\text{mac_src}, \text{mac_dst}], [:03, :0C]), 3)\} & \text{if } \text{ip_dst}(h) \in 192.168.x.x - \\ & (192.168.1.x \cup 192.168.2.x) \\ \{\} & \text{otherwise.} \end{array} \right.$$

The TTL decrement can also be included in the final transfer function. $T_{ttl}(\cdot)$ drops the packet if $\text{ip_ttl}(h)$ is 0 and otherwise does $\mathcal{R}(h, \text{ip_ttl}(), \text{ip_ttl}(h) - 1)$. $T_{checksum}(\cdot)$ updates the IP checksum. Depending on the problem at hand, one can include as much detail as desired in the final transfer function. For most applications, only including $T_{fwd}()$ or $T_{fwd}(T_{ttl}())$ will suffice.

2.2.2 Firewall

A firewall blocks access to certain IP address or transport port numbers, based on a set of rules that is usually called a access control list, or ACL. As an example, consider a very simple ACL with two rules: one that denies access to IP address A unless TCP port = Q , and a second rule that denies access to IP address B if source IP address is C . The transfer function of the firewall is

$$T_{acl}(h, p) = \begin{cases} \{\} & \text{if } ip_dst(h) = A \ \& \ tcp_dst(h) \neq Q \\ \{\} & \text{if } ip_dst(h) = B \ \& \ ip_src(h) = C \\ \{(h, p)\} & \text{otherwise} \end{cases}$$

2.2.3 Tunneling End Points

We can model a tunneling end point using a shift and a rewrite operator. To model an encapsulation action that put a k -bit encapsulating header, h_{encap} , at bit position s of an L bit header, h , we can do the following:

$$\begin{aligned} \text{Let } M &:= \underbrace{1 \dots 1}_{s \text{ times}} \underbrace{0 \dots 0}_{(L-s) \text{ times}} \\ \text{Let } h_1 &:= (h \ \& \ M) \\ \text{Let } h_2 &:= [(h \ \& \ \bar{M}) \gg k] \ | \ [h_{encap} \gg s] \\ \text{Then } T(h, p) &= \{(h_1 \ | \ h_2, p_{out})\} \end{aligned}$$

Here, h_1 keeps the first s bits of packet header unchanged. h_2 shifts² the last $L - s$ bits to the right by k bit positions and rewrites h_{encap} at those k positions. The final result is the concatenation of the first s bits from h_1 and the last $L - s$ bits from h_2 , achieved by a logical *OR* operation.

²Logical shift where zeros are shifted in to replace the discarded bits.

The decapsulation action is similar to encapsulation, but the shift is in the opposite direction. To decapsulate the above packet, we can do the following:

$$\begin{aligned}
 \text{Let } M &:= \underbrace{1 \dots 1}_{s \text{ times}} \underbrace{0 \dots 0}_{(L-s) \text{ times}} \\
 \text{Let } W &:= \underbrace{0 \dots 0}_{L-k \text{ times}} \underbrace{x \dots x}_{k \text{ times}} \\
 \text{Let } h_1 &:= (h \& M) \\
 \text{Let } h_2 &:= [(h \ll k) \& \bar{M}] \mid W \\
 \text{Then } T(h,p) &= \{(h_1 \mid h_2, p_{out})\}
 \end{aligned}$$

Again, h_1 keeps the first s bits of the packet header unchanged. h_2 drops bits s to $s+k$ by shifting them to the left by k bits and then masking them out. It also puts k wildcard bits at the right-most positions, because the values of these bits are unspecified after decapsulation.

2.2.4 Network Address Translator

Network Address Translators (NATs) are deployed at the boundary of private networks to share one public IP address among several private hosts. A NAT remembers the source IP address and transport port number of outbound packets and rewrites the transport source port to a unique number, which we represent as $\text{NAT}_{\text{out}}(h)$. For the inbound packets, it looks up the destination transport port number in its lookup table using $\text{NAT}_{\text{in}}(h)$, and returns the corresponding $(\text{ip_dst}, \text{tcp_dst})$ pair to be written back to the header. More concisely:

$$\text{NAT}_{\text{out}} : (\text{ip_src}(h), \text{tcp_src}(h)) \rightarrow T_{\text{out}},$$

where T_{out} is the source transport port of outbound packet and

$$\text{NAT}_{\text{in}} : \text{tcp_dst}(h) \rightarrow (T_{\text{in}}, \text{IP}_{\text{in}}),$$

where T_{in} is the destination transport port and IP_{in} the destination IP address to be written back to the inbound packet.

NAT boxes can be modeled in two levels of detail, depending on whether we are interested in the exact mapping state of the NAT box or not.

Exact Transfer Function of a NAT Box: Suppose a NAT box has a public IP address, IP_{nat} , and ports P_i and P_o are connected to private and public sides, respectively. Then we can write the exact transfer function of NAT box as follows:

$$T_{nat}(h, p) = \begin{cases} \{(\mathcal{R}(h, [\text{ip_src}, \text{tcp_src}], [IP_{nat}, T_{out}]), P_o)\} & \text{if } p = P_i \ \& \\ & \text{NAT}_{out}(h) = T_{out} \\ \{(\mathcal{R}(h, [\text{ip_dst}, \text{tcp_dst}], [IP_{in}, T_{in}]), P_i)\} & \text{if } p = P_o \ \& \\ & \text{NAT}_{in}(h) = (IP_{in}, T_{in}) \end{cases}$$

There remains one problem: The following transfer function uses the current translation state of the NAT box and uses that state to build the transfer function. As a result, it won't predict the behavior of the NAT box when a packet with new $[\text{ip_src}, \text{tcp_src}]$ is going through the box. This is due to the static nature of transfer functions, which makes them incapable of modeling dynamic boxes. To overcome such limitation, we need to resort to more general models such as the one that comes next.

Coarse Transfer Function of a NAT Box: Let's assume that the network behind the NAT has subnet S_{nat} . Also let \mathbf{X} denote a wildcard on a packet field. Then the overall behavior of the NAT box can be modeled using this transform function³:

$$T_{nat}(h, p) = \begin{cases} \{(\mathcal{R}(h, [\text{ip_src}, \text{tcp_src}], [IP_{nat}, \mathbf{X}]), P_o)\} & \text{if } p = P_i \ \& \\ & \text{ip_src}(h) \in S_{nat} \\ \{(\mathcal{R}(h, [\text{ip_dst}, \text{tcp_dst}], [S_{nat}, \mathbf{X}]), P_i)\} & \text{if } p = P_o \ \& \\ & \text{ip_dst}(h) = IP_{nat} \end{cases}$$

³Note that by rewriting ip_dst to S_{nat} , which is a set of IP addresses and not one IP address, we mean that the output packet can be any of the packets whose IP address $\in S_{nat}$.

The coarse model of NAT overcomes the shortcomings of the detailed model by replacing the `tcp_src` for outbound packets and `tcp_dst` for inbound packets with an unspecified port number⁴, X .

2.2.5 Load Balancer

A load balancer sends incoming packets to one of the output ports at random. One possible way to create a transfer function for a load balancer is to transfer input packets on all the output ports: $T(h, p) = \{(h, p_1), \dots, (h, p_n)\}$, where $\{p_1, \dots, p_n\}$ is the set of output ports. This is done to explore all possibilities at the output. Another model would be to add probability as the third variable to the transfer function:

$$T(h, p, r) = \{(h, p_1, r/n), \dots, (h, p_n, r/n)\}.$$

Here r is the probability of receiving a packet with header h , on port p . Each of the output packets is generated with probability r/n because there are n equal probability choices at the output.

2.3 Header Space Algebra

To use HSA to check network correctness conditions such as reachability, lack of forwarding loops or isolation of network slices, we need to determine how different header spaces overlap, whether a region of header space is a subset of the other or what headers at which input ports will generate a certain header at an output port. We therefore need to define basic set operations on \mathcal{H} : *intersection*, *union*, *complementation* and *difference*. We also define the *Domain*, *Range* and *Inverse* for transfer functions. In Chapter 3, we use this algebra for checking properties of networks.

⁴Note that the port translation in a NAT box is not deterministic.

2.3.1 Set Operations on \mathcal{H}

While set operations on bit vectors are well-known, we need set operations on *wildcard* expressions. Since all header space regions can be represented as a union of wildcard expressions, defining set operations on wildcard expressions allows these operations to carry over to any header space region. For the rest of this section, we overload the term *header* to refer to both packet headers (points in \mathcal{H}) and wildcard expressions (hyper-cubes in \mathcal{H}).

Intersection: For two headers to have a non-empty intersection, both headers *must* have the same bit value at every position that is not a wildcard. If two headers differ in bit b_i , then the two headers will be in different hyper-planes, defined by $b_i = 0$ and $b_i = 1$. On the other hand, if one header has an x (wildcard) in a position while the other header has a 1 or 0, the intersection is non-empty. Thus, the *single-bit intersection* rule for $b_i \cap b'_i$ is defined as follows:

	b'_i	0	1	x
b_i	0	0	z	0
	1	z	1	1
	x	0	1	x

In the table, z means the bitwise intersection is empty. The intersection of two headers is found by applying the single-bit intersection rule, bit-by-bit, to the headers. z is an “annihilator”: if any bit returns z, the intersection of all bits is empty. As an example, $10xx \cap 1xx0 = 10x0$ and $10xx \cap 0xx0 = z0x0 = \phi$. A simple trick allows efficient software implementation. If each bit in the header is encoded using *two* bits: $0 \rightarrow 01$, $1 \rightarrow 10$, $x \rightarrow 11$ and $z \rightarrow 00$, then the intersection is simply an *AND* operation on the encoded headers.

Union: In general, a union of wildcard expressions cannot be simplified. For example, no single header can represent the union of $11xx$ and $00xx$. This is why a header space object is defined as a union of wildcard expressions. In some cases, we can simplify the union (e.g., $11xx \cup 10xx$ simplifies to $1xxx$) by simplifying an equivalent Boolean expression. For example, $10xx \cup 11xx$ is equivalent to $b_4\bar{b}_3 \oplus b_4b_3$.

This allows the use of Karnaugh Maps and Quine-McCluskey [4] algorithms for logic minimization.

Complementation: The complement of header h —the union of all headers that do *not* intersect with h —is computed as follows:

```

 $h' \leftarrow \phi$ 
for bit  $b_i$  in  $h$  do
  if  $b_i \neq x$  then
     $h' \leftarrow h' \cup x \dots x \bar{b}_i x \dots x$ 
  end if
end for
return  $h'$ 

```

The algorithm finds all non-intersecting headers by replacing each 0 or 1 in the header with its complement and putting a wildcard in all other bit positions. This follows because just one non-intersecting bit (or z) in a term results in a disjointed header. For example, $(100x)' = 0xxx \cup x1xx \cup xx1x$.

Difference: The difference (or minus) operation can be calculated using intersection and complementation. $A - B = A \cap B'$. For example,

$$\begin{aligned}
 & 1xxx - 101x \\
 &= 1xxx \cap (101x)' \\
 &= 1xxx \cap (0xxx \cup x1xx \cup xx0x) \\
 &= \phi \cup 11xx \cup 1x0x \\
 &= 11xx \cup 1x0x;
 \end{aligned}$$

i.e., $1xxx - 101x$ will be all packets whose first bit is 1, and whose second bit is 1 or whose third bit is 0. The difference operation can be used to check the subset and equality condition:

$$\begin{aligned}
 A \subseteq B &\iff A - B = \phi \\
 A = B &\iff A - B = \phi \ \& \ B - A = \phi.
 \end{aligned}$$

2.3.2 Domain, Range and Inverse of Transfer Function

To capture the destiny of packets through a box or set of boxes, we define the *domain*, *range* and *range inverse* as follows:

Domain: The domain of a transfer function is the set of all possible (*header*, *port*) pairs that the transfer function accepts. Even headers for which the output action is to drop the packet belong to the domain.

Range: The range of a transfer function is the set of all possible (*header*, *port*) pairs that the transfer function can output after applying all possible inputs on every port.

Inverse of Transfer Functions: Some applications of header space analysis such as finding reachability and detecting forwarding loops requires working backwards from an output header to determine what input (*header*, *port*) pairs could have produced it. For a given header at an output port, (h_o, p_o) , $T^{-1}(h_o, p_o)$ is the set of all input headers at input ports, (h_i, p_i) , such that $(h_o, p_o) \in T(h_i, p_i)$:

$$T^{-1}(h_o, p_o) := \{(h, p) \mid (h_o, p_o) \in T(h, p)\}.$$

A transfer function maps each (h, p) pair to a set of other pairs. By following the mapping backward, we can invert a transfer function. By applying the range to the inverse of transfer function, we find all headers at the input that can generate something at the output. This is called the *range inverse* of a transfer function.

2.4 Limitations

Transfer functions can perfectly capture the forwarding behavior of stateless devices, because the forwarding behavior is reflected entirely by their forwarding state. However, stateful devices, whose behavior changes based on external states (such as history of packets or time characteristics of flows) cannot be modeled accurately, because those external states are inaccessible to HSA. In fact, the header and port of a single packet are the only state variables used in HSA. A NAT box is an example of a stateful device, and as we have seen previously, we need to sacrifice accuracy to model

it using a transfer function. However, in principle, HSA can be extended by adding more state variables to the model. For example, beyond (h, p) , one can add a list of previous packets observed by a particular network box to HSA.

2.5 Related Work

The geometric view of packet headers in HSA is a generalization of the geometric approach to packet classification [29], in which packets are modeled based on their k header fields as points in a k -dimensional space. In HSA, instead of pre-specified fields, we look at each bit as an independent dimension. This allows HSA to be protocol-independent and work with emerging protocols and arbitrary field formats. Also, the notion of a transfer function in HSA is similar to ASE mapping defined in axiomatic routing [21], where the authors develop tools to analyze a variety of protocols. While the goals of these works (packet classification and protocol verification) are different from ours (network testing and debugging), the similarity in models is an indicator of more broader applicability of HSA abstractions.

2.6 Summary

In this chapter, I introduced the header space analysis (HSA) framework. HSA defines the following key concepts:

- *Header Space*: Each packet, based on its header bits, is modeled as a point in a $\{0, 1\}^L$ space called the header space.
- *Box Transfer Function*: The forwarding behavior of networking boxes is captured by a transfer function: $T(h, p) : (h, p) \rightarrow \{(h_1, p_1), (h_2, p_2), \dots\}$.
- *Topology Transfer Function*: The topology of a network is modeled using a topology transfer function, $\Gamma(h, p)$. For each unidirectional link from P_a to P_b , $\Gamma(h, P_a) = (h, P_b)$.

HSA also introduces a set algebra to find the intersection, union, complementation and difference of header space regions. In this chapter, we reviewed the transfer function of some of the widely used networking boxes and learned that while the transfer function is perfect at capturing the behavior of stateless boxes, it cannot accurately model the stateful network devices.

Chapter 3

Network Verification with Header Space Analysis

In this chapter, I will use the Header Space Analysis (HSA) framework to develop techniques and tools for network verification. In particular, I will describe algorithms for finding reachability between two hosts (Section 3.1), detecting forwarding loops (Section 3.2) and checking isolation of network slices (Section 3.3). We will see how these basic checks can be used to verify more complicated policies and invariants such as network black-hole freedom, isolation of flow paths, and maximum hop count constraint on flows. I will also describe an optimized implementation of the header space analysis techniques in a library called *Hassel* (Section 3.4), which enables fast and simple implementation of these verification checks. Finally, I will review how these checks may be applied in practical contexts (Section 3.5). The goal of this chapter is to demonstrate the power of HSA abstractions and to show how they can immediately lead to design of simple and practical verification algorithms for networks.

3.1 Finding Reachability

One of the most important checks in networks is determining the reachability of end hosts. This means answering the question of whether the end hosts can communicate

and finding the set of all reachable packet headers. More formally, given two end hosts, a and b , we want to answer the question of “*which packets from host a can reach host b ?*”

3.1.1 Reachability Algorithm

Xie et. al. [52] analyze reachability by tracing which of all possible packet headers at a source can reach a destination. We follow a similar approach but generalize to arbitrary protocols. Using HSA, we consider the space of all headers leaving the source, and then we track this space as it is transformed by each successive networking box along the path (or paths) to the destination. At the destination, if no header space remains, the two hosts cannot communicate. Otherwise, we trace the remained header spaces backward (using the inverse of transfer functions at each step) to find the set of headers that the source can send to reach the destination.

More formally, we define the reachability function R between a and b as

$$R_{a \rightarrow b} = \bigcup_{a \rightarrow b \text{ paths}} \{T_n(\Gamma(T_{n-1}(\dots(\Gamma(T_1(h, p))\dots)))\},$$

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

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

The *range* of $R_{a \rightarrow b}$ is the set of headers that can reach b from a . Notice that these headers are *seen at b* and not necessarily headers transmitted by a because headers may change in transit. We can find which packet headers can leave a and reach b by computing the range inverse. If header $h \in \mathcal{H}$ reached b along the $a \rightarrow S_1 \rightarrow \dots \rightarrow S_{n-1} \rightarrow S_n \rightarrow b$ path, then the original header sent by a is

$$h_a = T_1^{-1}(\Gamma(\dots(T_{n-1}^{-1}(\Gamma(T_n^{-1}(h, b))\dots))),$$

using the fact that $\Gamma = \Gamma^{-1}$.


```

Find_Reachability_Range( $a$  ,  $b$ )
 $result \leftarrow []$ 
 $r \leftarrow \{header : \bar{x}, port : a, history : []\}$ 
 $Q \leftarrow [r]$ 
while  $Q.size() > 0$  do
   $r \leftarrow Q.pop\_front()$ 
   $temp \leftarrow \Gamma(\Psi(r.h, r.p))$ 
  for ( $h, p$ ) in  $temp$  do
     $s \leftarrow \{header : h, port : p, history : r.history.copy().append(h, p)\}$ 
    if  $p == b$  then
       $result.append(s)$ 
    else if  $(* , p) \in r.history$  then
      Loop Detected.
    else
       $Q.push\_back(s)$ 
    end if
  end for
end while
return  $result$ 

```

Figure 3.1: DFS algorithm for finding the range of the reachability function between a and b .

We can efficiently compute the range and range inverse of the reachability function through a breadth-first-search (BFS) or depth-first-search (DFS) on the graph of network topology, taking into account the transformation at each node. To find the range of the reachability function—i.e., the set of reachable headers at the destination—we start by applying an *all-wildcard* header to the transfer function of S_1 , which is the box directly connoted to a : $T_1(X, a)$. The all-wildcard header represents the set of all packet headers that host a can generate, and the output of $T_1(X, a)$ shows all packet from host a that will be passed through S_1 . Then, we apply the resulting headers to the topology transfer function to find the next hop boxes and apply the headers to the transfer function of those boxes. By repeating this process, we can find all the destinations that are reachable from a (including b) and all of the headers that are reachable to those destinations. Figure 3.1 shows the formal description of the algorithm for finding the range of the reachability function.

Once we have computed the range of the reachability function together with the path history of each reachable header, we can find its range inverse or the set of reachable packet headers at the source. To do so, we apply each reachable header at the destination to the inverse of transfer functions along the traversed path and find the headers at the source. Note that in the algorithm described in Figure 3.1, the *history* contains the path information.

To illustrate the process more clearly, we perform the reachability analysis for the small toy example network in Figure 3.2. The transfer function of each box is shown in Figure 3.3. To keep things simple, we only use 8-bit headers; because we cannot easily depict eight dimensions, we represent the first 4 bits of the header on the x -axis and the last 4 bits on the y -axis. Note that in this example, A and C are miniature models of IP routers, B is a firewall, D is a simplified network address translator (NAT) box and E behaves like an Ethernet switch.

Figure 3.3 shows how the network boxes transform the all-wildcard header at the source along each path to the destination. By repeatedly applying the output of each transfer function to the input of the next transfer function in each path, we can find the range of the reachability function: $10010x10 \cup 01011x10$. Figure 3.4 shows the progress of the reachability algorithm (Figure 3.1) as the search spreads through the network. We refer to this graph as the *propagation graph* as it shows the propagation of a flow in the network. Each node in the propagation graph shows the set of packet headers, \mathbf{Hdr} , that reached a \mathbf{Port} and the set of ports visited previously on the path, $\mathbf{History}$. In other words, the nodes correspond with the elements put in the queue Q in the pseudo code of Figure 3.1. Each child node in the propagation graph is the result of applying $(\mathbf{Hdr}, \mathbf{Port})$ of the parent node to the network and topology transfer functions (i.e., $\Gamma(\Psi())$).

If instead we have composed the transfer functions along the two paths, the reachability function from a to b would become:

$$R_{a \rightarrow b}(h, p) = \begin{cases} \{(h, E_2)\} & \text{if } h=10010x10, p = A_0 \\ \{((h \& 00011111) | 01000000, E_2)\} & \text{if } h=10011x10, p = A_0 \end{cases}.$$

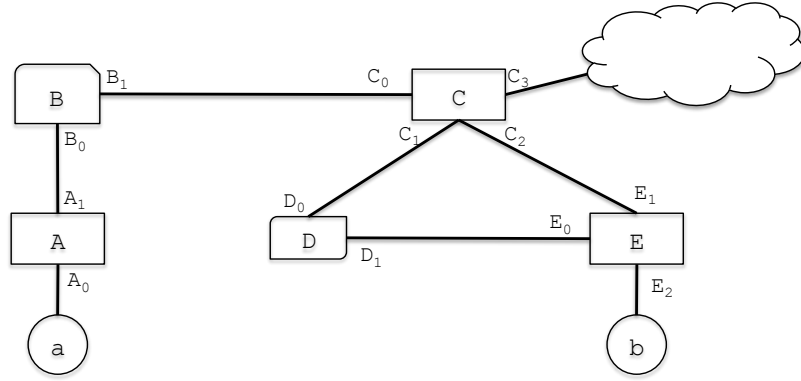


Figure 3.2: A toy network topology used for computing reachability from a to b .

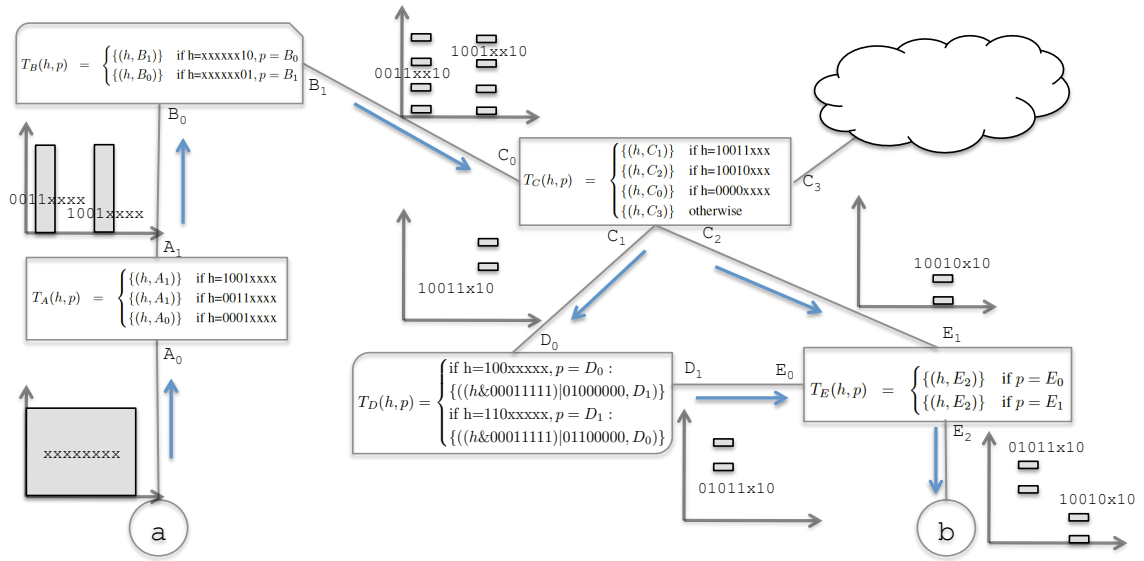


Figure 3.3: Computing reachability from a to b in a toy example network.

For simplicity, we assume a header length of 8 and show the first 4 bits on the x-axis and the last 4 bits on the y-axis. We show the range (output) of each transfer function composition along the paths that connect a to b . At the end, the packet headers that b will see from a are $01011x10 \cup 10010x10$.

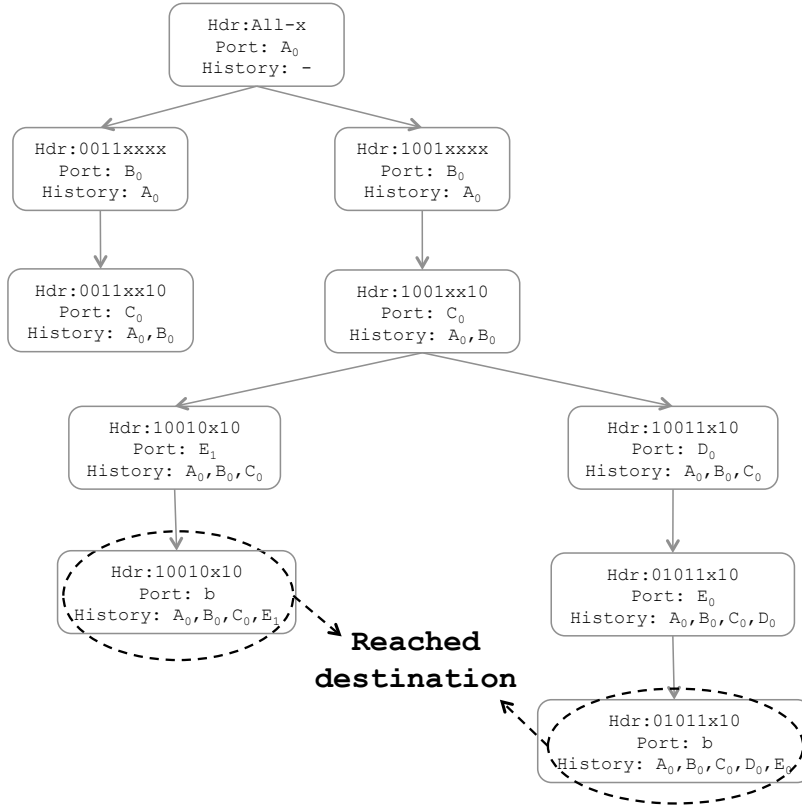


Figure 3.4: Propagation graph when finding reachability from a to b in the network of figure 3.2.

The range of $R_{a \rightarrow b}$ is $10010x10 \cup 01011x10$ as expected.

To find the set of headers that a can send to b , we can either compute

$$\begin{aligned}
 & T_A^{-1}(\Gamma(T_B^{-1}(\Gamma(T_C^{-1}(\Gamma(T_E^{-1}(10010x10, E_2))))))) \\
 & \cup \\
 & T_A^{-1}(\Gamma(T_B^{-1}(\Gamma(T_C^{-1}(\Gamma(T_D^{-1}(\Gamma(T_E^{-1}(01011x10, E_2))))))))),
 \end{aligned}$$

or compute the range inverse of $R_{a \rightarrow b}$, which both will be $10010x10 \cup 10011x10$.

3.1.2 Checking Path Predicates

A path predicate is a constraint on the path of some flows in the network. For example, the requirement that all http traffic from IP subnet A should pass through a middle box m is a path predicate on http traffic from subnet A . Reachability analysis enables us to check predicates on the path of flows in the network. The reachability algorithm presented above finds all of the flows that can reach from a source to a destination along with the path of those flows (*history* in the algorithm of Figure 3.1 captures the path). We can use this information to check more complicated predicates. Below, we review a few examples:

- *Host a can not talk to host b .* Compute reachability from a to b and verify that the reachable set is empty.
- *Host a can only talk to host b via middlebox m .* Compute reachability from a to b and verify that the path of all reachable flows pass through m .
- *Http and https traffic from host a to host b do not share the same path.* Compute reachability from a to b and verify that the path of http and https headers are different.
- *Traffic from host a to host b doesn't go through more than three hops.* Compute reachability from a to b and verify that the path length of all reachable flows is less than three.

As we will see in Section 3.2, detecting forwarding loops also requires computing reachability. However, we need to do more work in order to find the repetition of a loop.

3.1.3 Complexity of Finding Reachability

In the reachability algorithm presented in Figure 3.1, as we push the all-wildcard header toward the destination, the transfer function rules divide the input header space into smaller pieces. If the input header space consists of a union of R_1 wildcard expressions and the transfer function has R_2 rules, then the output can be a

header space with $O(R_1R_2)$ wildcard expressions. This is because each input wildcard expression will match on every rule, generating R_2 output wildcard expressions per input wildcard expression. For example, if the first router has 1000 destination IP prefixes, then the result of applying the all-wildcard flow to the first router can be a header space consisting of 1000 wildcard expressions. If this is applied to a second box with 100 ACLs to reject traffic from certain sources, then the output of the second switch can have $1000 \times 100 = 100,000$ wildcard expressions. However, this is a worst-case scenario and happens only if the match expressions of the transfer function are orthogonal to the input wildcard expressions, as in the example above.

In a real network whose purpose is to provide connectivity, there are certain bits, such as the VLAN tag, MPLS tag or destination IP address that determine the routing of packets in the core. Packet filtering, which is based on a different set of bits, such as the TCP port number or source IP address, happens at the edges. In fact, the rules at the core often aggregate flows into larger flows and do not divide them into smaller sub-flows. Therefore as R_1 wildcard expressions pass through R_2 rules at the core, they only match on a few of those rules, producing only cR_1 output wildcard expressions where $c \ll R_2$ is a small constant called the fragmentation factor. We call this, the *linear fragmentation* assumption.

Under the linear fragmentation assumption, the complexity of the reachability algorithm described in this section is $O(dR^2)$, where d is the maximum diameter of the network and R is the maximum number of rules in a network box. This is because at each hop along the propagation paths, we need to apply $O(R)$ input wildcard expressions to $O(R)$ rules in the boxes, which requires $O(R^2)$ computations. However, this generates only $O(R)$ output wildcard expressions (see Figure 3.5), which will then go through the next hop transfer functions. There are at most d boxes in any path; therefore, the overall computation cost will be $O(dR^2)$.

To check the validity of this assumption in real networks, we ran a test using the transfer functions of Stanford university's backbone network (see Section 3.5.1 for more details about the Stanford network). The test pushed an all-wildcard flow from one of the input ports at the edge and tracked the generated flows as they went through the box transfer functions. It then counted the total number of wildcard

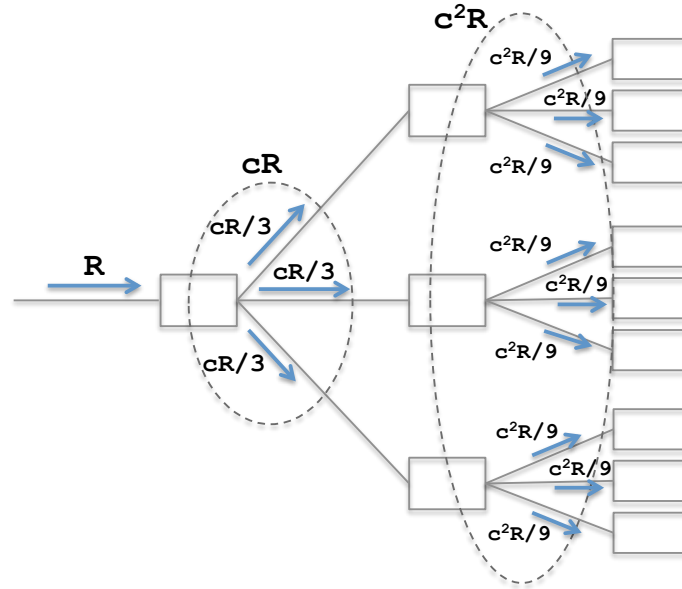


Figure 3.5: Fragmentation of the header space as it passes through the network. The numbers on the arrows show the number of wildcard expressions required to describe each header space.

expressions generated after each hop of forwarding across all the propagation paths. Figure 3.6 shows the total number of wildcard expressions produced versus the hop count for 12 different input ports. The figure suggests that after the first hop, the number of wildcard expressions increased by a small constant ($c < 10$), and that constant will be less than 1 during the rest of the path as the flows start reaching their destinations.

3.2 Detecting Forwarding Loops

A forwarding loop occurs when a packet returns to a port that it visited earlier. We are interested in detecting and avoiding forwarding loops as they cause forwarding storms and waste network resources. Using the header space analysis framework, we can find all of the forwarding loops in a given network, determine all packet headers that can loop and specify how many times each packet will loop.

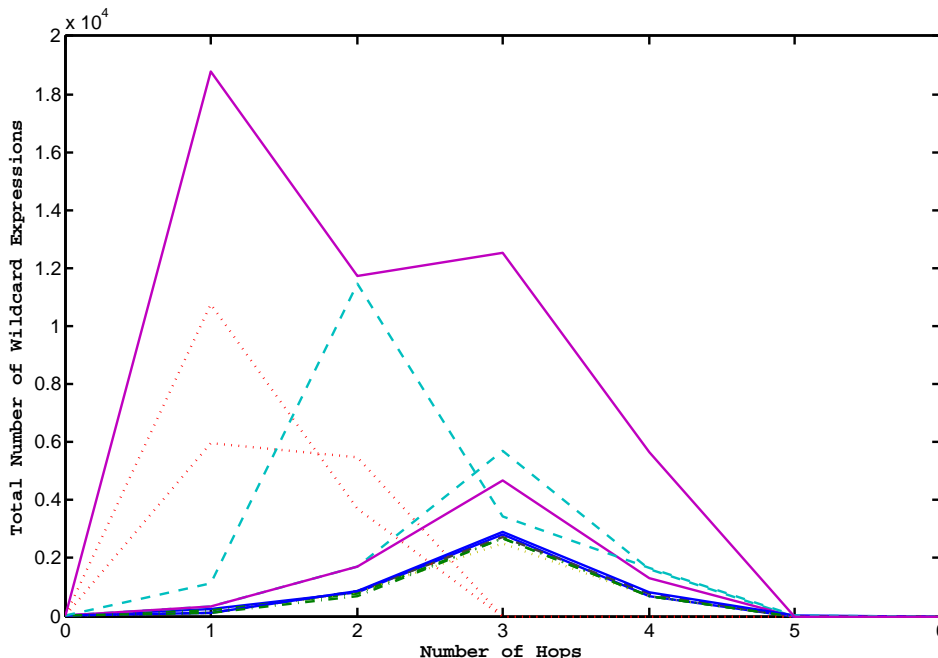


Figure 3.6: Total number of wildcard expressions generated in the process of computing reachability, counted at each propagation hop.

The graph is obtained by pushing 12 all-wildcarded flows from the edges to the core of the Stanford backbone network and counting the total number of wildcard expressions generated after each hop of forwarding across all of the paths.

The first step to detect a forwarding loop is to find out if there is any packet that can visit the same port twice. We call these loops *generic*, as packets may go through the loop only once before being dropped. Once we have detected the generic loops, we can run an extra test to find the repetition of the loops—i.e., the number of times that packets loop before being dropped.

3.2.1 Finding Generic loops

Given a network and its topology transfer function, we can detect all forwarding loops by injecting an all-wildcard test flow from *each port* in the network that can be part of a graph loop and track each resulting flow until:

- (Case 1) It leaves the network or is dropped;

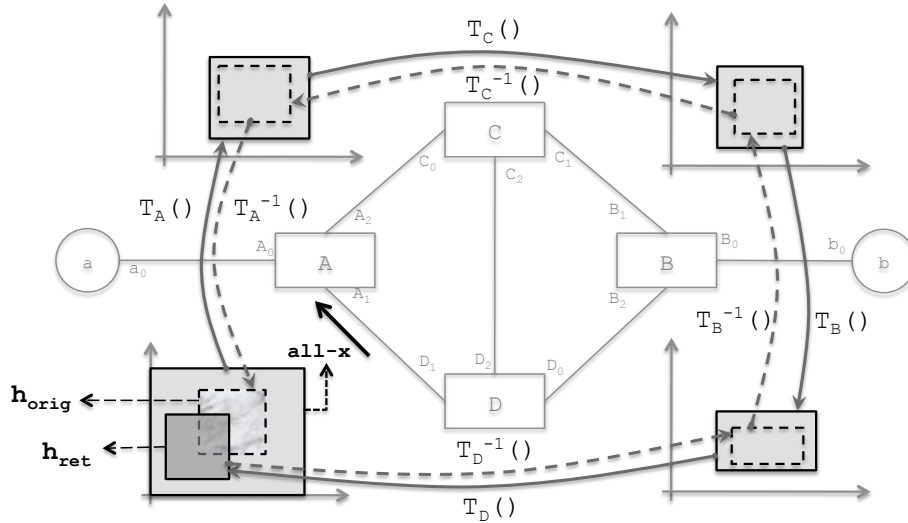


Figure 3.7: An example of running the complete loop detection check. The solid lines show the changes to the all-wildcarded test packet injected from A_1 as it propagates in the network. Once a generic loop is detected and h_{ret} is found, the inverse of transfer functions in the loop are used to find the loop-generating header space, h_{orig} . The dashed lines show the process of finding h_{orig} .

- (Case 2) It returns to a port already visited (P_{ret}); or
- (Case 3) It returns to the port¹ from which it was injected (P_{inj}).

Only in case 3 (i.e., when the packet comes back to its injection port) do we report a loop. Because we repeat the same procedure starting at every port, we will detect the loops detected by case 2 when we inject the test flow from P_{ret} . Ignoring case 2 avoids reporting the same loop twice.

This algorithm is similar to finding reachability except that the source and destination ports are the same. Therefore, we can implement it using a *breadth first search* similar to the one in Figure 3.1. As an example, consider the network in Figure 3.7 where an all-wildcard test flow is injected from port A_1 . Figure 3.8 is the corresponding propagation graph explored through the depth-first search procedure. As before, each child node in the propagation graph is the result of applying $(Hdr, Port)$ of

¹While we could define a loop as a packet returning to a *node* visited earlier, using ports helps detect infinite loops.

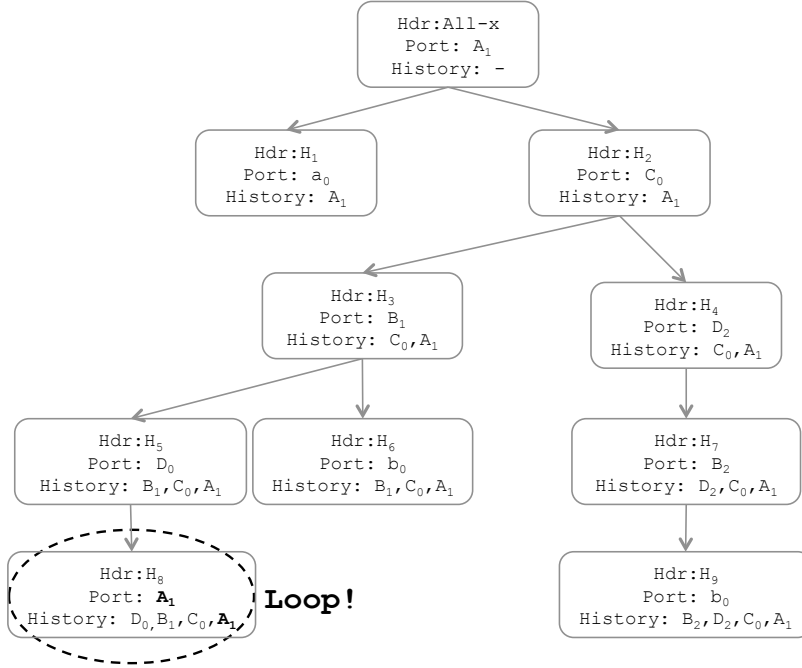


Figure 3.8: Example of a propagation graph for a test packet injected from port A_1 in the network of figure 3.7.

the parent node to the network and topology transfer functions. For example, in Figure 3.8:

$$\{(H_3, B_1), (H_4, D_2)\} = \Gamma(\Psi(H_2, C_0)).$$

At any point during the expansion of the propagation graph, if **Port** appears in the **History** list, we terminate that branch of the tree. If **Port** is the first element of **History**, we have detected a loop.

3.2.2 Finding Repetition of Forwarding Loops

Not all packets in a generic loop will loop the same number of times. Because the loop transfer functions may rewrite headers, some packets may get dropped in the next round, and some may loop indefinitely. Our goal here is to find out how many times each packet will loop. This is important because a loop that repeats only once is not as harmful as an infinite loop.

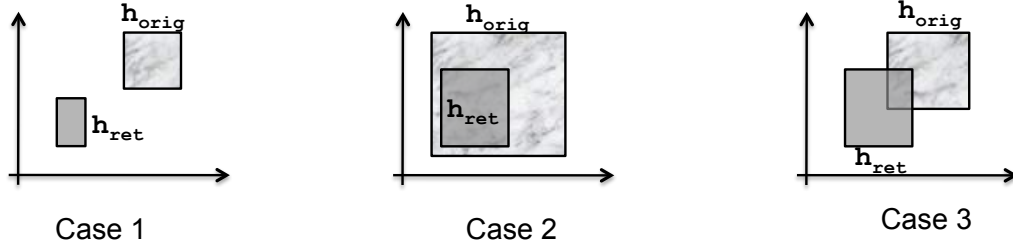


Figure 3.9: Relative position of h_{ret} and h_{orig} and their effect on loop repetition. In case 1, the loop only happens once. Case 2 represents an infinite loop. Case 3 is where different points in h_{orig} loop different number of times.

We will explain the process of finding the loop repetition through the example in Figure 3.7. Here, h_{ret} denotes the part of the header space that comes back to the injection port, A_1 . We can use the inverse of loop transfer functions to find the loop-generating header space, h_{orig} :

$$h_{orig} = \Phi^{-1}(\Phi^{-1}(\Phi^{-1}(\Phi^{-1}(h_{ret}, A_1)))) \Big|_{p=A_1},$$

where $\Phi(\cdot) = \Gamma(\Psi(\cdot))$. h_{orig} represents the set of all headers that can loop through port A_1 . Figure 3.7 demonstrates the process of finding h_{orig} symbolically.

h_{ret} and h_{orig} relate in one of three ways, as depicted in Figure 3.9:

1. $h_{ret} \cap h_{orig} = \phi$: In this case, all of the headers in h_{orig} will loop *only once*. This is because every point in h_{orig} is mapped to a point in h_{ret} , which is outside the loop-generating region, h_{orig} . Therefore, these headers cannot loop in the next round.
2. $h_{ret} \subseteq h_{orig}$: In this case, the loop is infinite for all of the headers in h_{orig} because every point in h_{orig} is mapped by the transfer function of the loop to *at least one* point in h_{ret} . Because h_{ret} is completely within h_{orig} , those points will loop again and will be mapped to other point(s) in h_{ret} . This process will never stop, as points cannot escape from this mapping; therefore, the loop continues indefinitely.

3. *Neither of the above:* In this case, different points within h_{orig} will loop a different number of times (between one and infinity). First, note that $h_{ret} - h_{orig}$ completely satisfies case 1's condition and therefore cannot loop again. However, we must examine $h_{ret} \cap h_{orig}$. Thus, we define $h'_{ret} := h_{ret} \cap h_{orig}$ and calculate the new loop-generating header space, h'_{orig} which generates h'_{ret} . We repeat the check for h'_{orig} and h'_{ret} , taking into account that they have already looped once. For example if $h'_{ret} \cap h'_{orig} = \phi$, h'_{orig} headers will loop twice, while $h_{orig} - h'_{orig}$ will loop only once. By repeating this check, we can determine the repetition of each header involved in the loop. Figure 3.10 formally defines the algorithm for checking loop repetition.

The forwarding loop that is taken care of by IP TTL is an example of the third case. For a loop of length n , $h_{ret} = ttl$ for $0 < ttl < 256 - n$ and $h_{orig} = ttl$ for $n < ttl < 256$. In the first iteration, we find that packets with $n < ttl \leq 2n$ will loop once. In the next round, $h'_{ret} := h_{ret} \cap h_{orig} = ttl$ for $n < ttl < 256 - n$ and $h'_{orig} := ttl$ for $2n < ttl < 256$, and we find that packets with $2n < ttl \leq 3n$ will loop twice. In subsequent iterations, h_{ret} and h_{orig} shrink by n per iteration.

3.2.3 Dealing with Tortuous Forwarding Loops

Tortuous loops are forwarding loops where a packet passes through multiple loops before coming back to a first loop. Figure 3.11 shows an example of a tortuous loop consisting of two loops. We can generalize the loop repetition-counting algorithm as follows: first, detect all of the loops that have originated from a single port, and for each, find h_{ret_i} and h_{orig_i} . Define

$$h_{ret} := \bigcup_{i=1}^n h_{ret_i}, \quad h_{orig} := \bigcup_{i=1}^n h_{orig_i},$$

and then perform the same check on h_{ret} and h_{orig} to decide whether the combination of loops is infinite or not. By combining all of the h_{orig} and h_{ret} regions, we have considered the effect of all of the loops simultaneously. This enables us to automatically

```

Find_Loop_Repetition( $h_{ret}$ ,  $h_{orig}$ ,  $p_{inj}$ ,  $T_{loop}$ ,  $count = 0$ )
#  $h_{ret}$ : Returned header space to the injection port ( $p_{inj}$ )
#  $h_{orig}$ : Loop-generating header space
#  $p_{inj}$ : injection port
#  $T_{loop}$ : Loop transfer function—composition of transfer functions in the loop
# @Return: a set of ( $h$ ,  $count$ ) where  $h$  is a header wildcard expression in  $h_{orig}$  and
 $count$  is its loop repetition count
if  $h_{ret} \cap h_{orig} = \phi$  then
  return  $\{(h_{orig}, count + 1)\}$ 
else if  $h_{ret} \subseteq h_{orig}$  then
  return  $\{(h_{orig}, \infty)\}$ 
else
   $h'_{ret} \leftarrow h_{ret} \cap h_{orig}$ 
   $h'_{orig} \leftarrow T_{loop}^{-1}(h'_{ret}, p_{inj}) \Big|_{p=p_{inj}}$ 
  return  $\{(h_{orig} - h'_{orig}, count + 1)\} \cup$ 
    Find_Loop_Repetition( $h'_{ret}$ ,  $h'_{orig}$ ,  $p_{inj}$ ,  $T_{loop}$ ,  $count + 1$ )
end if

```

Figure 3.10: A recursive algorithm for finding loop repetition of each header involved in a loop.

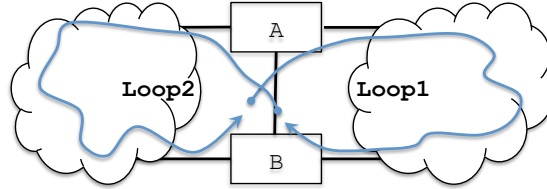


Figure 3.11: Example of a tortuous loop consisting of multiple loops.

evaluate any combination of single loops that results in a complete tortuous loop as we have considered h_{orig} of all of the single loops simultaneously.

3.2.4 Complexity of Finding Loops

The algorithm for detecting forwarding loops is very similar to finding reachability. The only difference is that we repeat the check on P ports that are part of the network graph loops. Therefore, the complexity of loop detection check is $O(dPR^2)$. However,

there is no such polynomial bound for finding the repetition of loops. Consider this adversarial example: a header consisting of L bits, where all of the L bits act as TTL. If this packet loops in a two-node network, finding the loop repetition requires 2^{L-1} iterations.

3.3 Checking Isolation of Network Slices

Slicing a network is a way to share network resources among multiple entities. For example, two different banks may have branches in a financial center and share the network equipments in the building. In order to make network slicing secure and practical, the slices should be isolated from one another. This is like virtualization of computer hardware, where different virtual machines are isolated from one another and are given the illusion of full control over the hardware resources. Also, network operators often limit the communication between different groups of hosts (or users) by putting them in different isolated slices. A common requirement for isolated slices is that traffic stay within its slice and not leak to another slice.

Traditionally, network slicing is done by VLANs, where each slice is defined by a unique VLAN ID. However, for software defined networks (SDNs), FlowVisor [48] can create dynamic slices based on any set of header fields (e.g., all packets with source IP address 192.168.1.0/24 may define a slice). Similar to [16], we define a slice as:

- a topology consisting of switches, ports and links.
- a collection of predicates on packets belonging to the slice, one on each ingress port in the slice topology.

Therefore, a network slice is defined by a region of network space, \mathcal{N} , in header space language.

Two slices of the network are isolated if:

1. The network space representation of the two slices are disjointed. This means that no packet belongs to and is control by both slices simultaneously.

2. The packets in one slice cannot leak to the other slice after being forwarded to the next hop.

The first isolation condition should be checked when a slice is created. The second condition should be checked every time a new forwarding rule is installed in the network.

3.3.1 Checking Disjointness of Slice Definitions

Assume we have two network slices with the following network space definitions:

$$N_a = \{(\alpha_i, p_i) \mid p_i \in \mathcal{S}\} \quad , \quad N_b = \{(\beta_i, p_i) \mid p_i \in \mathcal{S}\},$$

where α_i and β_i are headers belonging to slice a and b , respectively, on each port, p_i , in the network.

If the two slices do not overlap, they have no header space in common on any common port, i.e., $\alpha_i \cap \beta_i = \phi$, for all i . If they intersect, we can determine precisely where (which links) and how (which headers) by finding their intersection:

$$N_a \cap N_b = \{(\alpha_i \cap \beta_i, p_i) \mid p_i \in N_a \& p_i \in N_b\}.$$

A set intersection could, for example, be used to statically verify that communication is allowed at one layer or is allowed with one protocol but not with another. A simple check for overlap is extremely useful in any slicing environment (e.g., VLANs or FlowVisor) to check for violations prior to creation of new slices. The test can flag violations, or it could be used to create one slice to monitor another.

3.3.2 Detecting Packet Leakage

Even if two slices do not overlap anywhere, packets can still leak from one slice to another when headers are rewritten. We can use another algorithm to check whether packets can leak. If there is leakage, the algorithm finds the set of offending (*header, port*) pairs.

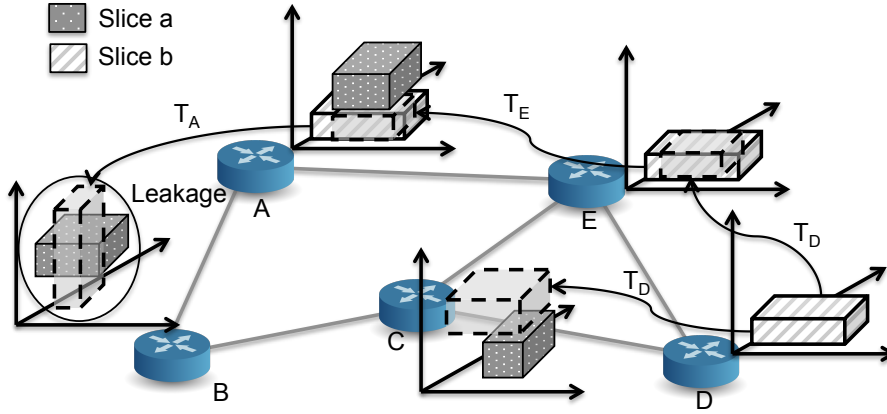


Figure 3.12: Detecting slice leakage by finding the image of slice a under network transformation and intersecting it with slice b .

Assume that slices a and b have network space definition N_a and N_b , as above. Leakage occurs when a packet in a given slice at any switch-port can be rewritten and forwarded to fall into the network space of another slice. If packets cannot leak at any switch-port, then they cannot leak anywhere. Therefore, to check if packets will not leak from slice a to slice b , we apply the network space definition of slice a to the network transfer function. We refer to this as the *image* of slice a under network transformation. This image shows all possible packets in slice a after they are forwarded by network boxes. If this image does not have any intersection with the network space definition of slice b , then packets cannot leak from a to b . More formally, packets will not leak from slice a to slice b if and only if $\Gamma(\Psi(N_a)) \cap N_b = \phi$. Figure 3.12 graphically represents this check. Note that the leakage test should be repeated for every new forwarding rule added to the network.

3.3.3 Complexity of Checking Slice Isolation

To check for the isolation of two network slices, we need to find the intersection of one slice's network space definition with all of the other ones. If there are N slices in the network, each described by $O(W)$ wildcard expressions on each of the P ports in the slice topology, then finding the intersection has $O(NW^2)$ complexity, as we

need to find intersection of $O(W)$ wildcard expressions with $O(NW)$ other wildcard expressions. If the definition of a slice is different on each port of the slice, then we need to repeat the computation once for each port, and hence, the complexity will be $O(PNW^2)$.

To check if a new rule added to the network will result in packets leaking from slice a to other slices, we need to apply the network space definition of slice a to the new transformation rule. This has complexity $O(W)$ as slice a is described by $O(W)$ wildcard expressions on each port, each of which needs to be transformed by the new rule. Then, we need to intersect the result of the transformation with all the existing slices. This is similar to slice isolation check and will require $O(NW \times W) = O(NW^2)$ work. This is the run time for incrementally performing this test on every new rule insertion. If we want to perform this test for all of the rules in the network from scratch, then we need to repeat it for all of the $O(R)$ rules in the transfer function of $O(k)$ boxes controlled by the slice. Therefore, the run time would be $O(kRNW^2)$.

3.4 Hassel: the Header Space Library

The Header Space Library or *Hassel* is a set of tools written in Python 2.7 and C that implement the header space framework and the applications described in this chapter. The source code is available in [24]. Hassel's basic building block is a header space class, which implements HSA set operations (Section 2.3.1). Internally, it stores a union of wildcard expressions that describes the header space object. In Hassel, *transfer function objects* that implement network transfer functions are configured by a set of rules. Given a header space object and port, a transfer function generates a list of output header space objects and ports. Transfer functions can be built from standard rules (i.e., by matching on an input port and wildcard expression) or from custom rules supplied by the programmer through function pointers. Hassel can automatically compute the inverse of a transfer function for standard rules. Hassel also includes a Cisco IOS parser and a Juniper Junos parser that parses router configurations and command line (CLI) outputs and generates a transfer function object that models the static behavior of the router. While making a transfer function, the

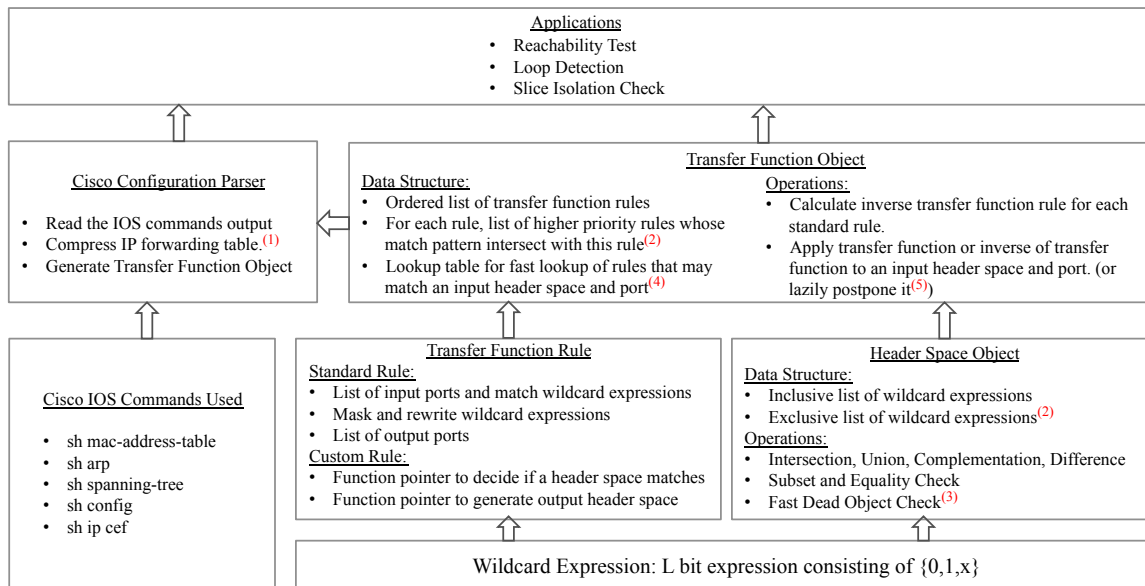


Figure 3.13: Architecture of Hassel—the Header Space Library.

parser keeps a mapping from each transfer function rule to the CLI line numbers that generate the rule. This will come handy when we want to investigate the root cause of problems.

Figure 3.13 shows the block diagram of Hassel. The parser uses the MAC address table, the ARP table, the spanning tree, the IP forwarding table and the router configuration of Cisco or Juniper boxes to generate the transfer function. The resulting transfer function object is then used by applications such as loop detection.

3.4.1 Algorithmic Optimizations

Our implementation employs five key optimizations marked with superscript indices in Figure 3.13 that are keyed to the rows in Table 3.1. Table 3.1 reports the impact of disabling each optimization, in turn, when analyzing Stanford’s backbone network (see Section 3.5.1 for more information about the Stanford network). For example, loop detection for a single port with all optimizations enabled took 11 seconds, however, disabling IP compression increased running time by 19x and disabling lazy subtraction inflated running time by 400x. Because the optimizations are orthogonal,

Disabled Optimization	T.F. Generation	Reachability Test	Loop Test
None	160s	12s	11s
(1) IP Table Compression	10.5x	15x	19x
(2) Lazy Subtraction	1x	>400x	>400x
(3) Dead Object Deletion	1x	8x	11x
(4) Lookup-based Search	0.9x	2x	2x
(5) Lazy T.F. evaluation	1x	1.2x	1.2x

Table 3.1: Impact of optimization techniques on the run time of the reachability and loop detection algorithms.

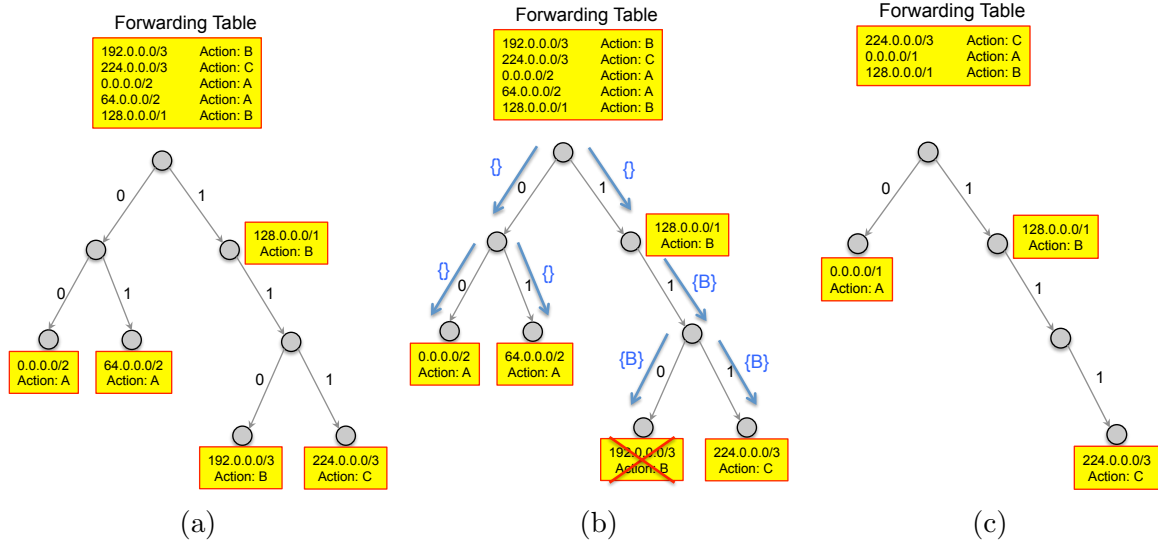


Figure 3.14: Compressing IP forwarding table using a binary tree representation.

the overall effect of all optimizations is around 10,000x, transforming Hassel from a prototype to a tool.

IP table compression: We used IP forwarding table compression techniques in [9] to reduce the number of transfer function rules and to make the run time of Hassel faster. The compression works as follows:

- We store all of the IP forwarding rules in a binary tree. In this tree, the root corresponds with the left-most bit of the IP address, the children of the root correspond with the second bit of the IP address, and so on. The left edge out

of each node matches on 0, and the right edge matches on 1. By following the path from the root to each node in the tree, we can label the node with the bit pattern that it represents. Each IP forwarding rule is stored at the node whose label matches the non-wildcard bits of the IP address. Figure 3.14(a) shows the binary tree representation of a forwarding table consisting of five IP forwarding rules.

- Once we have stored all of the rules in the binary tree, we can compress them. We start from the root of the tree and propagate the action of the rule stored in the root (if any) downward. When reaching any rule stored in the tree, we compare its action against the propagating action. If they are the same, we safely remove that rule from the tree and the table because it is covered by a more general rule. If the actions do not match, we stop propagating that action down that branch and instead propagate the action of the newly encountered rule. Figure 3.14(b) shows this process.
- Finally, when visiting each node during the propagation, we examine the left and right children. If the two children store rules with the same action, we merge them into one rule and store it in their parent node. In our example, $0.0.0.0/2$ and $64.0.0.0/2$ are merged into one rule ($0.0.0.0/1$), resulting in the final graph and forwarding table in Figure 3.14(c).

Lazy subtraction: Forwarding rules in a transfer function are sorted according to their priority for processing packets. For example, if a router has two destination IP addresses, $10.1.1.x$ and $10.1.x.x$, and uses the longest prefix match, only packets matching on $10.1.x.x - 10.1.1.x$ will be processed by the second rule. Expressing these matching headers as a union of wildcard expressions will require eight wildcard expressions. To avoid this, the notion of a header space object is expanded to accept a union of wildcard expressions *minus a union of wildcard expressions*: $\cup\{w_i\} - \cup\{w_j\}$. Then, when we want to process the input header by the $10.1.x.x$ rule, we simply *subtract* from the final result the headers processed by the first rule. Lazy subtraction allows for the postponement of the expansion of terms during intermediate steps,

as this expansion is only done at the end. As the table suggests, performance is dramatically improved.

Dead object deletion: At intermediate steps, header space objects often evaluate to empty and should be removed. Lazy subtraction masks such empty objects, as it postpones the evaluation of the subtracted expressions. As a result, in our reachability or loop algorithm, we propagate an object that would have been evaluated to empty. Therefore, a quick test is added to detect empty header space objects without explicit subtraction. This check is based on a simple heuristic that is fast, doesn't have any false positives, and works well in practice but may not catch all empty objects. The simple trick is that $\cup\{w_i\} - \cup\{w_j\}$ is a dead object if for each w_i wildcard expression, there is a w_j , that is its superset.

Lookup-based search: To evaluate a transfer function on an input header space, we must find which transfer function rules match the input header space. The simple way to implement this is to do a linear search through the rules in the transfer function. We avoid an inefficient linear search via a lookup table. The key to the lookup table is a configurable set of bytes in the packet header, which defines the buckets in the lookup table. Each bucket stores all of the rules that match on those bits. Because the search key may have wildcard bits, the lookup key also contains all possible combinations of 0,1 and x.

Lazy evaluation of transfer function rules: The number of wildcard expressions required to describe the result of a reachability or loop detection test may grow as the cross-product of the rules in transfer functions. For example, if some boxes forward packets based on D destination addresses, while others filter packets based on S source IP addresses, the range of the composed transfer function will have $D \times S$ wildcard expressions. If two transfer functions are orthogonal (meaning that they process packets based on different sets of bits), Hassel uses commutativity of transfer functions to delay the computation of one set of rules until the end. To do so, a header space object can carry a list of unevaluated transfer function rules. These unevaluated rules may be applied to the header space object once it has reached the intended destination. By delaying the computation, we prevent the explosion of wildcard expressions at the intermediate steps. Figure 3.15 depicts the effect of

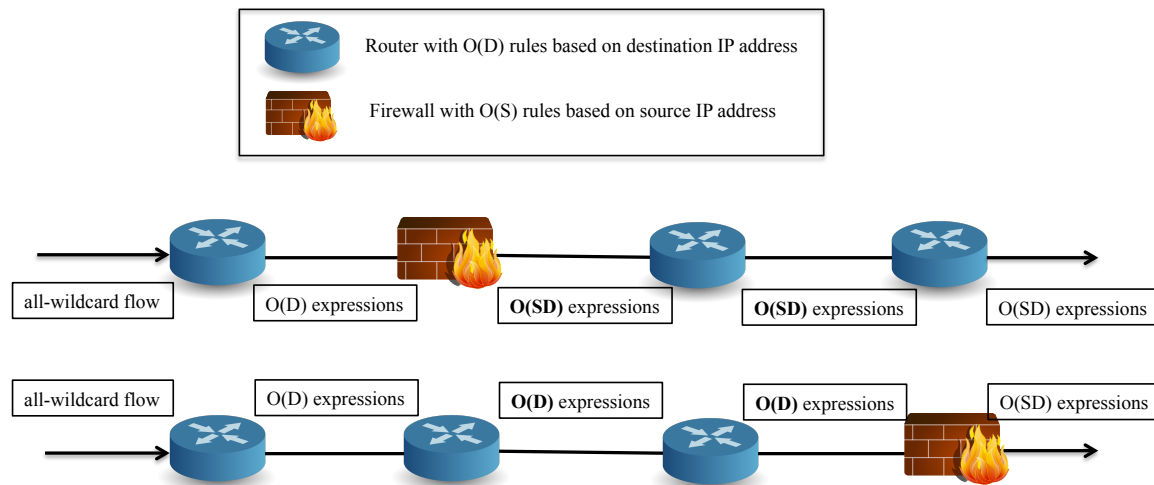


Figure 3.15: Lazy evaluation of transfer function rules: by switching the order of applying transfer function rules, the overall computation complexity is reduced.

changing the order of applying transfer function rules on the overall complexity of . The current implementation of Hassel may be configured with a list of header bits to be treated as the *primary* bits for forwarding (e.g., IP destination bits). Rules that process packets based on non-primary bits are lazily evaluated.

3.4.2 Implementation Optimizations

The implementation of Hassel in C is optimized for performance. To maximize performance, it deploys some memory optimization and parallelism techniques on top of the algorithmic optimizations discussed above.

Deferred memory allocation: When computing reachability or running a loop detection test, we need to intersect the input header space of a transfer function with the match wildcard expression of all of the rules in that transfer function. We observed that a vast majority of these intersections resulted in an empty header space object. Therefore, we deferred allocating a new header space object until the object was guaranteed to be non-empty. This reduced the number of memory allocations by 90%.

Wildcard expression reuse: Many rules are repeated in the transfer function objects of a given network. All of these rules share the same match wildcard expression. For example, in the Stanford backbone network, 84% of rules are duplicated more than once. To avoid unnecessarily enlarging the working set of Hassel, we stored a unique copy of each rule in memory.

Data structure compaction: Building on the previous optimization, we observed that our performance was limited by misses in the CPU cache. We thus packed the transfer function data into a dense binary file. This reduced the size of each rule by a factor of 4.

Parallelism: We added parallelism to the C-based Hassel by creating a thread for each transfer function. Intuitively, this means that each router processes header space objects independently of all other routers. Each transfer function (running in its separate thread) maintains a global “task” queue containing the set of header space objects to be processed by that transfer function. Once a transfer function finishes processing a header space, it finds the next transfer function to process its results and puts its results in the task queue of that thread.

Parallelism by itself and without the previous optimizations does not improve performance significantly, as the tasks are memory-bound rather than CPU-bound. However, once the working set is reduced in order to avoid too many cache misses, parallelism starts to show its benefits.

3.5 Evaluation

In this section, I first demonstrate the functionality of Hassel on Stanford university’s backbone network and report performance results of the reachability and loop detection algorithms on an enterprise network. Then, I benchmark the performance of the slice isolation test on random slices created on Stanford’s backbone network, which are similar to the existing VLAN slices. Finally, I showcase the applicability of Hassel to new protocols. All of these tests ran on a Macbook Pro, with Intel core i7, 2.66 Ghz quad core CPU and 4 GB of RAM.

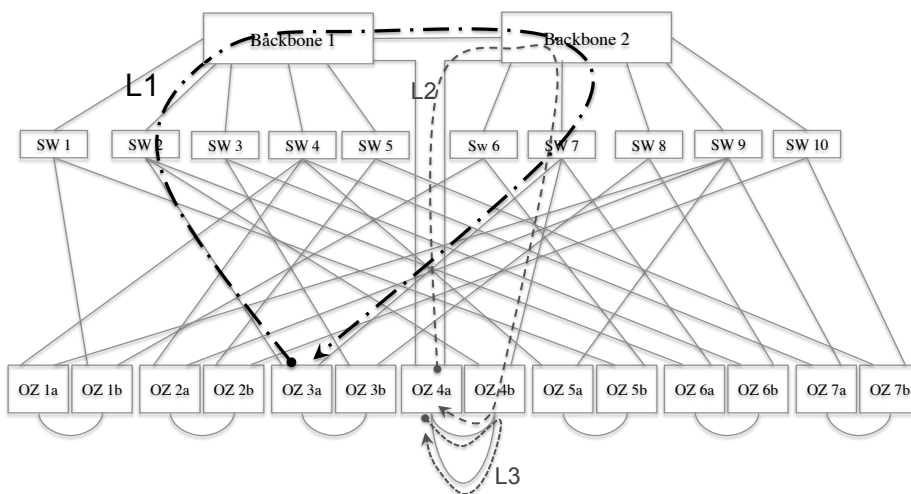


Figure 3.16: Topology of the backbone network of Stanford university and three types of loops detected using Hassel. Overall, we found 26 loops on 14 loop paths, 10 of which are infinite.

3.5.1 Verification of an Enterprise Network

We start by running Hassel on Stanford university’s backbone network. With a population of more than 15,000 students, 2,000 faculty, and five /16 IPv4 subnets, Stanford is a relatively large enterprise network. Figure 3.16 shows the network that connects departments and student dorms to the outside world. There are 14 operational zone (OZ) routers at the bottom of Figure 3.16 that are connected via 10 switches to two backbone routers that connect Stanford to the outside world. Overall, the network has more than 757,000 forwarding entries and 1,500 ACL rules.

The goals of this experiment is to demonstrate the utility of running Hassel checks and to measure Hassel’s performance in a production network. When generating transfer functions, the learned MAC addresses are intentionally removed from the model. As a result, packets will be flooded on the spanning tree rather than directly forwarded to the destination. This allowed us to discover problems that can be masked by learned MAC addresses but may surface when learned entries expire.

Checking for loops: The loop detection test was performed on the entire backbone network by injecting test packets from 30 ports. It took 151 seconds to compress

the forwarding table and to generate transfer functions. The python version of Hassel needs 560 seconds to run the loop detection test for all 30 ports, while the C version is done in 2.0 seconds. IP table compression reduced the forwarding entries² to about 4,200.

The loop detection test *found 12 infinite loop paths* (ignoring TTL), such as path L1 in Figure 3.16, for packets destined to 10 different IP addresses. These loops are caused by interaction between spanning tree protocols of two VLANs: A packet broadcast on VLAN 1 can reach the leaves of the spanning tree of VLAN 1, where IP forwarding on a leaf node forwards it to VLAN 2. Then, the packet is broadcasted on VLAN 2 and is forwarded at the leaf node of VLAN 2 back to the original VLAN, where the process can continue.

Although IP TTL will terminate this process, if the TTL is 32 and the normal path length is 3, this consumes 10 times the normal resources during looping periods. More importantly, it shows how protocol interactions can lead to subtle problems. Each VLAN has a separate spanning tree that prevents loops, but VLANs are often defined manually. More generally, individual protocols often contain automated mechanisms that guarantee correctness for the protocol by itself, but the interaction between protocols is often done manually. Such manual configuration often leads to errors which Hassel can check for; route redistribution [32] provides another example of how manual connection of different routing protocols can lead to errors.

We also found four other loop paths, similar to L1, L2 and L3 in Figure 3.16 for packets destined to 16 subnets. However, these loops were single-round loops because when the packets return to the injection port, they are assigned to a VLAN not defined on that box and hence will be dropped. Table 3.2 summarizes performance results. Note that we can trivially speed up the loop detection test by running each per-port test on a separate machine.

Detecting possible configuration mistakes: As a second example, consider a configuration mistake that could cause packets to loop between backbone router

²There were four routers that together had 733,000 forwarding entries because no default BGP route was received. As a result, they kept one entry for every Internet subnet of which they knew, but all with the same output port. IP compression reduced their table size by three orders of magnitude.

	Python	C
Time to generate transfer functions	151 s	-
Runtime of loop detection test (30 ports)	560 s	2.0 s
Average per port runtime	18.6 s	66 ms
Max per port runtime	135 s	500 ms
Min per port runtime	8 s	5 ms
Average runtime of reachability test	13 s	40 ms

Table 3.2: Run time of the loop detection and reachability tests on Stanford backbone network, using implementations of Hassel in Python and C.

1 and the Internet. Stanford owns the IP subnet 171.64.0.0/14. However, not all of these IP addresses are currently in use. The backbone routers have an entry to route those IP addresses that are in use to the correct OZ router. Also, the default route in the backbone routers (0.0.0.0/0) is to send packets to the Internet. To avoid sending packets destined to the unused Stanford IP addresses to the outside world, the backbone routers have a manually installed null rule that drops all packets destined to 171.64.0.0/14 if they do not match any other rule.

Suppose that by mistake, the null rule is set to drop 171.64.0.0/16 IP addresses (i.e., the /14 is fat-fingered to a /16). Assume that the ISP's router does not filter incoming traffic from Stanford destined to Stanford. Then, packets sent to unused addresses in 17.64.0.0/14 that are not in 171.64.0.0/16 will loop between the backbone routers and the ISP's router. This scenario was simulated, and the test successfully detected the loop in less than 10 minutes (as in Table 3.2). More importantly, the tool allowed the loop to be traced to the line in the configuration file that caused the error. In particular, the tool output shows that packets in the loop match the default 0.0.0.0/0 forwarding rule and not the 171.64.0.0/16 rule in the backbone router.

Verifying reachability to an OZ router: As a third example, Hassel was used to calculate the reachability function from the OZ router connected to the student dorms to the OZ router connected to the computer science department. This verified that all of the intended security restrictions, as commented by the admin, in the config file were met. These restrictions included ports and IP addresses that were closed to outside users. Table 3.2 shows the run time for this test. We have heard

from managers that many restrictions and ACLs were inserted by earlier managers and are still preserved because current managers are afraid to remove them. Hassel allows managers to do “what if” analysis to see the effect of deleting an ACL.

3.5.2 Checking Slice Isolation

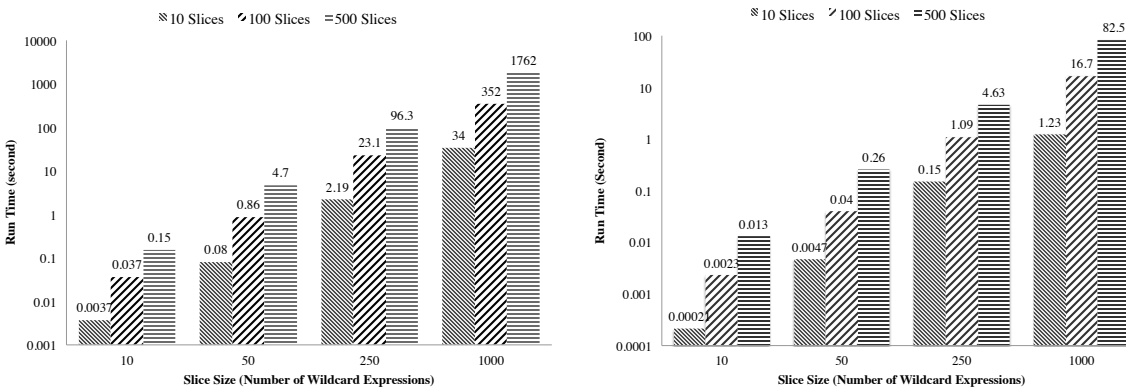
Suppose we want to replace VLANs in the Stanford backbone network with the more flexible slices made possible by FlowVisor. Stanford’s VLANs mostly carry traffic belonging to a particular subnet—e.g., VLAN 74 carries subnet 171.64.74.0/24. VLAN 74 is equivalent to a FlowVisor slice across the same routers with header space: $ip_dst(h) = 171.64.74.0/24$ or $ip_src(h) = 171.64.74.0/24$.

We would like to understand how quickly we can verify new slices which are created on-demand in the Stanford network. Recall from Section 3.3, we need to perform two checks:

1. When creating a slice, we need to make sure its network space does not overlap with an existing slice.
2. Whenever a new forwarding rule is added, we need to check that it cannot cause packet leakage.

In this experiment, we generated random slices with a topology similar to the existing VLANs, as follows: For each slice, we randomly picked two operational zones together with all router ports and switches that connect them. Then, we added random pieces of header space to the network space definition of each slice by picking X source or destination subnets of random prefix length (or random TCP ports). Here, X , the number of wildcard expressions used to describe a slice, denotes the slice’s *complexity*. While all existing VLAN slices in Stanford require fewer than 10 wildcard expressions³, we explored the limits of performance by varying X from 10 to 1,000. We ran experiments to create new slices of varying complexity, X , while there were 10, 100, or 500 existing slices.

³In most cases, two expressions suffice.



(a) Disjointness of slice definitions.

(b) Non-leakage of a new forwarding rule.

Figure 3.17: Run time of the slice isolation check on Stanford backbone network for randomly generated slices. The check uses the Python based implementation of Hassel.

In the first experiment, we randomly created a new slice and checked the disjointness of slice definitions by looking for intersection with all the existing slices. This test should be done every time a new slice is created, and we hope it will complete in a few minutes or less. In the second experiment, we emulated the behavior of adding a new rule with a rewrite action. We generated random rules and checked if it could possibly cause a packet leak to another slice. This test has to run every time a new rule is added, so it needs to be really fast.

Figure 3.17(a) and Figure 3.17(b) shows the run time of our tests using the Python implementation of Hassel. As the figures suggest, if the slices are not very complex (i.e., can be explained with fewer than 50 wildcard expressions)⁴, then the tests run almost instantly. Surprisingly, the tests are very fast even when there are 500 slices—more than adequate for existing networks. If there are only 10 or 100 slices, the checks can be done on very complex slices. The experimental run times matches the expected complexity in Section 3.3.3, which is *quadratic* in the number of wildcard expressions per slice and *linear* in the number of slices.

⁴This includes wildcard expressions that are included in, or excluded from, the definition of a slice.

3.5.3 Debugging a Protocol Design

This section describes a plausible scenario in which a loop is caused by a protocol design mistake. The scenario allows the loop size to be parameterized to examine how detection time varies with loop size. It also showcases how Hassel can model IP options and other variable length fields using custom transfer function rules.

In our scenario, Alice—a networking researcher—invents a new loose source routing protocol, IP*. IP* allows a source to specify the sequence of middle boxes that a packet must pass through. Alice’s protocol has the header format shown in Figure 3.18(a). IP* works exactly like normal IP, except that it updates the header at the first router where a packet enters the IP* network. Figure 3.18(b) shows an example of header update for a stack size of three. The header update operation sets the current source address to the “sender IP address” field, rewrites the destination IP address to the address at the top of the stack, and rotates all the IP addresses in the stack. After processing, a destination middlebox swaps the destination and source IP addresses and resends the packet to a router. Alice designs IP* to allow tunneling across existing IP networks.

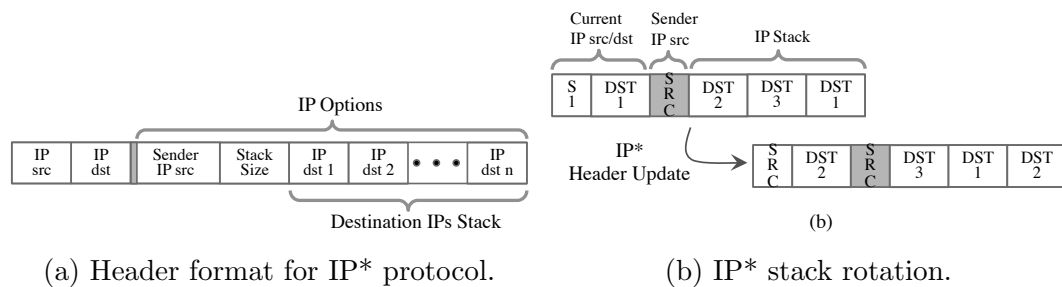


Figure 3.18: Header format and stack rotation of IP* protocol.

IP* stack rotation consists of the following steps: 1. The packet’s IP source is replaced by the sender IP source. 2. The packet’s IP destination is replaced by the top of the stack. 3. The stack is rotated.

Alice tries IP* in the network topology of Figure 3.19 and verifies that packets are successfully routed via middle boxes $M1$ and $M2$. Figure 3.19 also shows how the packet header changes as it passes through $M1$ and $M2$ to the final destination,

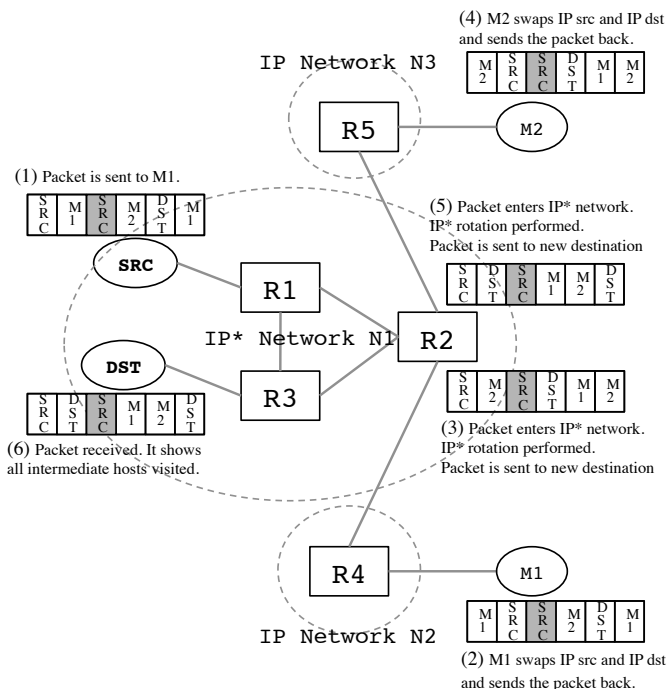


Figure 3.19: Example of an IP* network with routing through middleboxes. The figure labels 6 steps of packet processing along with the transformed header at each step. R_2 is the entry point to the IP* network which performs IP* header updates.

DST , in six steps. At DST , the stack contains the IP address of all middleboxes visited.

To continue her verification of IP*, Alice tries the more complex network in Figure 3.20 where the destination is attached to a different IP* network than the source. Before deploying IP*, she uses the loop detection algorithm from Section 3.2 and finds several loops. The most interesting one is an infinite loop consisting of the two strange loops below:

- 1) $R_2 \rightarrow R_5 \rightarrow M_2 \rightarrow R_5 \rightarrow R_2 \rightarrow R_4 \rightarrow M_1 \rightarrow R_4 \rightarrow R_2 \rightarrow R_3 \rightarrow R_6 \rightarrow R_3 \rightarrow R_2$
- 2) $R_2 \rightarrow R_4 \rightarrow M_1 \rightarrow R_4 \rightarrow R_2 \rightarrow R_5 \rightarrow M_2 \rightarrow R_5 \rightarrow R_2 \rightarrow R_3 \rightarrow R_6 \rightarrow R_3 \rightarrow R_2$

Alice now realizes that having a second IP* network in the path can cause loops. She removes the concept of a “first” router, instead adding a pointer that describes

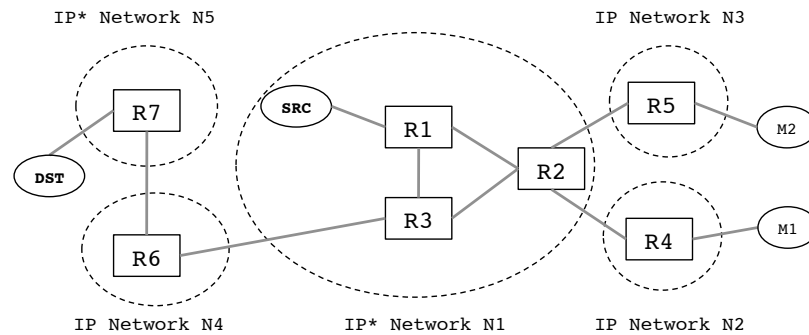


Figure 3.20: Alice's second network topology can cause infinite loops.

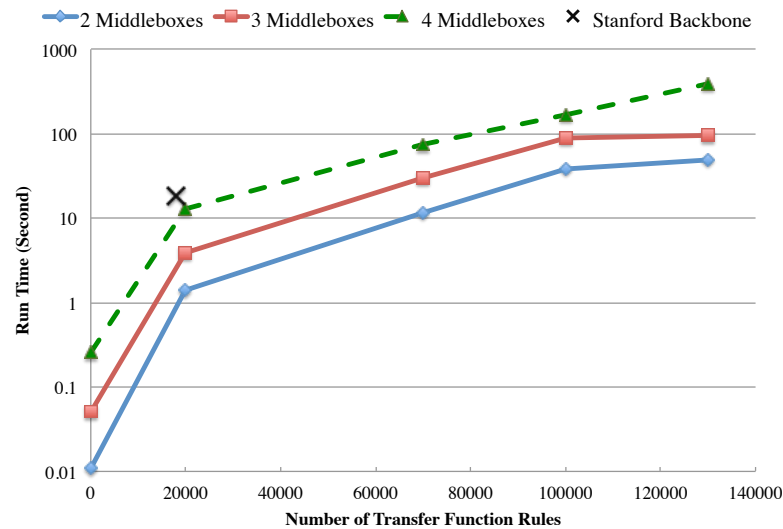


Figure 3.21: Running time of the loop detection test on Alice's IP* network using the Python based Hassel.

the middlebox to visit next. We should be clear that by no means are we proposing IP* as a viable protocol. Instead, we hope this example suggests that Hassel could be a useful tool for protocol designers as well as network managers. While this particular loop could be caught by a simulation, if there were many sources and destinations and the loop was caused by a more obscure pre-condition, then a simulation may not uncover the loop. By contrast, static checking using Hassel will find all loops.

Figure 3.21 shows the *per-port* performance of the infinite loop detection algorithm for the ports participating in the loops. We varied the number of middle boxes connected to R_2 (e.g., $M1$ and $M2$) and the number of router forwarding entries. For four middle boxes, the loop has a length of 72 nodes! Finding a large and complex loop in a network with 100,000 forwarding rules and *custom* actions, in less than four minutes using less than 50 lines⁵ of code (Figure 4.9), demonstrates the power of the header space analysis framework.

```
def detect_loop(NTF, TTF, ports, test_packet):
    loops = []
    for port in ports:
        propagation = []
        p_node = {}
        p_node["hdr"] = test_packet
        p_node["port"] = port
        p_node["visits"] = []
        p_node["hs_history"] = []
        propagation.append(p_node)

        while len(propagation)>0:
            tmp_propag = []
            for p_node in propagation:
                next_hp = NTF.T(p_node["hdr"],p_node["port"])
                for (next_h,next_ps) in next_hp:
                    for next_p in next_ps:
                        linked = TTF.T(next_h,next_p)
                        for (linked_h,linked_ports) in linked:
                            for linked_p in linked_ports:
                                new_p_node = {}
                                new_p_node["hdr"] = linked_h
                                new_p_node["port"] = linked_p
                                new_p_node["visits"] = list(p_node["visits"])
                                new_p_node["visits"].append(p_node["port"])
                                new_p_node["hs_history"] = list(p_node["hs_history"])
                                new_p_node["hs_history"].append(p_node["hdr"])
                                if len(new_p_node["visits"]) > 0 \
                                    and new_p_node["visits"][0] == linked_p:
                                    loops.append(new_p_node)
                                    print "loop detected"
                                elif linked_p not in new_p_node["visits"]:
                                    tmp_propag.append(new_p_node)
            propagation = tmp_propag

    return loops
```

Figure 3.22: Loop detection code

⁵Not counting the underlying Hassel implementation

3.6 Limitations

Hassel is a tool designed for static analysis of network, to detect forwarding and configuration errors. It is no panacea, serving as one tool among many needed by protocol designers, software developers and network operators. For example, while it might tell us that a routing algorithm is broken because routing tables are inconsistent, it does not tell us why. Even if the routing tables are consistent, header space analysis offers no clues as to whether routing is efficient or meets the objectives of the designer. Despite this, HSA-based verifications could play a similar role in networks as post-layout verification tools do in chip design or static analysis checkers do in compilation. It checks the low-level forwarding state against a set of universal invariants, without understanding the intent or aspirations of the protocol designer. To analyze protocol correctness, other approaches such as [21] should be used.

Similarly, while Hassel can pinpoint the specific entry in the forwarding table or line in the configuration file that causes a problem, it does not tell us how or why those entries were inserted or how they will be evolved as the box receives future messages. Finally, like all static checkers, Hassel cannot deal well with churn in the network except to periodically run it based on snapshots; thus, it can only detect problems that persist longer than the sampling period. In Chapter 4, we introduce a new tool, based on header space analysis that can run these verifications incrementally and in real time.

3.7 Related Work

Roscoe et al.'s predicate routing [45] introduces the notion of pushing a predicate as part of designing a routing mechanism. This is similar to pushing an all-wildcard flow which is used in our reachability and loop detection algorithms. Xie et al.'s reachability analysis [52] uses test packets to determine reachability (not detecting loops) for the special case of TCP/IP networks. The static analysis tools described in [2,38,54] are designed specifically for TCP/IP firewalls, and Feamster et al.'s work in [13] finds reachability failures in BGP routers.

HSA is broader in two ways. First, it is a *framework* rather than a tool for detecting a single type of network failure. It can perform a range of network verification tasks and, as we will see in the next chapters, can be used as the foundation for building more tools and techniques for testing and debugging networks. Second, the algorithms developed in this framework are independent of protocols.

Model checkers, SAT solvers and theorem provers are among other commonly used tools for network verification [36]. However, when these tools detect a violation of a specification (e.g., reachability violations), they are limited to providing a *single counterexample* and not the *full set* of failed packet headers that header space analysis provides. Also, these tools are at least two orders of magnitude slower than is the C-based Hassel when performing the same checks, as these off-the-shelf tools do not benefit from various domain-specific optimizations that have gone into Hassel.

3.8 Summary

In this chapter, I used the header space analysis framework to design algorithms for checking three important properties of networks: finding reachability between end hosts, detecting forwarding loops and checking isolation of network slices. I described the implementation of these algorithms using the header space library (Hassel) and evaluated their functionality and performance. As we saw in Section 3.5, the loop detection test could find a number of loops in the Stanford backbone network, the reachability analysis verified correct implementation of access control policies in this network, and the slice isolation check could verify isolation of complex slices created randomly on the Stanford network.

Chapter 4

Real-Time Policy Checking using Rule Dependency Graph

Network state may change rapidly in response to customer demands, load conditions or configuration changes. The network must also ensure correctness conditions such as isolating tenants from each other and from critical services. Hassel and other offline policy checkers cannot verify correctness in real time because of the need to collect “state” information from the entire network and the time it takes to analyze this state. Software Defined Networks (SDNs) offer an opportunity in this respect as they provide a logically-centralized view from which every proposed change can be checked for policy compliance. But there remains a need for a fast compliance checker.

This chapter introduces a real-time policy checking tool called *NetPlumber* based on the header space analysis framework. NetPlumber *incrementally* checks for the compliance of state changes using a novel conceptual representation, called *the Rule Dependency Graph*, that maintain a dependency graph between rules. It improves upon Hassel in two ways. First, by running HSA checks incrementally, NetPlumber enables real-time checking of updates, which in turn can detect rule changes violating a policy before they hit the data plane or raise an alarm as soon as a link failure breaks a policy. Second, NetPlumber provides a flexible way to express and check complex policy queries without writing new ad hoc code for each policy check.

NetPlumber can detect simple invariant violations such as loops and reachability failures. It can also check more sophisticated policies that reflect the desires of human operators; for example, “Web traffic from A to B should never pass through waypoints C or D.” Or “Packets between A and B should not go through more than 3 hops.”

While NetPlumber is a natural fit for SDNs, its network model is conceptually applicable to conventional networks as well. In SDNs, NetPlumber is deployed in line with the control plane, and observes state changes (e.g., OpenFlow messages) between the control plane and the switches (Figure 4.1). NetPlumber checks every event, such as installation of a new rule, removal of a rule, port or switch up and down events, against a set of policies and invariants. Upon detecting a violation, it calls a function to alert the user or block the change. In conventional networks, NetPlumber can acquire state change notifications through SNMP traps or by frequently polling switches.

NetPlumber’s speed easily exceeds the requirements for an enterprise network in which configuration state changes infrequently—say once or twice per day. But in modern multi-tenant data centers, fast programmatic interfaces to the forwarding plane allow control programs to rapidly change the network configuration—perhaps thousands of times per second. For example, we may move thousands of virtual machines (VMs) to balance the load, with each change requiring a tenant’s virtual network to be reconfigured. With an average of less than *1 ms* policy check time on Google SDN, Internet2 and Stanford’s backbone network, NetPlumber can handle more than 1,000 rule updates per second.

4.1 NetPlumber Architecture

4.1.1 Overview of NetPlumber

NetPlumber has a much faster update time than Hassel because instead of recomputing all of the transformations each time the network changes, it incrementally updates only the portions of the results affected by the change. Underneath, NetPlumber still uses HSA. Thus, it inherits the ability to verify a wide range of policies

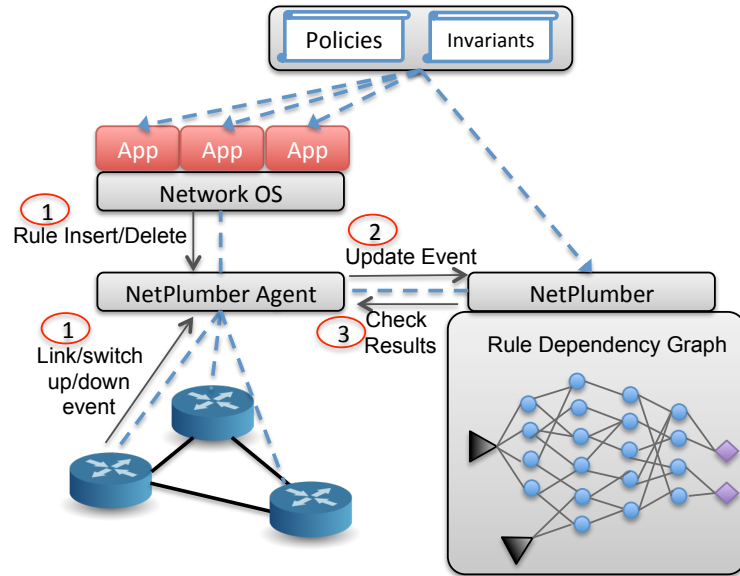


Figure 4.1: Deploying NetPlumber as a policy checker in SDNs.

from HSA, including reachability between ports, loop-freedom, and isolation between groups, while remaining protocol agnostic.

Figure 4.1 shows NetPlumber checking policies in an SDN. An agent sits between the control plane and switches and sends every state update (installation or removal of rules, link up or down events) to NetPlumber. Internally, NetPlumber creates and maintains a model of the network, which is used to verify policies in real time. In response to network state changes, NetPlumber updates its network model and re-evaluates the policy checks affected by the update. If a violation occurs, NetPlumber performs a user-defined action such as removing the violating rule or notifying the administrator.

The heart of NetPlumber is its network model, called *the Rule Dependency Graph (RDG)*, which captures all possible paths of flows¹ through the network. Nodes in the graph correspond to the *rules* in the network, and directed edges represent the *next hop dependency* of these rules.

¹In what follows, a flow corresponds to any region of header space.

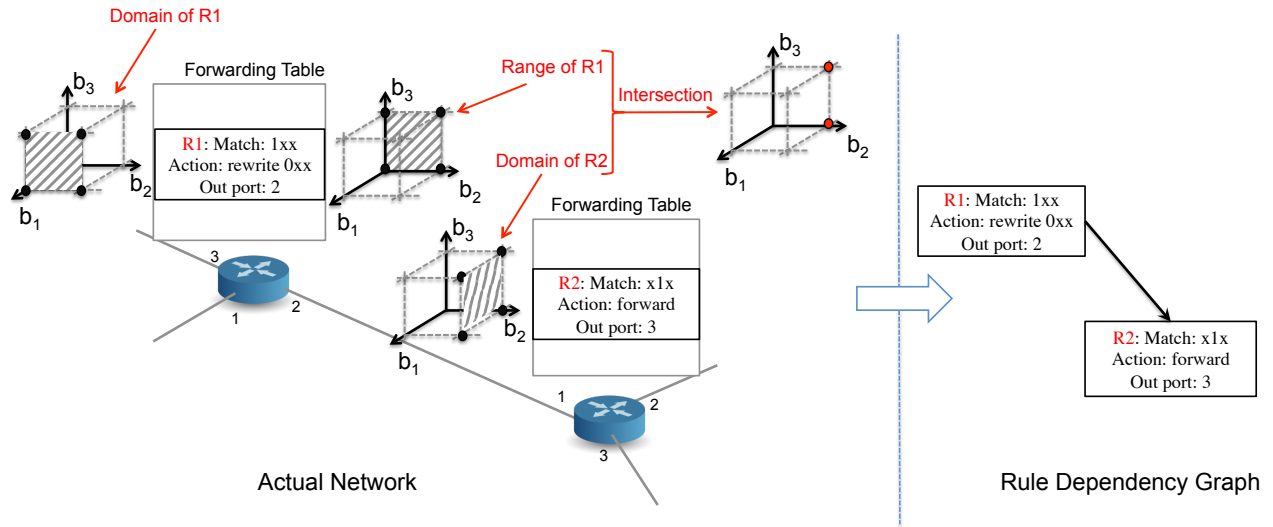


Figure 4.2: Example of finding next hop dependency in NetPlumber. Here, the switches are connected by a link and the range of $R1$ and the domain of $R2$ has some intersection: $01x$.

- A rule is an OpenFlow-like $\langle \text{match}, \text{action} \rangle$ tuple where the action can be `forward`,² `rewrite`, `encapsulate`, `decapsulate`, etc.
- Rule a in box A has a next hop dependency to rule b in box B if (1) there is a physical link from A to B , and (2) the domain of rule b and the range of rule a have a non-empty intersection on the A - B link. The domain of a rule is the set of input headers and ports, $\{(h_i, p_i)\}$, that match the rule. The range is the set of output headers and ports, $\{(h_o, p_o)\}$, created by the `action` transformation on the rule's domain. To clarify, consider the example in Figure 4.2 consisting of two rules, $R1$ and $R2$, on two boxes that are connected by a physical link. The range of $R1$ is $\{(0xx, 2)\}$ and the domain of $R2$ is $\{(x1x, 1), (x1x, 2), (x1x, 3)\}$. On the connecting link ($1 \leftrightarrow 2$), the intersection or domain of $R1$ and range of $R2$ is $01x$. Therefore, there will be a next hop dependency from $R1$ to $R2$ in the rule dependency graph.

Initialization: NetPlumber is initialized by examining the forwarding tables to build the rule dependency graph. It then computes reachability by computing the set

²A drop rule is a special case of a forward rule with an empty set of output ports.

of packets from source port s that can reach destination port d . To do so, it starts by pushing an “all-wildcard flow” into the rule dependency graph from s and propagating it along the edges of the graph. At each rule node, the flow is filtered by the `match` part of the rule and then transformed by the `action` part of the rule. The resulting flow is then propagated along the outgoing edges to the next node. The portion of the flow, if any, that reaches d is the set of all packets from s that can reach d . To speed up future calculations, whenever a rule node transforms a flow, it stores a copy of the flow locally in the node object before forwarding the flow to the next hop nodes. This caching lets NetPlumber quickly update reachability results every time a rule changes.

Operation: In response to the insertion or deletion of rules in switches, NetPlumber adds or removes nodes and updates the routing of flows in the dependency graph. It also re-runs those policy checks that need to be updated.

4.1.2 The Rule Dependency Graph

As mentioned in the last section, NetPlumber creates and maintains a network model in the form of a forwarding graph called the rule dependency graph. The nodes of this graph are the forwarding rules, and directed edges represent the next-hop dependency of these rules. We call these directed edges *pipes* because they represent possible paths for flows. A pipe from rule a to b has a *pipe filter* that is the intersection of the range of a and the domain of b . When a flow passes through a pipe, it is filtered by the pipe filter. Conceptually, the pipe filter represents all packet headers at the output of rule a that can be processed by b .

A rule node corresponds to a rule in a forwarding table in a switch. Forwarding rules have priority; when a packet arrives to the switch, it is processed by the highest priority matching rule. Similarly, the NetPlumber needs to consider rule priorities when deciding which rule node will process a flow. For computational efficiency, each rule node keeps track of higher priority rules in the same table. It calculates the domain of each higher priority rule, subtracting it from its own domain. We refer to this as *intra-table dependency* of rules.

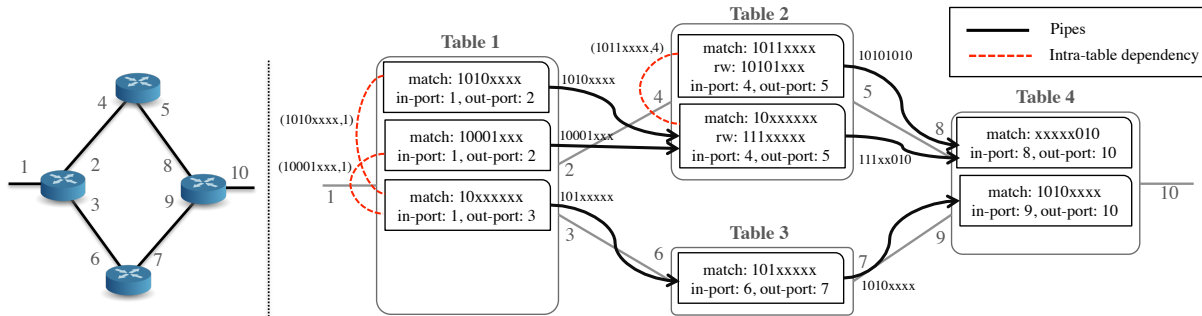


Figure 4.3: Rule dependency graph of a simple network.

The network consists of 4 switches, each with one table. Arrows represent pipes. Pipe filters are shown on the arrows. Dashed lines indicate intra-table rule dependency. The intersecting domain and input port is shown along the dashed lines.

Figure 4.3 shows an example network and its corresponding rule dependency graph. It consists of 4 switches, each with one forwarding table. For simplicity, all packet headers are 8 bits. We will use this example through the rest of this section.

Let's briefly review how the rule dependency graph in Figure 4.3 was created:

1. There is a pipe from rule 1 in Table 1 (rule 1.1) to rule 2 in Table 2 (rule 2.2) because
 - (a) ports 2 and 4 are connected, and
 - (b) the range of rule 1.1 (1010xxxx, 2) and the domain of rule 2.2 (10xxxxxx, 4) has a non-empty intersection (pipe filter: 1010xxxx).
2. Similarly, there is a pipe from rule 2.2 to rule 4.1 because
 - (a) ports 5 and 8 are connected, and
 - (b) the range of rule 2.2 (111xxxxx, 5) and the domain of rule 4.1 (xxxxx010, 8) has a non-empty intersection (pipe filter: 111xx010).
3. Rule 1.1 has an intra-table influence on rule 1.3 because their domains and input port sets have a non-empty intersection (intersection: (1010xxxx, 1)).
4. The rest of this graph was created in a similar fashion.

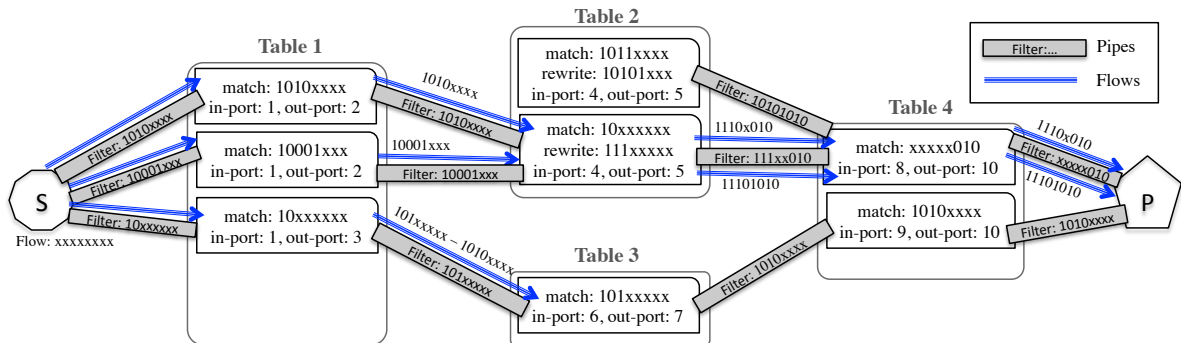


Figure 4.4: Finding reachability between S and P using the rule dependency graph. Source node S is generating all-wildcard flow and inserting it into the rule dependency graph. The solid lines show the path of flow from the source to the destination. Flow expressions are shown along the flows.

4.1.3 Source and Sink Nodes

NetPlumber converts policy and invariants to equivalent reachability assertions. To compute reachability, it inserts flow from the source port into the rule dependency graph and propagates it toward the destination. This is done using a “flow generator” or *source node*. Just like rule nodes, a source node is connected to the rule dependency graph using directed edges (pipes), but instead of processing and forwarding flows, it generates flow.

Continuing our example, we compute reachability between port 1 and 10 in Figure 4.4 by connecting a source node, generating an all-wildcard flow, to port 1. We have also connected a special node called a *probe* node to port 10. Probe nodes will be discussed in the next section. The propagation of flows in the rule dependency graph occurs as follows:

1. The flow generated by the source node reaches rules 1.1, 1.2 and 1.3.
2. Rule 1.1 and 1.2 are not affected by any higher priority rules and do not rewrite flows. Therefore, the input flow is simply forwarded to the pipes connecting them to rule 2.2 (i.e., 1010xxxx and 10001xxx flows reach rule 2.2).

3. Rule 1.3 has an intra-table dependency to rule 1.1 and 1.2. This means that from the incoming $10xxxxxx$ flow, only $10xxxxxx - (1010xxxx \cup 10001xxx)$ should be processed by rule 1.3. The remainder has already been processed by higher priority rules. Rule 1.3 is a simple forward rule and will forward the flow, unchanged, to rule 3.1. However, when this flow passes through the pipe filter between rule 1.3 and 3.1 ($101xxxxx$), it shrinks to $101xxxxxx - 1010xxxx$.³
4. The flows that have reached rule 2.2 continue propagating through the rule dependency graph until they reach the probe node (P), as depicted in Figure 4.4.
5. The flow which has reached rule 3.1 does not propagate any further as it cannot pass through the pipe connecting rule 3.1 to rule 4.2 because the intersection of the flow ($101xxxxxx - 1010xxxx = 1011xxxx$) and pipe filter ($1010xxxx$) is empty.

Sink Nodes: Sink nodes are the dual of source nodes. A sink node absorbs flows from the network. Equivalently, a sink node generates “sink flow,” which traverses the rule dependency graph in the reverse direction. When reaching a rule node, a sink flow is processed by the inverse of the rule.⁴ Reachability can be computed using sink nodes. If a sink node is placed at the destination port D , then the sink flow at source port S gives us the set of packet headers from S that will reach D . Sink nodes do not increase the expressive power of NetPlumber; they only simplify or optimize some policy checks, as explained in Section 4.2.

4.1.4 Probe Nodes

A fourth type of node, called a *probe node*, is used to check policy or invariants. Probe nodes can be attached to appropriate locations of the rule dependency graph and can be used to check the path and header of the received flows for violations of expected behavior. In Section 4.2, we discuss how to check a policy using a source (sink) and

³ $[10xxxxxx - (1010xxxx \cup 10001xxx)] \cap 101xxxxx = 101xxxxx - 1010xxxx$.

⁴The inverse of a rule gives us all input flows that can generate a given flow at the output of that rule.

probe node. As a simple example, consider checking the policy that in Figure 4.3, port 1 and 10 can only talk using packets matching `xxxxx010`. To check this policy, we place a source node at port 1 (S), a probe node at port 10 (P), and configure P to check whether all flows received from S match `xxxxx010` (Figure 4.4).

There are two types of probe nodes—*source probe nodes* and *sink probe nodes*. The former check constraints on flows generated by source nodes, and the latter check flows generated by sink nodes. We refer to both as probe nodes.

4.1.5 Updating NetPlumber State

As events occur in the network, NetPlumber needs to update its rule dependency graph and re-route the flows. There are six events that NetPlumber needs to handle:

1. **Adding New Rules:** When a new rule is added, NetPlumber first creates pipes from the new rule to all potential next hop rules and from all potential previous hop rules to the new rule. It also needs to find all intra-table dependencies between the new rule and other rules within the same table. In our example in Figure 4.5, a new rule is added at the 2nd position of Table 1. This creates three new pipes to rules 2.1, 2.2 and the source node and one intra-table dependency for rule 1.4.

Next, NetPlumber updates the routing of flows. To do so, it asks all of the previous hop nodes to pass their flows on the *newly created* pipes. The propagation of these flows then continues normally through the rule dependency graph. If the new rule has caused any intra-table dependency for lower priority rules, we need to update the flows passing through those lower priority rules by subtracting their domain intersection from the flow. Continuing our example in Figure 4.5, after adding the new rule, the new highlighted flows propagate through the network. Also, the intra-table dependency of the new rule on rule 1.4 is subtracted from the flow received by rule 1.4, thus shrinks the flow to the extent that it cannot pass through the pipe connecting it to rule 3.1 (empty flow on the bottom path).

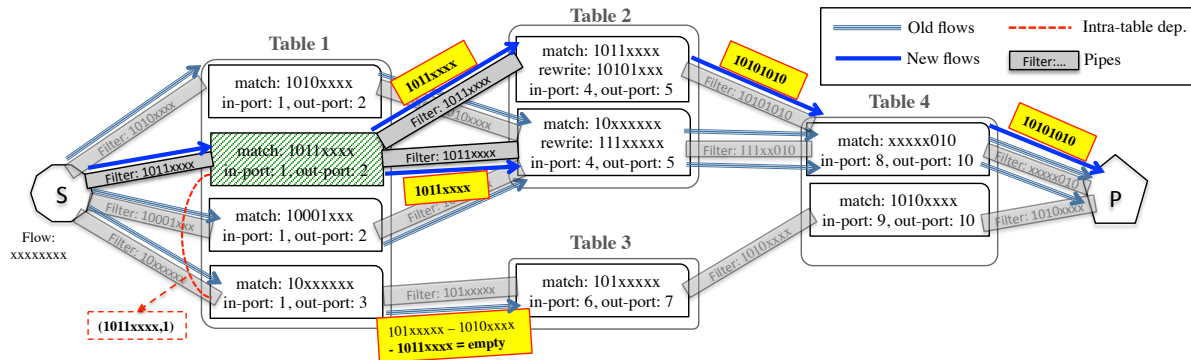


Figure 4.5: Updating the rule dependency graph after rule insertion.

Rule 1.2 (shaded in green) is added to Table 1. As a result a) Three pipes are created connecting rule 1.2 to rule 2.1 and 2.2 and to the source node. b) Rule 1.4 will have an intra-table dependency to the new rule (1011xxxx,1). c) The flows highlighted in bold will be added to the rule dependency graph. Also, the flow going out of rule 1.4 is updated to empty.

2. **Deleting Rules:** Deleting a rule causes all flows that pass through that rule to be removed from the rule dependency graph. Further, if any lower priority rule has any intra-table dependency on the deleted rule, the effect should be added back to those rules. Figure 4.6 shows the deletion of rule 1.1 in our example. Note that deleting this rule causes the flow passing through rule 1.3 to propagate all the way to the probe node, because the influence of the deleted rule is now added back.
3. **Link-Up:** Adding a new link to the network may cause additional pipes to be created in the rule dependency graph, because more rules will now have physical connections between them (first condition for creating a pipe). The nodes on the input side of these new pipes must propagate their flows on the new pipes, and then through the rule dependency graph as needed. Usually, adding a new link creates a number of new pipes, making a link-up event slower to process than a rule update.

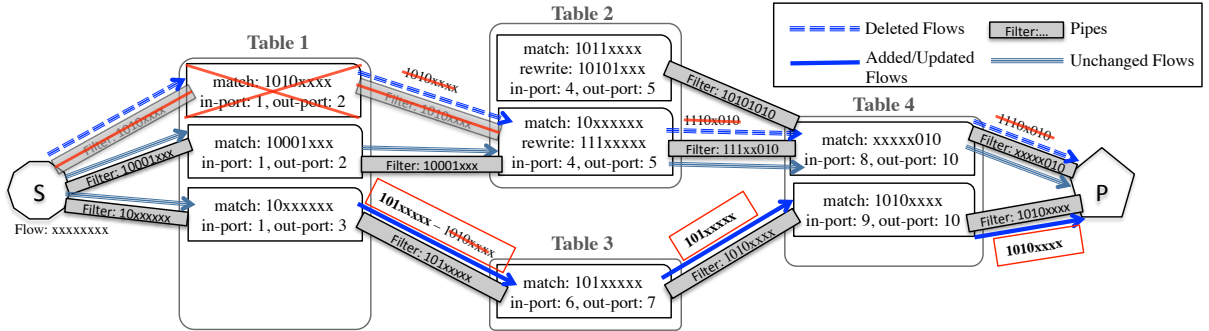


Figure 4.6: Updating the rule dependency graph after rule deletion.

Deleting rule 1.1 in Table 1 causes the flow that passes through it to be removed from the rule dependency graph. Also, since the intra-table dependency of rule 1.3 to this rule is removed, the flow passing through 1.3 via the bottom path is updated.

4. **Link-Down:** When a link goes down, all of the pipes created on that link are deleted from the rule dependency graph, which in turn removes all of the flows that pass through those pipes.
5. **Adding New Tables:** When a new table (or switch) is discovered, the rule dependency graph remains unchanged. Changes only occur when new rules are added to the new table.
6. **Deleting Tables:** A table is deleted from NetPlumber by deleting all of the rules contained in that table.

4.1.6 Complexity Analysis

The complexity of NetPlumber for the addition of a single rule is $O(r + spd)$, where r is the number of entries in each table and s is the number of source (sink) nodes attached to the rule dependency graph (which is roughly proportional to the number of policies we want to check), p is the number of pipes to and from the rule, and d is the diameter of the network.

The run time complexity arises as follows: When a new rule is added, we need to first find intra-table dependencies. These require intersecting the match portion of

the new rule with the `match` of all of the other rules in the same table. We also need to create new pipes by doing $O(r)$ intersections of the range of the new rule with the domain of rules in the neighboring tables ($O(r)$ such rules).

Next, we need to route flows. Let us use the term *previous nodes* to denote the set of rules that have a pipe to the new rule. First, we need to route the flows at previous nodes to the new rule. There are $O(s)$ flows on each of these previous nodes because each source (sink) node that is connected to NetPlumber can add a flow. We need to pass these flows through $O(p)$ pipes to route them to the new rule. This is $O(sp)$ work. With a linear fragmentation⁵ argument (as discussed in Section 3.1.3), there will be $O(s)$ flows that will survive this transformation through the pipes,⁶ and not $O(sp)$ flows. The surviving flows will be routed in the same manner through the rule dependency graph, requiring the same $O(sp)$ work at each node in the routing path. Since the maximum path length is the diameter d , the overall run time of this phase is $O(spd)$.

We also need to take care of intra-table dependencies between this rule and lower priority rules, and subtract the domain intersection from the flows received by lower priority rules. This subtraction is done lazily and is therefore much faster than flow routing; hence, we ignore its contribution to overall run time. Therefore, the overall run time of NetPlumber for a single rule insertion will be $O(r + spd)$.

4.2 Checking Policies and Invariants

A probe node monitors flows received on a set of ports and check policy conditions on these flows. In the rule dependency graph, a probe node may be attached to the output of every rule sending out flows on those ports. Each probe node is configured with a *filter* flow expression and a *test* flow expression. A flow expression, or *flowexp*

⁵As a reminder, this argument states that if we have R flows at the output of a transfer function, and we apply these flow to the next hop transfer functions with R rules per transfer function, we will get cR flows at the output where $c \ll R$ is a constant. This assumption is based on the observation that flows are routed end-to-end in networks. They are usually aggregated, and not randomly fragmented in the core of the network.

⁶An alternate way to reach the same conclusion is as follows: The new rule, after insertion, will look like any other rule in the network and should on average have $O(s)$ flows.

for short, is a regular expression specifying a set of conditions on the path and the header of the flows. The *filter* flowexp constrains the set of flows that should be examined by the probe node, and the *test* flowexp is the constraint that is checked on the matching flows. Probe nodes can be configured in two modes, *existential* and *universal*. A probe fires when its corresponding predicate is violated. An existential probe fires if none of the flows examined by the probe satisfy the test flow expression. In contrast, a universal probe fires when a single flow is received that does *not* satisfy the test constraint. More formally:

A universal probe checks the following: $\forall\{f \mid f \sim \text{filter}\} : f \sim \text{test}$. All flows f that satisfy the filter expression, satisfy the test expression as well.

An existential probe checks the following: $\exists\{f \mid f \sim \text{filter}\} : f \sim \text{test}$. There exist a flow f that satisfies both the filter and test expressions.

Sometimes, during a sequence of state updates, transient policy violations may be acceptable (e.g., a black hole is acceptable while installing a path in a network). NetPlumber probes can be turned off during the transition and turned back on when the update sequence is complete.

Using flow expressions described via the flowexp language, probe nodes are capable of expressing a wide range of policies and invariants. Section 4.2.1 will introduce the flowexp language. Sections 4.2.2 and 4.2.3 discuss techniques for checking for loops, black holes and other reachability-related policies.

4.2.1 Flowexp Language

Each flow at any point in the rule dependency graph, carries its complete path history, because it has a pointer to the corresponding flow at the previous hop (node). By traversing these pointers backward, we can examine the entire path history of the flow and all of the rules that processed this flow along the path. The flow history always begins at the generating source (or sink) node and ends at the probe node checking the condition.

```

Constraint → True | False | !Constraint
           | (Constraint | Constraint)
           | (Constraint & Constraint)
           | PathConstraint
           | HeaderConstraint;
PathConstraint → list(Pathlet);
Pathlet → Port Specifier [p ∈ {Pi}]
         | Table Specifier [t ∈ {Ti}]
         | Skip Next Hop [.]
         | Skip Zero or More Hops [.*]
         | Beginning of Path [^]
         | (Source/Sink node)
         | End of Path [$]
         | (Probe node);
HeaderConstraint → Hreceived ∩ Hconstraint ≠ φ
                 | Hreceived ⊂ Hconstraint
                 | Hreceived == Hconstraint;

```

Figure 4.7: Flowexp language grammar.

Flowexp is a regular expression language designed to check constraints on the history of flows received by probe nodes. Figure 4.7 shows the grammar of flowexp in a standard BNF syntax. Flowexp consists of logical operations (i.e., *and*, *or*, and *not*) on constraints enforced on the *path* or *header* of flows received on a probe node.

A *PathConstraint* is used to specify constraints on the path taken by a flow. It consists of an ordered list of *pathlets* that are checked sequentially on the path of the flow. A pathlet is a constraint on part of the flow path, which may include a specific set of ports or tables in the path or specified or unspecified number of hops. The complete set of pathlets in flowexp language is available in Figure 4.7. For example, a flow that originates from source S , with the path $S \rightarrow A \rightarrow B \rightarrow C \rightarrow P$ to probe P , will match flowexp “ $^(p = A)$ ” because port A comes immediately after the source node. It also matches “ $(p = A).(p = C)$ ” because the flow passes through exactly one intermediate port from A to C .

A *HeaderConstraint* can check the following

1. If the received header has any intersection with a specified header; this is useful when we want to ensure that some packets of a specified type can reach the probe.

2. If the received header is a subset of a specific header; this is useful when we wish to limit the set of headers that can reach the probe.
3. If the received header is exactly equal to a specified header; this is useful to check whether the packets received at the probe are exactly what we expect.

Since flowexp is very similar to (but much simpler than) standard regular expression language, any standard regexp checking technique can be used at probe nodes.

4.2.2 Checking Loops and Black Holes

As flows are routed through the rule dependency graph, each rule by default (i.e., without adding probe nodes for this purpose) checks received flows for loops and black holes. To check for a loop, each rule node examines the flow history to determine if the flow has passed through the current table before. If it has, a loop-detected callback function is invoked.⁷ If we are interested in knowing the repetition of the loop, we can invoke the algorithm described in Section 3.2.2 inside the callback function.

Similarly, a black hole is automatically detected when a flow is received by a non-drop-rule R that cannot pass through any pipes emanating from R . In this case, a black-hole-detected callback function is invoked.

4.2.3 Checking Policies on Path Predicates

In this section, we describe how to express reachability-related policies and invariants such as the isolation of two ports, reachability between two ports, reachability via a middle box, and a constraint on the maximum number of hops in a path. We express and check for such reachability constraints by attaching one or more source (or sink) nodes and one or more probe nodes in appropriate locations in the rule dependency graph. The probe nodes are configured to check the appropriate filter and test flowexp constraints, as shown below.

Basic Reachability Policy: Suppose we wish to ensure that a server port S should not be reachable from guest machine ports $\{G_1, \dots, G_k\}$.

⁷The callback function can optionally check the repetition of the loop.

Solution using a source probe: Place a source node that generates a wildcarded flow at each of the guest ports. Next, place a source probe node on port S and configure it to check for the flow expression: $\forall f : f.path \sim ![\wedge (p \in \{G_1, \dots, G_k\})]$, that is, a *universal* probe with no filter constraint and a test constraint that checks that the source node in the path is not a guest port.

If the policy instead requires S to be reachable from $\{G_1, \dots, G_k\}$, we could configure the probe node as follows: $\exists f : f.path \sim [\wedge (p \in \{G_1, \dots, G_k\})]$. Intuitively, this states that there exists some flow that can travel from guest ports to the server S . Note that the server S is not specified in the flow expression because the flow expression is placed at S .

Dual Solution using a sink probe: Alternately, we can put a sink node at port S and a sink probe node in each of the G_i ports. We also configure the probes with flowexp $\forall f : f.path \sim ![\wedge (p \in \{S\})]$, that is, the probes at the guest ports should not see any sink flow from the sink node at port S .

Reachability via a Waypoint: Next, suppose we wish to ensure that all traffic from port C to port S must pass through a “waypoint” node M .

Solution: Put a source node at C that generates a wildcarded flow and a probe node at S . Configure the probe node with the flowing expression: $\forall \{f \mid f.path \sim [\wedge (p \in \{C\})]\} : f.path \sim [\wedge .*(t = M)]$. This is a universal probe that filters flows that originate from C and verifies that they pass through the waypoint M .

Path length constraint: Suppose we wish to ensure that no flow from port C to port S should go through more than 3 switches. This policy was desired for the Stanford network, for which we found violations. The following specification does the job, assuming that each switch has one table.

Solution: Place a probe at S and a source node at C as in the previous example. Configure the probe node with the following constraint: $\forall \{f \mid f.path \sim [\wedge (p \in \{C\})]\} : f.path \sim [\wedge .\$ \mid \wedge ..\$ \mid \wedge ...\$]$. The filter expression ensures that the check is only done for flows from C , and the test expression only accepts a flow if it is one, two, or three hops away from the source.

Source probes versus Sink probes: Roughly speaking, if a policy is checking something at the destination, regardless of where the traffic comes from, then using

sink probes is more efficient. For example, suppose a manager wishes to specify that all flows arriving at a server S pass through waypoint M . Using source probes would require placing one source probe at every potential source. This can be computationally expensive as the run time of NetPlumber grows linearly with the number of source or sink nodes. On the other hand, if the policy is about checking a condition for a particular source, for example computer C should be able to communicate with all other nodes, then using a source probe will be more efficient. Intuitively, we want to minimize the amount of flow in the rule dependency graph required to check a given policy, as generating flow is computationally expensive.

4.2.4 Policy Translator

So far we have described a logical language called flowexp which is convenient for analysis and specifying precisely how flows are routed within the network. Flowexp is, however, less appropriate as a language for network managers to express higher level policy. Thus, for higher level policy specification, we decided to reuse the policy constructs proposed in the Flow-based Management Language (FML) [18], a high-level declarative language for expressing network-wide policies about a variety of different management tasks. FML essentially allows a manager to specify predicates about groups of users (e.g., faculty, students) and stipulates which groups can communicate. FML also allows additional predicates on the *types* of communication allowed, such as the need to pass through waypoints.

Unfortunately, the current FML implementation is tightly integrated with an OpenFlow controller, and so cannot be easily reused in NetPlumber. We worked around this by encoding a set of constructs inspired by FML in Prolog. Thus, network administrators can use Prolog as the frontend language to declare various bindings inspired by FML, such as hosts, usernames, groups, and addresses. Network administrators can also use Prolog to specify different policies. For example, the following policy describes 1) the `guest` and `server` groups, and 2) a policy: “Traffic should go through firewall *if* it flows from a guest to a server.”

```
guest(sam).
```

```

guest(michael).
server(webserver).
waypoint(HostSrc, HostDst, firewall):-
    guest(HostSrc),
    server(HostDst).

```

We have written a translator that converts such high-level policy specifications written in Prolog into 1) the placement of source nodes, 2) the placement of probe nodes, and 3) the filter and test expressions for each probe node. In the example above, the translator generates two source nodes at Sam and Michael’s ports and one probe node at the web server’s port. The `waypoint` keyword is implemented by `flowexp:.*(t=firewall)`.

The output of the translator is, in fact, a C++ struct that lists all source, sink, and probe nodes. The source probes and sink probes are encoded in `flowexp` syntax using ASCII text. Finally, NetPlumber reads the output of the translator and attaches the source and probe nodes to the rule dependency graph as instructed.

Note that because FML is not designed to declare path constraints that can be expressed in `flowexp`, we found it convenient to make the translator extensible. For example, two new policy constructs we have built-in beyond the FML-inspired constructs are “at most N hops” and “immediately followed by,” and it is easy to add more constructs.

4.3 Distributed NetPlumber

NetPlumber is memory-intensive because it maintains considerable data about every rule and every flow in the rule dependency graph. For very large networks, with millions of rules and a large number of policy constraints, NetPlumber’s memory requirements can exceed that of a single machine. Further, as shown in Section 4.1.6, the run time of NetPlumber grows linearly with the size of the tables. This can be potentially unacceptable for very large networks.

Thus, a natural approach is to run parallel instances of NetPlumber, each verifying a subset of the network and each small enough to fit into the memory of a single

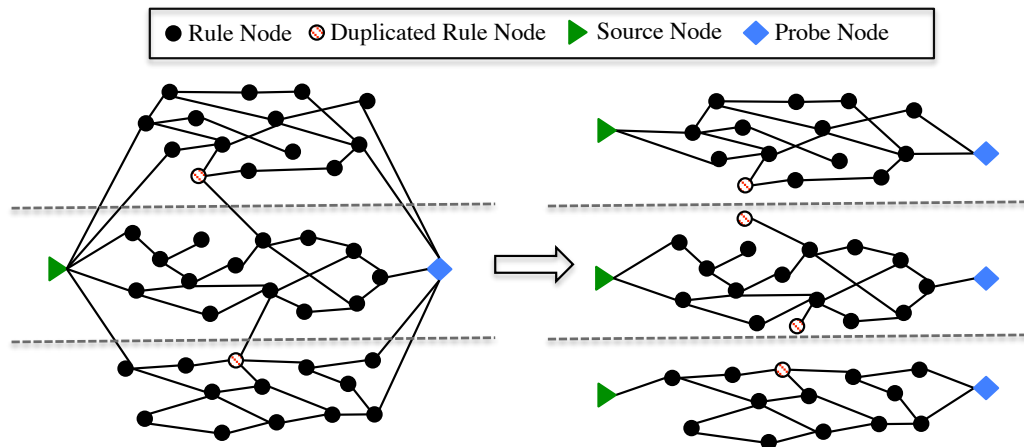


Figure 4.8: Clustering the rule dependency graph to enable parallelization. A typical rule dependency graph consists of clusters of highly dependent rules corresponding to FECs in the network. There may be rules whose dependency edges cross clusters. By replicating those rules, we can create clusters without dependencies and run each cluster as an isolated NetPlumber instance running on a different machine.

machine. Finally, a collector can be used to gather the check results from every NetPlumber instance and produce the final result.

One might expect to parallelize based on switches, that is, each NetPlumber instance creates a rule dependency graph for a subset of switches in the network (*vertical distribution*). This can address the memory bottleneck, but may not improve performance, as the NetPlumber instances can depend on each other. In the worst case, an instance may not be able to start its job unless the previous instance is complete. This technique can also require considerable communication between instances.

A key observation is that in every practical network we have seen, the rule dependency graph looks like Figure 5.5: There are clusters of highly dependent rules with very few dependencies between rules in different clusters. This is caused by forwarding equivalence classes (FECs) that are routed end-to-end in the network with possible aggregation. The rules belonging to a forwarding equivalence class have a high degree of dependency among each other. For example, 10.1.0.0/16 subnet traffic might be a FEC in a network. There might be rules that further divide this FEC into smaller subnets (such as 10.1.1.0/24, 10.1.2.0/24), but there are very few rules

outside this range that have any interaction with rules in this FEC (an exception is the default 0.0.0.0/0 rule).

Our distributed implementation of NetPlumber is based on this observation. Each instance of NetPlumber is responsible for checking a subset of rules that belong to one cluster (i.e., a FEC). Rules that belong to more than one cluster will be replicated on all of the instances with which they interact (see Figure 5.5). Probe nodes are replicated on all instances to ensure global verification. The final probe result is the aggregate of results generated by all the probes—that is, all probe nodes should meet their constraints in order for the constraint to be verified. The instances do not depend on each other and can run in parallel. The final result will be ready after the last instance completes its job.

The run time of distributed NetPlumber, running on n instances for a single rule update, is $O(m_{avg}(r/n + spd/m))$, where m is the number of times that the rule is replicated and m_{avg} is the average replication factor for all rules. This is because on each replica, the size of tables are $O(m_{avg}r/n)$ and the number of pipes to a rule that is replicated m times is $O(m_{avg}p/m)$. Note that if we increase n too much, most rules will be replicated across many instances ($m, m_{avg} \rightarrow n$) and the additional parallelism will not add any benefit.

How should we cluster rules? Graph clustering is hard in general; however, for IP networks, we generated natural clusters heuristically. We start by creating two clusters based on the IP address of our network. For example, if the host IP addresses in the network belong to subnet 10.1.0.0/16, create two clusters, one for rules that match this subnet, and one for the rest (i.e., 10.1.0.0/16 and 0.0.0.0/0 - 10.1.0.0/16 subnets). Next, divide the first cluster into two clusters based on bit 17 of the destination IP address. If one of the resulting clusters is much larger than the other, divide the larger cluster based on the next bit in the IP destination address. If two clusters are roughly the same size, divide both clusters further. This process continues until division does not reduce cluster size further (because of replication) or the specified number of clusters is reached.

Note that while we originally introduced the rule dependency graph to facilitate incremental computation, it also allows us to decompose the computation much more effectively than the naive decomposition by physical nodes.

4.4 Implementation

NetPlumber is implemented in C++ using Hassel's foundation layer. The source code is available at [24]. Figure 4.9 shows a simple block diagram of our implementation of NetPlumber and its dependency on Hassel. As shown in the figure, the two systems share the foundation layer, that is, the wildcard expression and header space objects, but they create different models of the network and policies: Hassel uses transfer function while NetPlumber uses rule, probe, and source nodes and flowexps language. Also, the two systems have different ways of checking policies and invariants: Hassel provides the basic functionality for checking reachability. Custom policy checks can be implemented by invoking the basic reachability function and writing extra code to check the specific policy on the result of the reachability check. On the other hand, NetPlumber uses the rule dependency graph and the flow routing techniques described Section 4.1 as a unified way to run all of the checks.

The NetPlumber management layer is the object that manages and controls different nodes and the rule dependency graph. It provides the following API for updating the NetPlumber state and checking policies:

- *Add Table*. Adds a new table to NetPlumber with a list of input/output ports belonging to the table. For a single-table box, this is equivalent to adding the box to the NetPlumber.
- *Remove Table*. Removes a table with a given table ID.
- *Add New Rule*. Adds a rule with the specified matching wildcard expression, input ports, action, and output port and returns an ID for the created table.
- *Delete Rule*. Deletes a rule with the given ID.
- *Add Link*. Adds a unidirectional link from a source port to a destination port.

- *Remove Link.* Removes a unidirectional link from a source port to a destination port.
- *Add Probe Node.* Attaches a probe node to a particular port in the rule dependency graph. The probe is configured at its creation time by a filter flowexp, a test flowexp, a universal/existential mode bit, and a callback function to be invoked upon violation. Returns an ID for the probe.
- *Remove Probe Node.* Removes a probe node with a given ID.
- *Add Source/Sink Node.* Attaches a source/sink node to a given port in the rule dependency graph. The source/sink node is configured with the header space of the flow it should generate.
- *Remove Source/Sink Node.* Removes a source/sink node with a given ID.
- *Set Loop Detection Callback Function.* Sets a global callback function for the loop detection check. The callback function will receive all information about the loop, that is, port, h_{ret} and the loop transfer functions.
- *Set Black Hole Detection Callback Function.* Sets a global callback function for black holes detection.

NetPlumber gets its live stream of network state updates through its built-in JSON server. We have also provided a JSON client that can feed a list of state updates to the server.

4.5 Evaluation

In this section, we evaluate the performance and functionality of our C++ based implementation of NetPlumber on three real world networks: the Google interdatacenter WAN, Stanford's backbone network, and the Internet 2 nationwide network. All of the experiments are run on Ubuntu machines, with six cores, hyper-threaded Intel Xeon processors, a 12 MB L2-cache, and 12 GB of DRAM.

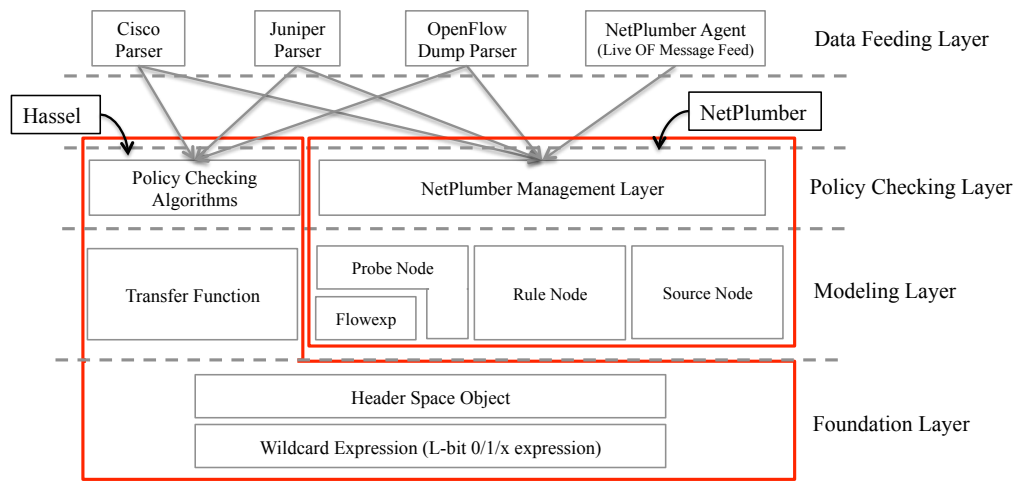


Figure 4.9: NetPlumber software block diagram and its dependency on Hassel.

To feed the snapshot data from these networks into NetPlumber, we used Hassel’s Cisco IOS and Juniper Junos parser and implemented an OpenFlow parser to parse OpenFlow table dumps in protobuf [43] format. We used a json-rpc-based client to feed this data into NetPlumber. NetPlumber has json-rpc server capability and can receive and process updates from a remote source.

4.5.1 Our Data Set

In our evaluation of NetPlumber, the following data sets are used:

Google WAN: This is a software-defined network, consisting of OpenFlow switches distributed across the globe. It connects Google data centers world-wide. Figure 4.10 shows the topology of this network. Overall, there are more than 143,000 OpenFlow rules installed in these switches. Google WAN is one of the largest SDNs deployed today; therefore, we stress-test NetPlumber on this network to evaluate its scalability.

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 [19], with the exception of ACL rules, which are removed for security reasons. We only use the IPv4 network of Internet 2 in this paper.

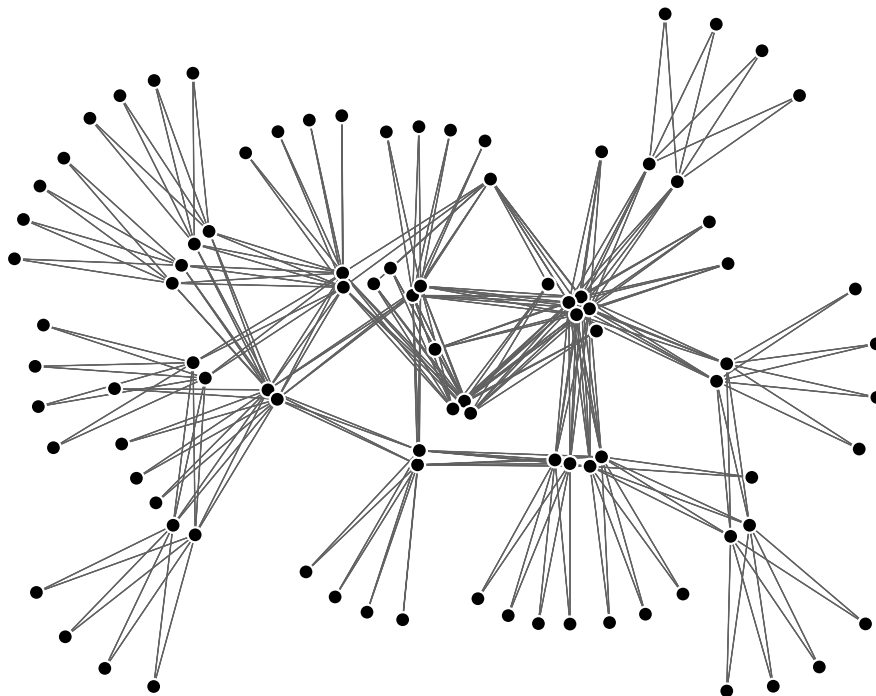


Figure 4.10: Google inter-datacenter WAN network—July 2012.

Stanford University Backbone Network. This is the same data set used in Section 3.5.1.

4.5.2 All-pair Connectivity of Google WAN

As an internal inter-datacenter WAN for Google, the main goal of Google WAN is to ensure connectivity between different data centers at all times. Therefore, in our first experiment, we checked for the *all-pair connectivity* policy between all 52 leaf nodes (i.e., data center switches). We began by loading a snapshot of every OpenFlow rule for Google WAN—taken at the end of July 2012—into NetPlumber. NetPlumber created the initial rule dependency graph in 33.39 seconds (an average per-rule runtime of $230\mu s$). We then attach one probe and one source node at each leaf of the network and set up the probes to look for one flow from each of the sources. If no probes fire, then all data centers can reach each other. The initial all-pair connectivity test took around 60 seconds. Note that the above run times

are for the *one-time initialization* of NetPlumber. Once NetPlumber is initialized, it can incrementally update check results much faster when changes occur. Note that the all-pair reachability check in Google WAN corresponds to 52^2 or more than 2600 pair-wise reachability checks.

Next, we used a second snapshot taken six weeks later. We found the difference of the two snapshots and applied them to simulate incremental updates. The difference includes both the insertion and deletion of rules. Since we did not have timing information for the individual updates, we knew the *set* of updates in the difference but not the *sequence* of updates. So we simulated two different orders. In the first ordering, we applied all of the rule insertions before the rule deletions. In the second ordering, we applied all deletions before all insertions.

As expected, the all-pair connectivity policy was maintained during the first ordering of update events because new reachable paths are created before old reachable paths are removed. However the second ordering resulted in violations of the all-pair connectivity constraint *during* the rule deletion phase. Of course, this does not mean that the actual Google WAN had reachability problems because the order we simulated is unlikely to have been the actual order of updates. At the end of both orderings, the all-pair connectivity constraint was met.

NetPlumber was able to check the compliance of each insertion or deletion rule in an average time of 5.74ms with a median time of 0.77ms. The average run time is much higher than the median because there are a few rules whose insertion and deletion take a long time (about 1 second). These are the default forwarding rules that have a large number of pipes and dependencies from/to other rules. Inserting and deleting default rules require significant changes to the rule dependency graph and routing of flows. The solid line in Figure 4.11 shows the run time CDF for all of the updates.

To test the performance of distributed NetPlumber we repeated the same experiment in distributed mode. We ran a simulation⁸ of NetPlumber on 2–8 machines and measured the update times (dashed lines in Figure 4.11). Table 4.1 summarizes the

⁸To simulate, we run the the instances in serial on the same machine and collected the results from each run. For each rule insertion/deletion, we reported the run time as the maximum run time across all instances, because the overall job will only be done when the last instance is complete.

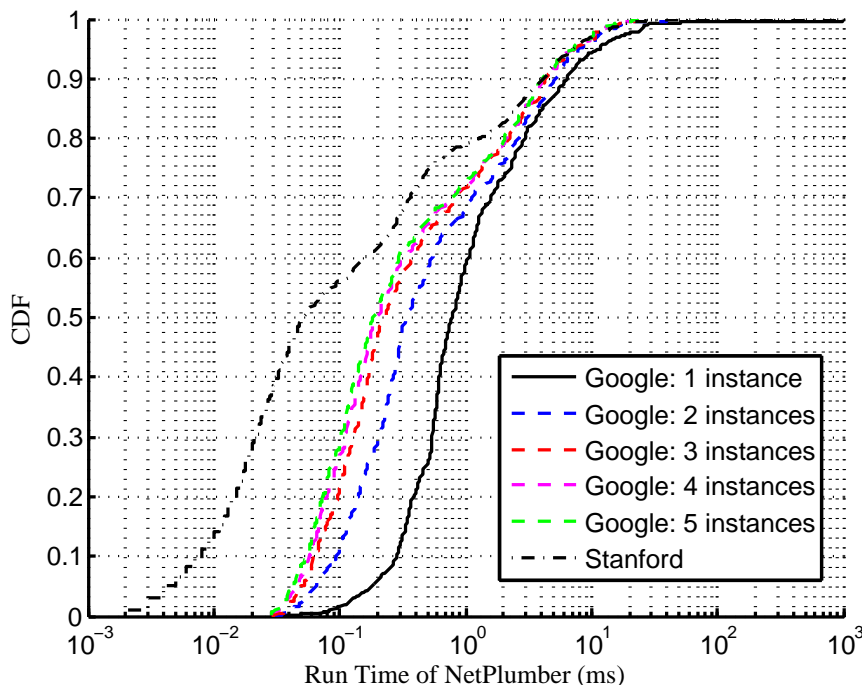


Figure 4.11: CDF of the run time of NetPlumber per update, when checking the all-pair reachability constraint in Google WAN with 1-5 instances and in Stanford backbone with a single instance.

#instances:	1	2	3	4	5	8
median (ms)	0.77	0.35	0.23	0.2	0.185	0.180
mean (ms)	5.74	1.81	1.52	1.44	1.39	1.32

Table 4.1: Average and median run time of the distributed NetPlumber, checking all-pair connectivity policy on Google WAN.

mean and median run times. This suggests that most of the benefits of distribution are achieved when the number of instances is 5 because in the rule dependency graph for the Google WAN, there are about 5 groups of FECs whose rules do not influence each other. Trying to put these rules in more than 5 clusters will result in duplication of rules, and the added benefit will be minimal.

4.5.3 Checking Policy in Stanford network

Unlike the Google WAN, there are a number of reachability restrictions enforced in the Stanford network by different ACLs. Examples of such policies include isolating machines belonging to a particular research group from the rest of the network, or limiting the type of traffic that can be sent to a server IP address. For example, all TCP traffic to the computer science department is blocked except for those destined to specific IP addresses or TCP port numbers. In addition, there is a global reachability goal that every edge router be able to communicate with the outside world via the uplink of a specified router called `bbra_rtr`. Finally, due to the topology of the network, the network administrators desired that all paths between any two edge ports be no longer than 3 hops long to minimize network latency.

In this experiment, we test all of these policies. To do so, we connect 16 source nodes, one to each router in the rule dependency graph. To test the maximum-three-hop constraint, we connected 14 probe nodes, one to each operational zone (OZ) router. We also placed a probe node at a router called `yoza_rtr` to check reachability policies at the computer science department. NetPlumber took 0.5 second to create the initial rule dependency graph and 36 seconds to generate the initial check results. We found no violation of the computer science department’s reachability policies. However, NetPlumber did detect a dozen un-optimized routes, whose paths take 4 hops instead of 3. We also found 10 loops similar to those reported in Section 3.5.1.

We then tested the per-update run time of NetPlumber by randomly selecting 7% of rules in the Stanford network, deleting them and then adding them back. Figure 4.11 shows the distribution of the per-update run time. Here, the median runtime is $50\mu s$ and the mean is $2.34ms$. The huge difference between the mean and the median is due to a few outlier default rules that take a long time to get inserted and deleted into NetPlumber.

4.5.4 Performance Benchmarking

The previous two experiments demonstrated the scalability and functionality of NetPlumber when checking the actual policies and invariants of two production networks.

Network:	Google		Stanford		Internet 2	
Run Time	mean	median	mean	median	mean	median
Add Rule (ms)	0.28	0.23	0.2	0.065	0.53	0.52
Add Link (ms)	1510	1370	3020	2120	4760	2320

Table 4.2: Average and median run time of NetPlumber for a single rule and link update when only one source node is connected to NetPlumber.

However, the performance of NetPlumber depends on s , the number of sources in the network, which is a direct consequence of the quantity and type of policies specified by each network. Thus, it seems useful to have a metric that is per source node and even per policy, so we can extrapolate how run time will change as we add more independent policies, each of which require adding a new source node.⁹ We provide such a unit run time benchmark for NetPlumber running on all three data sets: Google WAN, Stanford, and Internet 2.

To obtain this benchmark, we connect a single source node at one of the edge ports in the rule dependency graph of each of our 3 networks. Then, we load NetPlumber with 90% of the rules selected uniformly at random. Finally, we add the last 10% and measure the update time. We then repeated the same experiment by choosing links in the network that are in the path of injected flows, deleting them, and then adding them back and measuring the time to incorporate the added link. The results are summarized in Table 4.2. As the table suggests, link-up events take much longer (seconds) to incorporate. This is in fact expected and acceptable because when a link is added, a potentially large number of pipes will be created, which significantly changes routing of flows. Fortunately, since the link-up/down event should be rare, this run time appears acceptable.

4.6 Discussion

In this section, I first discuss some other contexts in which we can use NetPlumber and then review the limitations of NetPlumber.

⁹In contrast, dependent policies can be checked using a single source node.

4.6.1 Other Use Cases

Deployment in conventional networks: Conceptually, NetPlumber can be used with conventional networks as long as a notification mechanism is implemented to acquire updated state information. One way to do this is through SNMP traps; every time a forwarding entry or link state changes, NetPlumber receives a notification. We can also implement a polling mechanism to obtain a new snapshot of the forwarding states and apply the difference between consecutive snapshots as updates to NetPlumber. The main drawback of these mechanisms is the resource consumption at the switches due to state collection. Also, in the polling approach, we may miss the transient violations that last less than the polling frequency.

Deployment in multi-tenant data centers: In multi-tenant data centers, the set of policies might change dynamically upon VM migration. NetPlumber can handle dynamic policy changes easily, but it needs to be set up appropriately in advance. To handle dynamic policy changes, we attach a source node to every edge port in the rule dependency graph (as we did in the case of Google WAN). This way, we can update policies by only changing the location and test condition of probe nodes. The idea is that flow routing, which is the most expensive operation in NetPlumber, is pre-computed for all possible input flows. The policy to be checked on these flows is updated on demand as VMs migrate. The update will be fast as long as the structure of the rule dependency graph and routing of flows does not change significantly.

Using NetPlumber as the query engine of other tools: NetPlumber is useful as a foundation that goes beyond static policy checking. It can be used alongside other tools to provide real-time answers to a set of queries of interest. For example, automatic test packet generation (ATPG), presented in Chapter 5 and [56], is a framework that generates a test suite for a given network that achieves maximum coverage with the minimum number of test packets. At its core, ATPG computes reachability between test terminals and uses the reachability results in crafting the test packets. NetPlumber can provide real-time updates to the reachability results and feed that to the ATPG framework, which in turn can update the test suite.

Also, NDB [17] may benefit from NetPlumber. Like GDB, NDB allows “break points” to be set in the network when a specified condition is met. When packets matching a specified condition is observed in the network, NDB gives a complete trace of the packet, including its path, header at each hop, and the encountered switch forwarding table. To achieve this goal, NDB adds a “postcard generating action” to each rule that captures and sends headers of matching packets (called postcards) to a central database. The current implementation of NDB adds postcard action to every rule and generates postcards for all of the packets in the network; then, a filtering system filters the postcards of interests. To make the system more efficient, NDB could install postcard action for only those rules that the packets of interest would match. Here, NetPlumber can be used as a query engine to detect those rules that will process packets of interest.

While these are only two examples, the ability to incrementally and quickly analyze header spaces and answer questions about flows in the network will be a fundamental building block for network verification tools going forward.

4.6.2 Limitations

NetPlumber, like Hassel, relies on reading the state of network devices and therefore cannot model middleboxes with dynamic state. To handle such dynamic boxes, the notion of “flow” should be extended to include other kinds of states beyond header and port. Another limitation of NetPlumber is its greater processing time for verifying link updates. As a result, it is not suitable for networks with a high rate of link-up/down events such as energy-proportional networks.

4.7 Related Work

Recent work on network verification, debugging, and troubleshooting, especially in SDNs, focuses on the following directions.

Programming foundations: The goal of these works is to provide high-level programming language abstractions for SDNs that make programming networks easier

and less error-prone. For example, Frenetic [14] is a SQL-like, high-level programming language for OpenFlow networks that automatically supports program composition and [44] enables per-packet and per-flow consistency during network updates. While high-level programming languages reduce the risk of errors, they will not eliminate them. Besides, they cannot prevent violations that resulted from a hardware failure. Therefore they cannot fill the need for verification and debugging tools.

Offline checking: Offline checking tools verify the correctness of the control plane or data plane offline. For example, NICE [6] applies model checking techniques to find bugs in OpenFlow control programs and verify the correctness of the control plane software. Hassel [23] is a snapshot-based tool that checks data plane correctness against network policies and invariants. Anteater [36] uses boolean expressions and SAT solvers for network modeling and performs similar checks to Hassel. However, offline checking cannot prevent bugs from damaging the network until the periodic check runs.

Online monitoring: The goal of online monitoring tools is to monitor, test or troubleshoot networks in run time. For example, OFRewind [51] captures and reproduces the sequence of problematic OpenFlow command sequences in order to use them for future diagnosis. ATPG [56] systematically generates test packets against router configurations, and monitors network health by periodically sending these tests packets. NDB [17] is a network debugger that gives a trace of network events that leads to a failure. These tools complement but do not replace the need for real-time policy checking, as they monitor the current state of network or capture faulty states. The goal of real time policy checking is to verify network states before they are installed.

Online checking: NetPlumber is an example of tools that fall in this category in that it verifies the stream of network state changes against network policy in real time. VeriFlow [25] is the work most closely related to NetPlumber. VeriFlow also verifies the compliance of network updates with specified policies in real time. It uses a *trie structure* to search rules based on equivalence classes (ECs), and upon an update, determines the affected ECs and updates the forwarding graph for that class. This in turn triggers a rechecking of affected policies. NetPlumber and VeriFlow offer

similar run-time performance. While both systems support verification of forwarding actions, NetPlumber additionally can verify arbitrary header modifications, including rewriting and encapsulation, is protocol-independent, and can handle wildcard bits at arbitrary locations in the match pattern of a rule.

4.8 Summary

This chapter introduces NetPlumber as a real-time policy checker for networks. Unlike Hassel, which checks periodic snapshots of the network, NetPlumber is fast enough to validate every update in real time. Users can express a wide range of policies to be checked using an extensible regular expression-like language, called flowexp. Since flowexp might be too low-level for administrators to use, we implemented a higher level policy language (inspired by FML) in Prolog.

The fundamental idea of the rule dependency graph benefits us in three ways. First, it allows incremental computation by allowing only the (smaller) dependency subgraph to be traversed when a new rule is added. Second, it naturally leads us to a flexible way to express and check policies by placement and configuration of source and probe nodes in the dependency graph. Third, clustering the graph to minimize inter-cluster edges provides a powerful way to parallelize computation.

Chapter 5

Automatic Test Packet Generation

Hassel and NetPlumber both verify the compliance of the forwarding state of a network with its policy. An orthogonal check is to ensure that the actual behavior of networking boxes complies with their forwarding state; a mismatch may result in a violation of some network policy. My goal in this chapter¹ is to design a testing framework for detecting mismatch between the expected network behavior, based on its forwarding state, and its observed behavior, based on the actual forwarding of packets in the network.

Figure 5.1 is a simplified view of network state. At the bottom of the figure is the forwarding state used to forward each packet, consisting of the L2 and L3 forwarding information base (FIB), access control lists, etc. The forwarding state is written by the control plane (which can be local, or as in the SDN model, remote), and should correctly implement the network administrator’s policy. We can think of the controller compiling the policy (A in Figure 5.1) into the device-specific forwarding state (B), which in turn determines the forwarding behavior of each packet (C). To ensure the network behaves as designed, all three steps should remain consistent at all times, that is, $A = B = C$. In addition, the topology, shown to the bottom right in the figure, should also satisfy a set of liveness properties L . Minimally, L requires that sufficient links and nodes are working; if the control plane specifies that

¹James Hongyi Zeng has contributed equally to the content of this chapter.

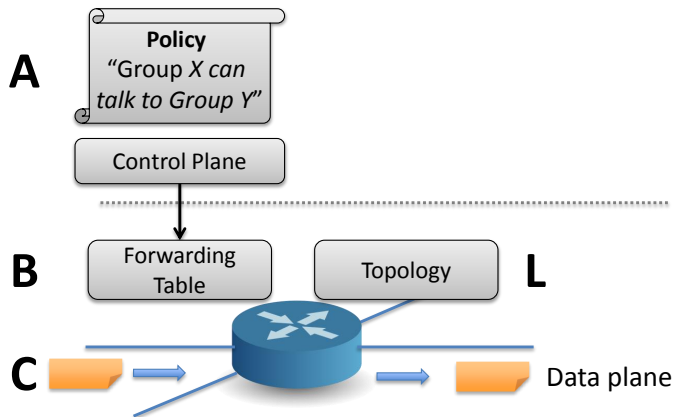


Figure 5.1: Condition for correctness of a network: policy = forwarding state = actual behavior.

a laptop can access a server, the desired outcome can fail if links fail. L can also specify performance guarantees that detect flaky links.

Chapter 3 and Chapter 4 of this dissertation have proposed tools to check that $A = B$, enforcing consistency between the *policy* and the *forwarding state*. While these approaches can find (or prevent) software logic errors in the control plane, they are *not* designed to identify liveness failures caused by failed links and routers, bugs caused by faulty router hardware or software, or performance problems caused by network congestion. Such failures require checking for L and whether $B = C$.

This chapter introduces the Automatic Test Packet Generation (ATPG) framework, which *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. It can also automatically generate packets to test *performance* assertions such as packet latency. 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—that is, at least one test packet matches

and is forwarded by every rule. 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.

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, and it 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 have tested ATPG on two real world data sets—the backbone networks of Stanford University and Internet2, representing an enterprise network and a nationwide ISP respectively. 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, which has 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 is even smaller—around 50 packets, which allows proactive liveness testing every millisecond using 1% of network bandwidth.

5.1 ATPG System

Based on the network model provided by the header space analysis framework, 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.

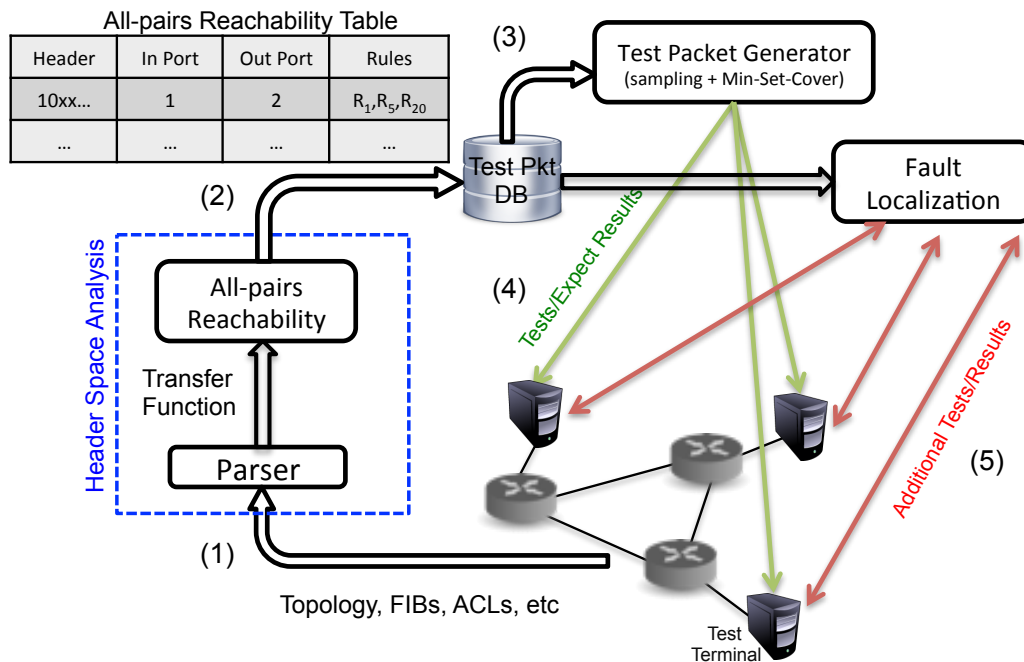


Figure 5.2: ATPG system block diagram.

Figure 5.2 is a block diagram of the ATPG system. The system first collects all the forwarding states from the network (step 1 in Figure 5.2). This usually involves reading the FIBs, ACLs, and config files, as well as obtaining the topology. ATPG uses header space analysis to compute reachability between all the test terminals (step 2). The result is then used by the test packet selection algorithm to compute a minimal set of test packets that can test all rules (step 3). These packets will be sent periodically by the test terminals (step 4). If an error is detected, the fault localization algorithm is invoked to narrow down the possible causes of the error (step 5).

5.1.1 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 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 transfer functions, $\{T_1, \dots, T_n\}$, and topology transfer 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 uses the reachability algorithm described in Figure 3.1 from every test terminal: on every terminal port, an all-wildcard header 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-wildcard header, the VLAN tag bits are set to A . As each packet traverses the network using the network transfer function, the set of all matched rules are recorded in *history*. Doing this for all pairs of terminal ports generates an *all-pairs reachability table* as shown in Table 5.1. For each row, the header column is a wildcard expression representing the equivalence 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.

Header	Ingress Port	Egress Port	Rule History
h_1	p_{11}	p_{12}	$[r_{11}, r_{12}, \dots]$
h_2	p_{21}	p_{22}	$[r_{21}, r_{22}, \dots]$
...
h_n	p_{n1}	p_{n2}	$[r_{n1}, r_{n2}, \dots]$

Table 5.1: All-pairs reachability table: all possible headers from every terminal to every other terminal, along with the rules they exercise.

	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 AC
(p_6)	dst_ip=10.2/16, tcp=80	P_C	P_B	r_{C1}, r_{B3}, r_{B4} , link BC

Table 5.2: Test packets for the example network depicted in Figure 5.3. p_6 is stored as a reserved packet.

Figure 5.3 shows a simple example network and Table 5.2 is the corresponding all-pairs reachability table. For example, an all-wildcard 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 5.2.

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. This scheme only detects faults for which all packets covered by the same rule experience the same fault (e.g., a link failure). At the other extreme, if we wish to detect faults specific to a header, then we need to select every header in every class. We discuss these issues and our fault model in Section 5.1.2.

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

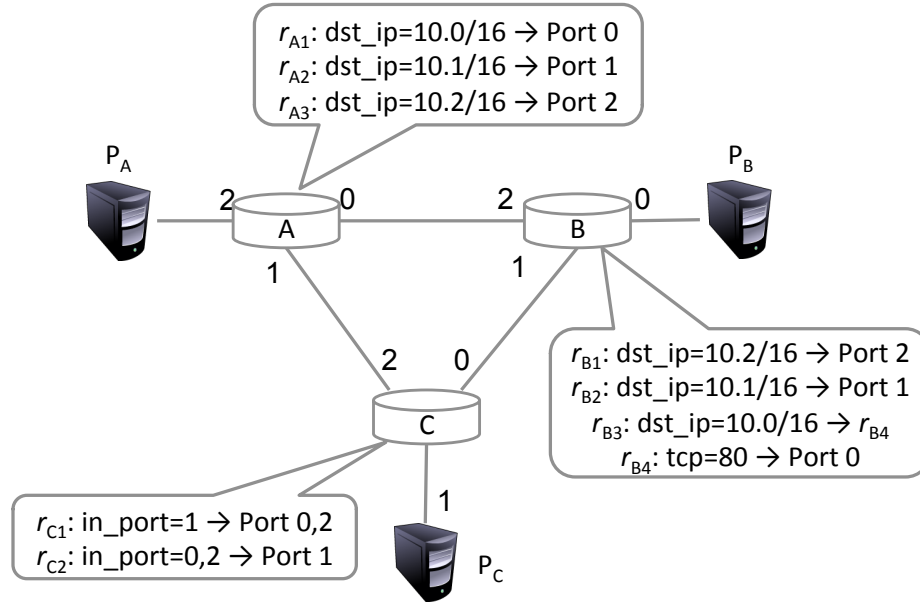


Figure 5.3: Example topology with three switches used for finding test packets.

to cover all links (for liveness only) or all router queues (for performance only). This is the classical Min-Set-Cover problem [50]. While NP-Complete, 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 5.2, $\{p_1, p_2, p_3, p_4, p_5\}$ are regular test packets and $\{p_6\}$ is a reserved test packet. Reserved test packets are useful for fault localization (Section 5.1.2).

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 [7].

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 $O(DR^2)$ as shown in Section 3.1.3, and this computation is repeated for each test terminal.

5.1.2 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 transfer 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 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 miswiring. On the other hand, match faults are harder to detect because they only affect *some* packets matching the rule: for example, when a rule matches a header it should not, or when a rule misses a header it should match. 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. 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 instance, 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. This leads to our fault propagation assumption:

Assumption 1 (Fault propagation) *A test packet that is received successfully by the test terminal indicates that all the rules, queues, and links that are exercised*

by the test packet are behaving correctly. More formally, $R(pk) = 1$ 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 test packets and their corresponding test results, $[(pk_0, R(pk_0)), (pk_1, R(pk_1)), \dots]$, find all rules, r , that causes the failure of a test packet. More formally, find all r that satisfy $\exists pk_i, R(pk_i, r) = 0$.*

We solve this problem opportunistically and in steps.

Step 1: Consider the test results obtained 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 . Any rule which is not part of a passing test is a failure suspect, therefore $F - P$ is a set of *suspect rules*.

Step 2: Next, ATPG 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 satisfies this condition and exercises rule r from 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.

False positives: If our fault propagation assumption holds, the fault localization method will not miss any faults, and therefore will have no *false negatives*. However, this method may introduce false positives for some rules left in the suspect set at

the end of step 3, that is, one or more rules in the suspect set may in fact behave correctly.

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

5.2 Use Cases

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

5.2.1 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 (i.e., a rule in the topology transfer function modeling the behavior of a unidirectional link) 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 were 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

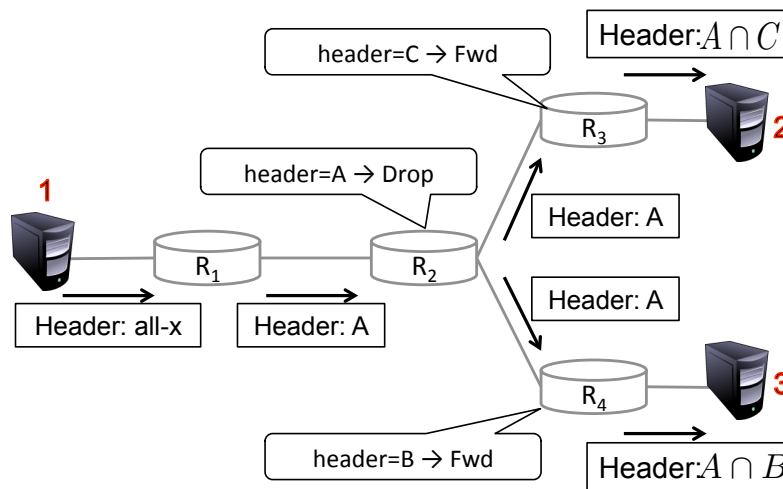


Figure 5.4: Generating packets to test drop rules by “flipping” the drop rule to a broadcast rule in the analysis.

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 5.4. To test the drop rule in R_2 , we inject the all-wildcard test packet at Terminal 1. If the drop rule were 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 R_2 , 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 R_2 ’s drop rule.

5.2.2 Performance Testing

We can also use ATPG to monitor the performance of links, queues and QoS classes in the network, and even monitor Service Level Agreements (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 desired.

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, 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, such as `iperf/netperf`, or more gentle approaches such as packet pairs and packet trains [28]. Suppose a manager specifies that the available bandwidth of a particular service class should not fall below a certain threshold; if it does, 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 are unaffected. If they are, fault localization can be used to pinpoint the problem.

5.3 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 [55].

5.3.1 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 [24] to construct switch and topology transfer 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.

5.3.2 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 look up test packets efficiently.

5.3.3 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 could 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 [39], 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.

5.4 Evaluation

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. Section 3.5.1 and Section 4.5.1 provide more details about these networks.

5.4.1 Test Packet Generation

We ran ATPG on a quad core Intel Core i7 CPU 3.2 GHz and 6 GB memory using 8 threads. For a given number of test terminals, we generated the minimum set of test packets needed to test all the reachable rules in the Stanford and Internet2 backbones. Table 5.3 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

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

Table 5.3: 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.

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 achieving 100% *link* coverage (instead of *rule* coverage) requires only 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 (which has been considered infeasible [42]) feasible for large networks.

Coverage is the ratio of the number of rules exercised to the total number of reachable rules. Our results show 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.

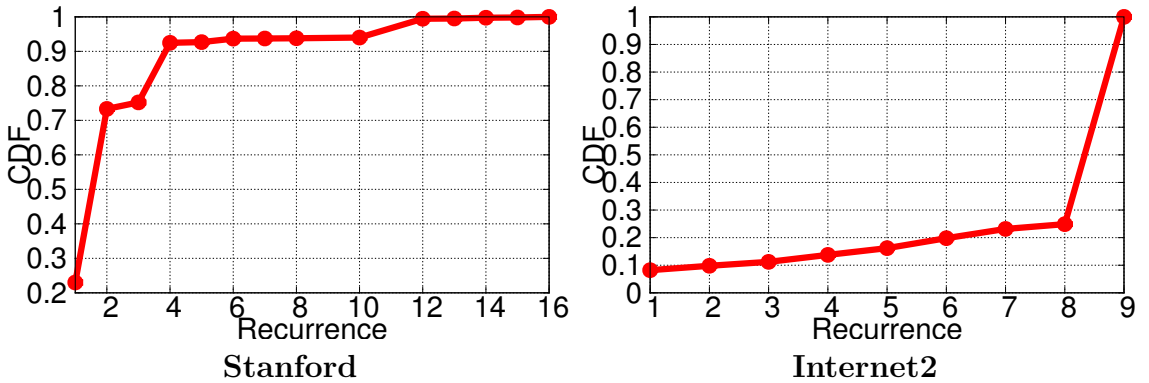


Figure 5.5: The cumulative distribution function of rule repetition, ignoring different action fields.

We can verify this structure by clustering rules in Stanford and Internet2 that match the same header patterns. Figure 5.5 shows the distribution of rule repetition in Stanford and Internet2. In both networks, 60%-70% of matching patterns appear in more than one router. We also find that this repetition is correlated to the network topology. In the Stanford backbone, which has a two-level hierarchy, matching patterns commonly appear in 2 (50.3%) or 4 (17.3%) routers, which represents the length of edge-to-Internet and edge-to-edge routes. In Internet2, 75.1% of all distinct rules are replicated 9 times, which is the number of routers in the topology.

5.4.2 Testing in an Emulated Network

To evaluate the network monitor and test agents, we replicated the Stanford backbone network in Mininet [30], a container-based network emulator. We used Open vSwitch

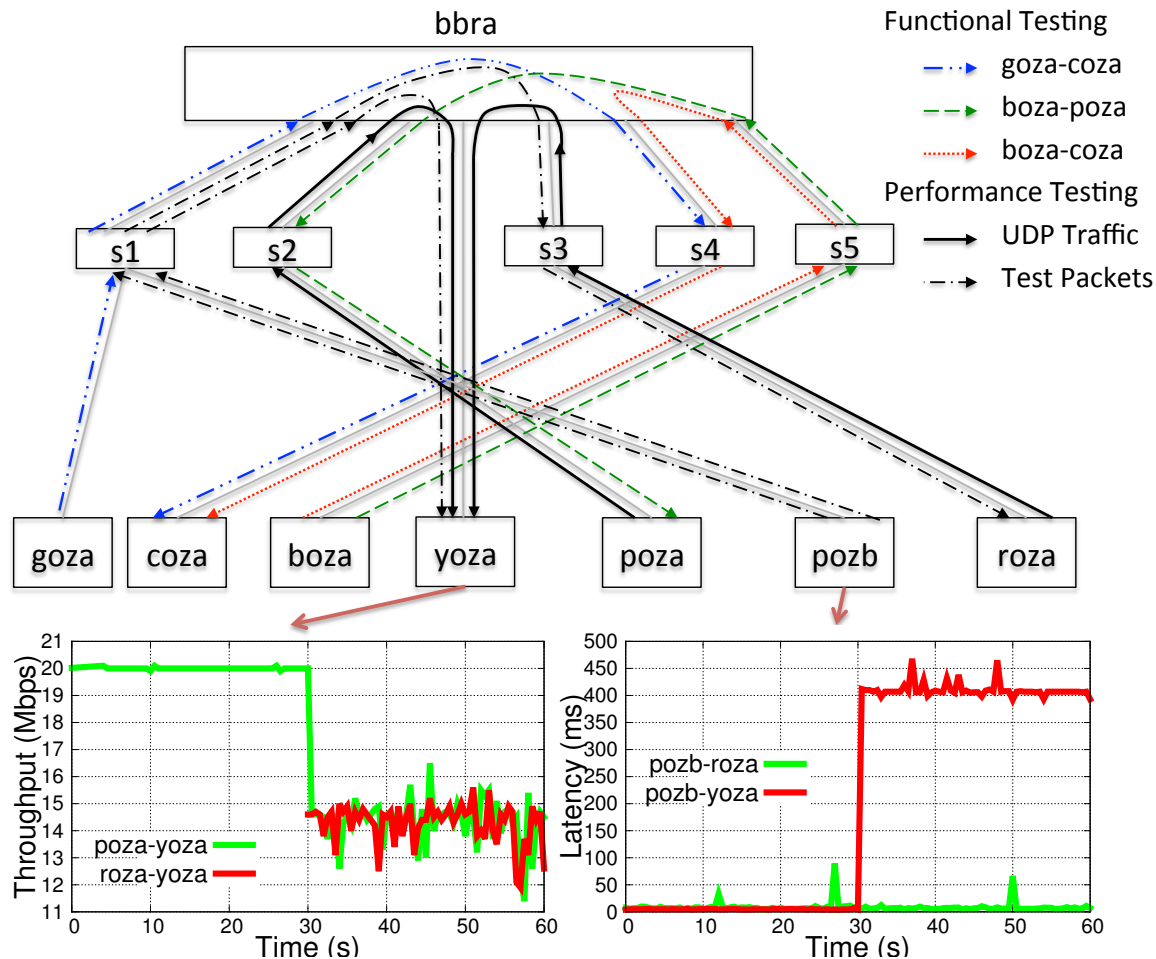


Figure 5.6: A portion of the Stanford backbone network showing the test packets used for functional and performance testing examples as described in Section 5.4.2.

(OVS) [41] 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 [39] rules and installed them in the OVS switches with Beacon [3]. We used emulated hosts to send and receive test packets generated by ATPG. Figure 5.6 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 *boza* to *coza* with $dst_ip = 172.20.10.33$ failed and were not received at *coza*. Table 5.4 shows two other test packets we used to localize and pinpoint the fault. These test packets, shown in Figure 5.6 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 5.1, 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.

Header	Ingress	Egress	Rule History	Result
$dst_ip = 172.20.10.33$	<i>goza</i>	<i>coza</i>	$[R_1^{goza}, L_{goza}^{S_1}, S_1, L_{S_1}^{bbra}, R_1^{bbra}, L_{bbra}^{S_4}, S_4, R_1^{coza}]$	Pass
$dst_ip = 172.20.10.33$	<i>boza</i>	<i>coza</i>	$[R_1^{boza}, L_{boza}^{S_5}, S_5, L_{S_5}^{bbra}, R_1^{bbra}, L_{bbra}^{S_4}, S_4, R_1^{coza}]$	Fail
$dst_ip = 171.67.222.65$	<i>boza</i>	<i>poza</i>	$[R_2^{boza}, L_{boza}^{S_5}, S_5, L_{S_5}^{bbra}, R_2^{bbra}, L_{bbra}^{S_2}, S_2, R_2^{poza}]$	Pass

Table 5.4: 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.

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 that the latency can be calculated with a single time-stamp and one-way communication².

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 5.6 shows the two UDP flows. After starting the second flow, the *bbra* – *yoza* queue was congested and the test packets going through this link experienced longer queuing delay. The bottom right graph

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

next to *poz**b* shows the latency experienced by two test packets, one from *poz**b* to *roza* and the other one from *poz**b* to *yoza*. At $t = 30s$, the *boz**b* – *yoza* test packet experiences much higher latency, correctly signaling congestion. Since these two test packets share the *boz**b* – 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 [20], 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 indicated that *boz**b* – *yoza* had an available bandwidth³ of 11.715 Mb/s, and *boz**b* – *roza* had an available bandwidth of 19.935 Mb/s, while the other (idle) terminals reported 30.60 Mb/s. Using the same argument as before, ATPG concluded that the *bbra* – *yoza* link was 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 sent the previously “failed” test packet, *poz**b* – *yoza*, on both the high and low-priority queues by changing the priority field in the IP header.⁴ Figure 5.7 shows the result. We created congestion on *bbra* – *yoza* link in different ways by congesting the low and high-priority queues respectively. When the low-priority queue was congested (i.e., both UDP flows mapped to low-priority queues), only the low-priority test packets were affected and experienced longer delay. However, when the high-priority slice was congested, both the low and high-priority test packets experienced congestion and were delayed.

5.4.3 Testing in a Production Network

We deployed an experimental ATPG system in three 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

³All numbers are the average of 10 repeated measurements.

⁴The Stanford data set does not include the priority settings.

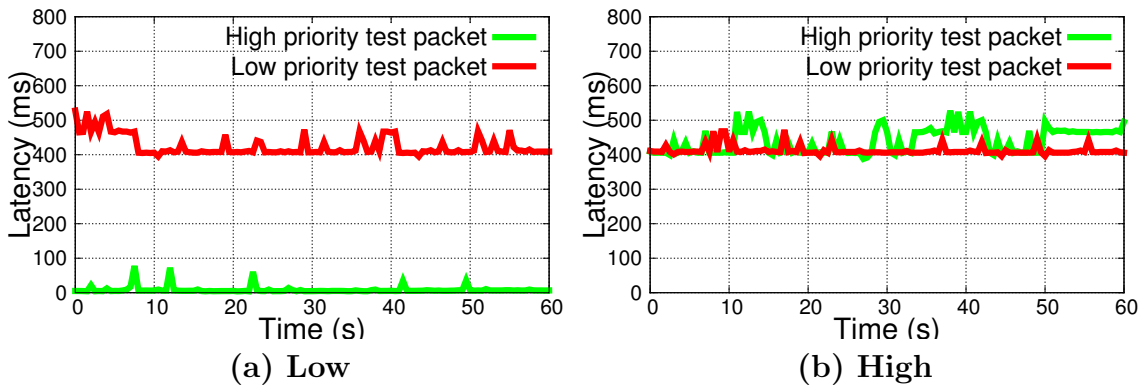


Figure 5.7: Priority testing: Latency measured by test agents when (a) low-priority or (b) high-priority slice is congested.

to the campus backbone. For test terminals, we utilized the 53 WiFi access points (running Linux) that were already installed throughout the buildings. This allowed us to achieve high coverage on switches and links. However, we could only run ATPG on essentially a layer 2 (bridged) network.

On October 1-10, 2012, the ATPG system was used for a 10-day `ping` experiment. Since the network configurations remained static during this period, instead of reading the configuration from the switches dynamically, we derived the network model based on the topology. In other words, for a layer 2 bridged network, it is easy to infer the forwarding entry in each switch for each MAC address without getting access to the forwarding tables in all 30 switches. We used only `ping` to generate test packets. `Ping` sufficed because in the subnetwork we tested there are no layer 3 rules or ACLs. Each test agent downloaded a list of `ping` targets from a central web server every 10 minutes, and conducted `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 5.8 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 used significantly fewer test packets. In fact, ATPG used 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

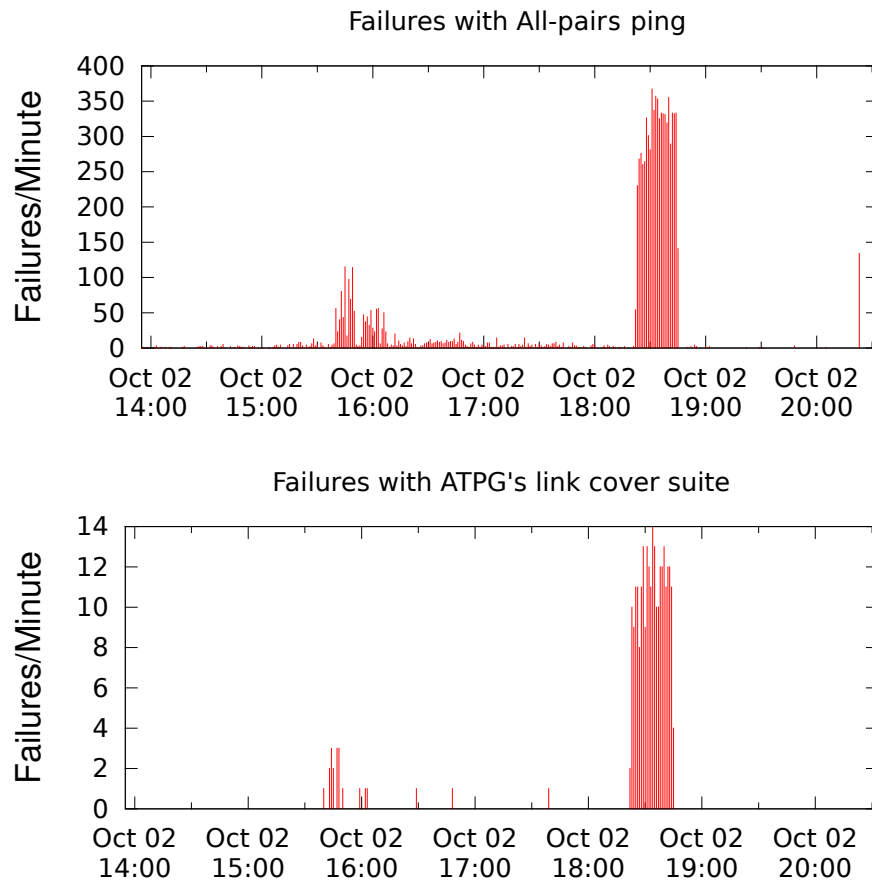


Figure 5.8: The Oct 2, 2012, production network outages, captured by the ATPG system, as seen from the lens of an inefficient cover (all-pairs, top picture) and an efficient minimum cover (bottom picture).

(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 occurred. 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.

5.5 Discussion

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 can reduce overhead by using NetPlumber to incrementally update the all-pair reachability results in real time. Test agents within terminals incur negligible overhead because they merely demultiplex test packets addressed to their IP address at a modest rate (e.g., 1 per millisecond) compared to the link speeds (> 1 Gbps) most modern CPUs are capable of receiving.

Limitations of ATPG framework: As with all testing methodologies, ATPG has its own limitations:

- *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.
- *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.
- *Invisible rules:* A failed rule can make a backup rule active, and as a result no changes may be observed by the test packets. This can happen when, despite a failure, a test packet is routed to the expected destination by other rules. In addition, an error in a backup rule cannot be detected in normal operation.

Another example is when two drop rules appear in a row: the failure of one rule is undetectable since the effect will be masked by the other rule.

- *Transient network states*: ATPG cannot uncover errors whose lifetime is shorter than the time between each round of tests. For example, congestion may disappear before an available bandwidth probing test concludes. Finer-grained test agents are needed to capture abnormalities of short duration.
- *Sampling*: ATPG uses sampling when generating test packets. As a result, ATPG can miss match faults since the error is not uniform across all matching headers. In the worst case (when only one header is in error), exhaustive testing is needed.

5.6 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 [6] attempts to exhaustively cover the code paths symbolically in controller applications with the help of simplified switch/host models. In the data plane, Ant eater [36] and Hassel can do static verification of network state against design policy. Recently, SOFT [27] 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 [8, 10, 11, 26, 34, 35, 37]. Recently, mining low-quality, unstructured data, such as router configurations and network tickets, has attracted interest [15, 31, 49]. By contrast, the primary contribution of ATPG is not fault localization, but determining a compact set of end-to-end test packets that can cover every link or every rule. Further, ATPG is not limited to liveness testing but can be applied to checking higher level properties such as performance.

There are many proposals to develop a measurement-friendly architecture for networks [12, 33, 40, 53]. 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 [5] 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.

5.7 Summary

This chapter introduces an automated and systematic approach for testing the actual behavior of networks called “Automatic Test Packet Generation” (ATPG). ATPG uses Hassel to read the configurations and forwarding tables of networking boxes and generate their device independent transfer function. It then uses the reachability algorithm of HSA to find all the forwarding equivalence classes (FECs) between test terminals along with the set of forwarding rules covered by each FEC. This is referred to as *all-pair reachability* of test terminals. The all-pair reachability results are then used to generate a minimum set of test packets to (minimally) exercise every link in the network or (maximally) exercise every rule in the network. Test packets are sent periodically, and detected failures trigger a separate mechanism to localize the fault. ATPG can detect both functional (e.g., incorrect firewall rule) and performance problems (e.g., congested queue).

We described our prototype ATPG implementation and results on two real-world data sets: Stanford University’s backbone network and Internet2. We found that a small number of test packets suffices to test all rules in these networks: For example 4000 packets can cover all rules in the Stanford backbone network, while 54 is enough to cover all links. Sending 4000 test packets 10 times per second consumes less than 1% of link capacity.

While many other industries such as ASIC and software design already have systematic tools for testing their systems, we hope that the ATPG framework plays a similar role for testing networks. 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 [1]. We hope network ATPG will be equally useful for automated dynamic testing of production networks.

Chapter 6

Conclusion

6.1 Summary of Dissertation

In this dissertation, I introduced Header Space Analysis (HSA) as a protocol-independent model for forwarding functionality of networks. HSA looks at packets as a flat sequence of 0s and 1s and models them as points in a $\{0, 1\}^L$ space, called the header space. When these packets are in the network, we need an extra dimension—the port ID dimension—to model their location: $\{0, 1\}^L \times \{1, \dots, p\}$. This new space—which shows the header bits and the location of a packet in the network—is called the network space. HSA reads the forwarding state of networking boxes and generates a unified model for the forwarding behavior of these boxes in the form of a transfer function. Transfer functions move packets from one point in the network space to another point (or set of points) in the same space.

HSA serves as a foundation for developing techniques and tools for network verification, testing, and debugging. In this dissertation, I described three use cases of HSA in network verification and testing:

1. **Offline Verification with Hassel:** Hassel (Header Space Library) implements the HSA framework and uses it for verifying network properties such as reachability of end hosts, predicates on the path of flows in the network, loop freedom and isolation of network slices. The checks are snapshot-based and any change in the network requires rerunning the checks from scratch.

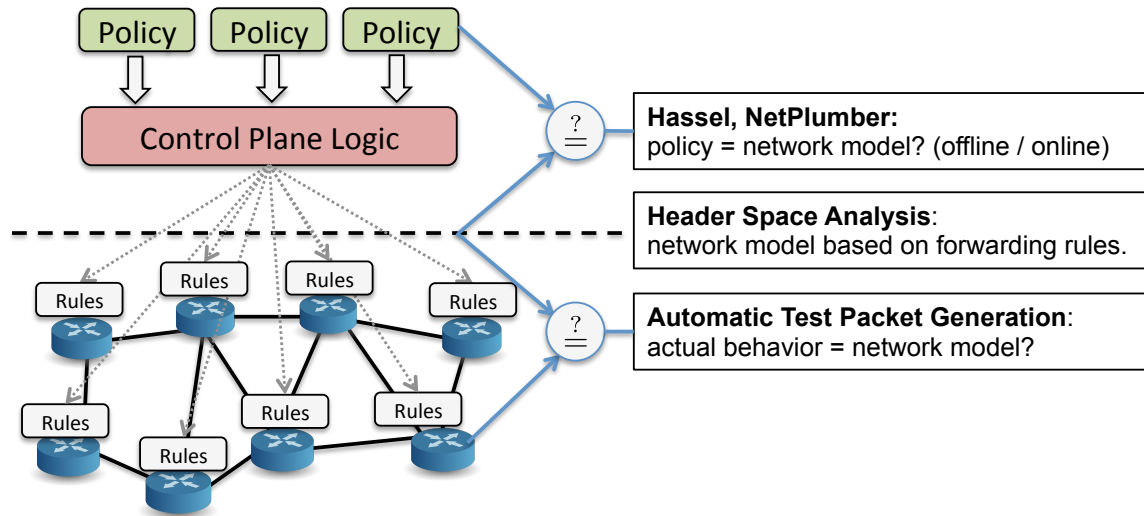


Figure 6.1: Relation between Hassel, NetPlumber, ATPG and the Header Space Analysis.

2. **Online Policy Checking with NetPlumber:** NetPlumber runs some of the Hassel checks online and incrementally on a stream of network state changes. The tool provides a flexible mechanism to express and test a wide range of path and header-based policies about the flows in the network.
3. **Online Testing and Monitoring with ATPG:** Automatic Test Packet Generation (ATPG) uses the reachability analysis of HSA to generate test packets for maximum coverage of rules, links, or queues in the network with the minimum number of test packets. It uses either Hassel (offline) or NetPlumber (online and incremental) to compute reachability between test terminals in the network.

Figure 6.1 shows the relation between HSA and these three tools: HSA provides a model of the network based on its forwarding state. Hassel and NetPlumber verify that the forwarding state, as modeled by HSA, implements the policies of the network and does not violate invariants such as loop freedom or black hole freedom. On the other hand, ATPG verifies that the actual network behavior, observed by its test

packets, matches the expected network behavior based on the HSA model of the forwarding state.

6.2 Contributions

This dissertation made the following contributions:

1. Introduced a protocol-independent framework for modeling and analyzing the forwarding behavior of networks (HSA).
2. Defined a set algebra on packet headers for finding intersection, difference, and complementation of flows and checking subset and equality conditions on flows.
3. Designed algorithms for determining reachability between end hosts, checking path predicates on the flows, finding forwarding loops along with loop repetition count, and checking isolation of network slices.
4. Implemented HSA and its checking algorithms as an open-source library as a foundation for other network verification and testing tools.
5. Designed a system that can verify policies on path and header of flows in real time (NetPlumber). The system also defined a flexible, regular expression-like language for expressing policies.
6. Implemented NetPlumber as an open-source tool for online network policy checking.
7. Designed ATPG, a framework for automatically and systematically generating test packets based on the forwarding state of the network to achieve maximum rule, queue, or link coverage with the minimum number of test packets.
8. Designed a fault localization scheme along with ATPG to localize a failure once a test packet experiences an error.
9. Implemented ATPG as an open-source tool for generating test packets and localizing faults.

10. Tested all of the techniques and tools developed in this dissertation on real networks such as Stanford University’s backbone network, Google inter-data center WAN, and Internet2 nationwide network.

6.3 Future Directions

Header Space Analysis—as a modeling framework—can be used as the foundation for building other tools for network verification, testing, debugging, or management. For example in [46], a technique called correspondence checking is used to check if the translation of policy by intermediate layers of SDN stack (e.g., logical view, physical view, etc.) is consistent. The technique uses HSA transfer function and all-pair reachability computation to compute the transfer function of a network between access links (links adjacent to hosts) and ensures that the transfer functions obtained from different layers are equivalent. Also, in [44], HSA models are used to design a mechanism for consistently updating a network from one state to another.

There are other applications in which HSA may be useful. Also, future work may focus on extending and improving the HSA framework itself. Below are a few examples of future directions for HSA.

1. HSA may be useful for inferring the policy of a network from its forwarding state.¹ This might help network managers to sanity check the inferred policy with their actual intention. HSA may come in handy here by providing the all-pair reachability results between access links and giving insights into how forwarding rules interact with each other through the rule dependency graph.
2. HSA may also be useful for detecting a sniffing attack, in which a compromised switch or router sends copies of certain type of traffic to a malicious destination.² Verification of forwarding state is a good first step here—it may detect the anomaly in routing of some flows. However, “traffic stealing” may happen

¹Thanks to George Varghese for mentioning this use case of HSA.

²Thanks to Hovav Shacham for mentioning this use case of HSA.

silently, and the forwarding rules may not reflect such traffic copying action. HSA may be useful here because of its ability to predict the value of rule counters at the core of the network and detect any mismatch with the expected values, which may signal that unexpected packets are being created. For example, given the counter value of each rule matching on the micro flows at the edge, HSA can find the path of those flows and predict the counter value of other matching rules in the other parts of network.³

3. HSA may be used to create a policy-to-rule translation tool for SDNs. The tool may translate policy into the necessary forwarding rules to be installed in networking boxes, taking into account the capabilities of boxes such as supported action types, number of forwarding tables, etc. In this dissertation, HSA was used to verify that the forwarding state correctly implements the network policy—this is a reverse application of HSA, to generate the forwarding state in a way to correctly implement the policy.
4. As discussed in Chapter 2, HSA cannot fully model stateful devices in which the behavior of the device may change based on external events such as previous packets or network load condition. This is because HSA only uses the header and port of the current packet as its input, thus, it is stateless. Making a stateful HSA framework, in which any necessary state, such as the previous k packets or the network traffic matrix, can be added to and handled by the model, is a useful extension to HSA.

6.4 Closing Remarks

Testing and verification is an essential part of all engineering disciplines, and networking is no exception. Networks are historically tested, verified, and debugged manually, using simple and ad hoc tools. Naturally, this leads to lots of configuration mistakes, security vulnerabilities, and network outages. As networks are getting

³HSA may not predict the counter value for rules matching on management traffic as that traffic is not coming to network from the edges.

bigger and more complicated, operating a network is becoming a very daunting task, requiring a lot of skill and experience. However, the recent attention of the networking community to rigorous, automated, and systematic approaches to network testing, verification, and management signals an imminent paradigm shift in the way that networks will be managed and operated in the future.

The very first—and probably the most important—step in that direction is creating the right model for the network: a model that is simple to use and captures the necessary information. Extracting simplicity is the key to creating a useful model for complex systems such as networks. Header space analysis was an attempt in that direction. I hope to convey the fact that networks—despite their apparent complexity—perform a very simple task: modifying and forwarding packets. And packets—despite carrying multiple protocols—are nothing but a series of bits. The techniques and tools introduced in this dissertation are all based on this simple model of a network, and as a result they are very simple to understand and to implement, and their correctness properties are easily provable. This is an especially noteworthy contribution given the fact that the problems that can be solved using these techniques have until now been considered very difficult.

Bibliography

- [1] Automatic Test Pattern Generation. http://en.wikipedia.org/wiki/Automatic_test_pattern_generation.
- [2] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: a novel firewall management toolkit. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*.
- [3] Beacon. <http://www.beaconcontroller.net/>.
- [4] S. Brown and Z. Vranesic. *Fundamentals of Digital Logic with Verilog Design*. McGraw-Hill Series in Electrical and Computer Engineering Series. McGraw-Hill, 2003.
- [5] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI 2008*.
- [6] M. Canini, D. Venzano, P. Perešini, D. Kostić, and J. Rexford. A NICE way to test openflow applications. In *Proceedings of NSDI 2012*.
- [7] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 1979.
- [8] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot. Netdiagnoser: troubleshooting network unreachabilities using end-to-end probes and routing data. In *Proceedings of ACM CoNEXT 2007*.

- [9] R. Draves, C. King, S. Venkatachary, and B. Zill. Constructing optimal ip routing tables. In *Proceedings of INFOCOM 1999*.
- [10] N. Duffield. Network tomography of binary network performance characteristics. *IEEE Transactions on Information Theory*, dec. 2006.
- [11] N. Duffield, F. Lo Presti, V. Paxson, and D. Towsley. Inferring link loss using striped unicast probes. In *Proceedings of INFOCOM 2001*.
- [12] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Transaction on Networking*, June 2001.
- [13] N. Feamster and H. Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of NSDI 2005*.
- [14] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. In *Proceedings of ACM SIGPLAN 2011*.
- [15] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011*.
- [16] S. Gutz, A. Story, C. Schlesinger, and N. Foster. Splendid isolation: a slice abstraction for software-defined networks. In *Proceedings of HotSDN 2012*.
- [17] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the debugger for my software-defined network? In *Proceedings of HotSDN 2012*.
- [18] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network management. In *Proceedings of WREN 2009*.
- [19] The Internet2 Observatory Data Collections. <http://www.internet2.edu/observatory/archive/data-collections.html>.

- [20] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with tcp throughput. *IEEE/ACM Transaction on Networking*, Aug. 2003.
- [21] M. Karsten, S. Keshav, S. Prasad, and M. Beg. An axiomatic basis for communication. In *Proceedings of SIGCOMM 2007*.
- [22] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proceedings of NSDI 2013*.
- [23] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proceedings of NSDI 2012*.
- [24] P. Kazemian, H. Zeng, and M. Chang. Header Space Library and NetPlumber repository. <https://bitbucket.org/peymank/hassel-public/>.
- [25] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proceedings of HotSDN 2012*.
- [26] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Ip fault localization via risk modeling. In *Proceedings of NSDI 2005*.
- [27] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A soft way for openflow switch interoperability testing. In *Proceedings of CoNEXT 2012*.
- [28] K. Lai and M. Baker. Nettimer: a tool for measuring bottleneck link, bandwidth. In *Proceedings of USITS 2001*.
- [29] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of SIGCOMM 1998*.
- [30] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of HotNets 2010*.

- [31] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb. Detecting network-wide and router-specific misconfigurations through data mining. *IEEE/ACM Transaction on Networking*, Feb. 2009.
- [32] F. Le, G. G. Xie, D. Pei, J. Wang, and H. Zhang. Shedding light on the glue logic of the internet routing architecture. In *Proceedings of SIGCOMM 2008*.
- [33] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iplane: an information plane for distributed services. In *Proceedings of OSDI 2006*.
- [34] A. Mahimkar, Z. Ge, J. Wang, J. Yates, Y. Zhang, J. Emmons, B. Huntley, and M. Stockert. Rapid detection of maintenance induced changes in service performance. In *Proceedings of CoNEXT 2011*.
- [35] A. Mahimkar, J. Yates, Y. Zhang, A. Shaikh, J. Wang, Z. Ge, and C. T. Ee. Troubleshooting chronic conditions in large ip networks. In *Proceedings of CoNEXT 2008*.
- [36] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteatr. In *Proceedings of SIGCOMM 2011*.
- [37] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot. Characterization of failures in an operational ip backbone network. *IEEE/ACM Transaction on Networking*, 2008.
- [38] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*.
- [39] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Computer Communication Review*, March 2008.
- [40] OnTimeMeasure. <http://ontime.oar.net/>.
- [41] Open vSwitch. <http://openvswitch.org/>.

- [42] All-pairs ping service for PlanetLab ceased. <http://lists.planet-lab.org/pipermail/users/2005-July/001518.html>.
- [43] Protobuf. <http://code.google.com/p/protobuf/>.
- [44] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of SIGCOMM 2012*.
- [45] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Jardetzky. Predicate routing: enabling controlled networking. *SIGCOMM Computer Communication Review*, January 2003.
- [46] R. C. Scott, A. Wundsam, K. Zarifis, and S. Shenker. What, where, and when: Software fault localization for sdn. Technical Report UCB/EECS-2012-178, EECS Department, University of California, Berkeley, Jul 2012.
- [47] S. Shenker. The future of networking, and the past of protocols. <http://opennetsummit.org/talks/shenker-tue.pdf>.
- [48] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *Proceedings of OSDI 2010*.
- [49] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: understanding the causes and impact of network failures. In *Proceedings of SIGCOMM 2010*.
- [50] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.
- [51] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: enabling record and replay troubleshooting for networks. In *Proceedings of USENIX ATC 2011*.
- [52] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *Proceedings of INFOCOM 2005*.

- [53] P. Yalagandula, P. Sharma, S. Banerjee, S. Basu, and S.-J. Lee. S3: a scalable sensing service for monitoring large networked systems. In *Proceedings of INM 2006*.
- [54] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*.
- [55] H. Zeng and P. Kazemian. ATPG code repository. <http://eastzone.github.com/atpg/>.
- [56] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. In *Proceedings of CoNEXT 2012*.
- [57] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. A Survey on Network Troubleshooting. Technical Report Stanford/TR12-HPNG-061012, Stanford University, June 2012.