

RICE UNIVERSITY

Efficient Traffic Trajectory Error Detection

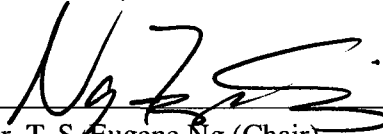
by

Bo Zhang

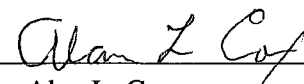
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

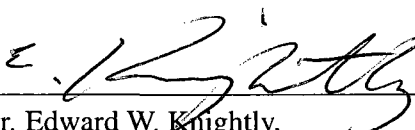
APPROVED, THESIS COMMITTEE:



Dr. T. S. Eugene Ng (Chair),
Assistant Professor,
Computer Science



Dr. Alan L. Cox,
Associate Professor,
Computer Science



Dr. Edward W. Knightly,
Professor,
Electrical and Computer Engineering

HOUSTON, TEXAS

MAY, 2010

UMI Number: 3421173

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

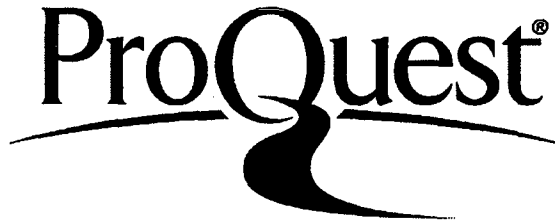
In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 3421173

Copyright 2010 by ProQuest LLC.

All rights reserved. This edition of the work is protected against unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

ABSTRACT

Efficient Traffic Trajectory Error Detection

by

Bo Zhang

Our recent survey on publicly reported router bugs shows that many router bugs, once triggered, can cause various *traffic trajectory errors* including traffic deviating from its intended forwarding paths, traffic being mistakenly dropped and unauthorized traffic bypassing packet filters. These traffic trajectory errors are serious problems because they may cause network applications to fail and create security loopholes for network intruders to exploit. Therefore, traffic trajectory errors must be quickly and efficiently detected so that the corrective action can be performed in a timely fashion. Detecting traffic trajectory errors requires the real-time tracking of the control states (e.g., forwarding tables, packet filters) of routers and the scalable monitoring of the actual traffic trajectories in the network. Traffic trajectory errors can then be detected by efficiently comparing the observed traffic trajectories against the intended control states. Making such trajectory error detection efficient and practical for large-scale high speed networks requires us to address many challenges.

First, existing traffic trajectory monitoring algorithms require the simultaneously monitoring of all network interfaces in a network for the packets of interest, which will cause a daunting monitoring overhead. To improve the efficiency of traffic trajectory monitoring, we propose the *router group monitoring* technique that only monitors the periphery interfaces of a set of selected router groups. We analyze a large number of real network topologies and show that effective router groups with high trajectory error detection rates exist in all cases. We then develop an analytical model for quickly and accurately estimating the detection rates of different router groups. Based on this model, we propose an

algorithm to select a set of router groups that can achieve complete error detection and low monitoring overhead.

Second, maintaining the control states of all the routers in the network requires a significant amount of memory. However, there exist no studies on how to efficiently store multiple complex packet filters. We propose to store multiple packet filters using a shared HyperCuts decision tree. To help decide which subset of packet filters should share a HyperCuts decision tree, we first identify a number of important factors that collectively impact the efficiency of the resulting shared HyperCuts decision tree. Based on the identified factors, we then propose to use machine learning techniques to predict whether any pair of packet filters should share a tree. Given the pair-wise prediction matrix, a greedy heuristic algorithm is used to classify packet filters into a number of shared HyperCuts decision trees. Our experiments using both real packet filters and synthetic packet filters show that our shared HyperCuts decision trees require considerably less memory while having the same or a slightly higher average height than separate trees. In addition, the shared HyperCuts decision trees enable concurrent lookup of multiple packet filters sharing the same tree.

Finally, based on the two proposed techniques, we have implemented a complete prototype system that is compatible with Juniper's JUNOS. We have shown in the thesis that, to detect traffic trajectory errors, it is sufficient to only selectively implement a small set of key functions of a full-fledged router on our prototype, which makes our prototype simpler and less error prone. We conduct both Emulab experiments and micro-benchmark experiments to show that the system can efficiently track router control states, monitor traffic trajectories and detect traffic trajectory errors.

Acknowledgment

First and foremost, I would like to thank my advisor, Dr. T. S. Eugene Ng, for his insights in helping guide the direction of this thesis. I am greatly indebted to him for his guidance, vision, and the freedom he has given to me to pursue my research interests. He has been incredibly patient to me from the very beginning of my graduate studies at Rice University. I feel very fortunate for having the chance to work closely with him. I would also like to thank Dr. Alan Cox and Dr. Edward Knightly for serving on my committee and providing me with valuable feedback to significantly improve the quality of this thesis.

I owe a great deal to Guohui Wang for his inspiring discussions on various aspects of my research. He also helped me a lot in building the prototype detector. I would also like to thank Charles Laubach for his help in setting up the Emulab testbed using the Juniper Olive router. Angela Yun Zhu helped formalize the router group selection problem and then proved that the optimal router group selection problem is NP hard. I am also thankful to Zheng Cai, Florin Dinu and Jie Zheng for not only helping in my thesis proposal and defense, but also for making my entire stay at Rice exciting and entertaining.

At last, I would like to thank my wife Angela Yun Zhu and my parents for the advice, affection, and support they have given me. Without them, this thesis would not exist.

Contents

List of Illustrations	viii
List of Tables	xii
1 Introduction	1
1.1 Defining Traffic Trajectory Errors	2
1.2 Traffic Trajectory Errors In The Wild	2
1.3 The Need For Detecting Traffic Trajectory Errors	4
1.4 Challenges of Efficient Traffic Trajectory Error Detection	5
1.5 Contributions	7
1.6 Thesis Outline	9
2 Background	10
2.1 Important Assumptions	10
2.2 Limitations of Existing Techniques	11
3 Monitoring Routers In Group	14
3.1 Effectiveness of Router Group Monitoring in Practice	18
3.1.1 Methodologies	20
3.1.2 Detection Rate of Forwarding Errors	22
3.1.3 Detection Rate of Packet Filtering Errors	26
3.2 Analytical Model for Error Detection Rate	28
3.2.1 Contributing Factors of Error Detection Rate	29
3.2.2 Analytical Model	32
3.2.3 Prediction Accuracy of Using Model vs. Sampling	35

3.3	Router Group Selection Algorithm	36
3.3.1	Correctness of Router Group Selection Algorithm	38
3.3.2	Optimality of Router Group Selection Algorithm	39
3.3.3	Heuristic Algorithm for Router Group Selection	44
3.3.4	Performance of Heuristic Router Group Selection Algorithm	47
3.3.5	Discussion	48
3.4	Applications of Router Group Monitoring	49
3.4.1	Applying to Trajectory Sampling	50
3.4.2	Applying to Fatih	52
3.5	Related Work	53

4 Constructing Shared HyperCuts Decision Trees for Multiple

Packet Filters		57
4.1	Background	58
4.1.1	Packet Filter Notations	58
4.1.2	The HyperCuts Data Structure and Algorithm	59
4.1.3	Extend the HyperCuts Data Structure and Algorithm	60
4.1.4	Efficiency Metrics of The HyperCuts Decision Tree	62
4.2	Challenges of Constructing Efficient Shared HyperCuts Decision Tree	63
4.2.1	Filter Data Sets	63
4.2.2	Making Randomly Selected Packet Filters Share HyperCuts Trees?	64
4.3	Clustering Packet Filters to Construct Efficient Shared HyperCuts Decision Trees	65
4.3.1	Factors Affecting the Efficiency of the Shared Trees	66
4.3.2	Predicting Good Pairs of Packet Filters	82
4.3.3	Clustering Packet Filters Based on Pair-wise Prediction	84
4.4	Performance Evaluation	85
4.4.1	Accuracy of Predicting Good Pairs	85

4.4.2	Performance of The Filter Clustering Algorithm	90
4.4.3	Computation Time Breakdown	93
4.5	Another Application of the Shared HyperCuts Decision Tree	95
4.5.1	The Need for Multiple Packet Filters on a Single Router	96
4.5.2	Challenges of Deploying Multiple Packet Filters on a Single Router	96
4.6	Related Work	97
5	System Design	99
5.1	Router Control State Collection	99
5.2	Traffic Trajectories Monitoring and Collection	105
5.3	Router Trajectory Error Detector	108
5.3.1	Discussion	112
6	Prototype Evaluation	114
6.1	Prototype System Implementation	114
6.2	Emulab Testbed Setup	114
6.3	Performance of Router Control State Collection	116
6.4	Performance of Traffic Trajectories Collection	117
6.5	Performance of Trajectory Error Detection	118
6.6	Integrated Trajectory Error Detection Demonstration	120
7	Conclusion and Future Work	122
7.1	Summary of Contributions	122
7.2	Future Work	124
	Bibliography	126

Illustrations

3.1	Example illustrating router group monitoring. (a) Per-router monitoring is not the most efficient. (b) Group monitoring is more efficient and can detect most trajectory errors within the group. (c) Zooming in to a finer scale to identify the misbehaving router. (d) Two different router groups with different scales can be concurrently monitored.	15
3.2	Illustration of router group monitoring technique.	15
3.3	The impact of router group topology on misforwarding detection.	23
3.4	Average detection rate of one forwarding error.	24
3.5	Average detection rate of two dependent forwarding errors.	26
3.6	Average detection rate of three dependent forwarding errors.	27
3.7	Overlap ratio between the 10% router groups with highest 1-error detection rates and the 10% router groups with the highest 2 or 3-error detection rates.	28
3.8	The detection performance for uRPF errors.	29
3.9	Illustration of how the number of exiting interfaces impacts the error detection rate.	31
3.10	Analytical formula for estimating error detection rate.	33
3.11	Prediction errors comparison of model-based and sampling-based approaches.	37
3.12	Computational speedup of computing error detection rates using model versus computing error detection rates using full simulation approach. . . .	37

3.13	The model accurately preserves the ranking order among pairs of router groups.	38
3.14	Comparison of average error detection speeds of different router group selection approaches.	42
3.15	Detection speedup when varying the sampling rate and the maximum number of interfaces concurrently monitored.	51
3.16	Percentage of monitored interfaces required to achieve the same detection speed as the original Trajectory Sampling.	52
3.17	Router group monitoring helps Fatih reduce the number of monitored interfaces.	54
3.18	Router group monitoring helps Fatih reduce communication overheads. . .	54
4.1	Example of a shared HyperCuts tree: (a) Two separate HyperCuts trees. (b) The corresponding shared HyperCuts tree.	61
4.2	(a) Memory consumption increases when randomly selected packet filters share a HyperCuts tree. (b) Average depths of leaf nodes increase when randomly selected packet filters share a HyperCuts tree.	66
4.3	Filter size V.S. memory consumption.	68
4.4	Number of internal nodes V.S. number of leaf nodes.	69
4.5	Filter size V.S. number of internal nodes.	71
4.6	Filter size V.S. number of leaf nodes.	72
4.7	Number of unique elements V.S. number of cuts.	73
4.8	Filter size V.S. height of trees.	74
4.9	Height of trees V.S. average depth of leaf nodes.	75
4.10	Size differences: good pairs V.S. bad pairs.	76
4.11	Difference of number of internal nodes: good pairs V.S. bad pairs.	77
4.12	Difference of number of leaf nodes: good pairs V.S. bad pairs.	78
4.13	Difference of memory consumption: good pairs V.S. bad pairs.	79

4.14	Correlation of number of unique elements in all dimensions: good pairs V.S. bad pairs.	80
4.15	Correlation of number of cuts on all dimensions: good pairs V.S. bad pairs.	81
4.16	Height difference: good pairs V.S. bad pairs.	82
4.17	Average leaf nodes depth difference: good pairs V.S. bad pairs.	83
4.18	Constructing a graph based on pair-wise prediction.	84
4.19	The filter classification algorithm helps alleviate the high false negative rate of the pair-wise prediction.	89
4.20	Performance of decision tree technique: (a) false positive rate (b) false negative rate.	89
4.21	Performance of generalized linear regression: (a) false positive rate (b) false negative rate.	90
4.22	Performance of naive Bayse classifier: (a) false positive rate (b) false negative rate.	90
4.23	Shared HyperCuts trees V.S. separate HyperCuts trees for Purdue data. . . .	92
4.24	The shared HyperCuts trees enable concurrent lookup of multiple packet filters sharing the same tree.	93
4.25	Shared HyperCuts trees V.S. separate HyperCuts trees: (a) Syn1-Exp (b) Syn1-100.	93
4.26	Shared HyperCuts trees V.S. separate HyperCuts trees: (a) Syn2-Exp (b) Syn2-100.	94
4.27	Shared HyperCuts trees V.S. separate HyperCuts trees: (a) Syn3-Exp (b) Syn3-100.	94
5.1	Using a modified Quagga daemon to compute a router's BGP routing state.	100
5.2	Preprocessing and supply of behavioral evidence to traffic trajectory error detectors.	109

5.3	Trajectory error detection mechanisms. (a) The baseline mechanism is to simulate a monitored router group hop-by-hop based on the routers' state. (b) By combining the FIBs in a router group, simulating the aggregated forwarding behavior requires only one longest address prefix match operation. (c) Advanced router behaviors are specified by packet filters. By aggregating the filters in a router group and performing an aggregated match, the detector can efficiently decide whether hop-by-hop simulation can be avoided.	110
6.1	Emulated Internet2 testbed on EmuLab.	115
6.2	The performance of OSPF and BGP control state collection. (a) OSPF. (b) BGP.	116
6.3	The performance of detector components. (a) Trie lookup time. (b) Decision tree lookup time. (c) Flow hash table lookup time.	119
6.4	The overall performance of trajectory error detection. (a) Average processing time for each flow. (b) Peak memory usage during the detection.	119

Tables

3.1	Summary of 12 real network topologies used in our experiment. The three numbers in the Degree column are (minimum degree, average degree and maximum degree).	22
4.1	A simple packet filter example with 10 rules defined on five packet header fields.	59
4.2	Summary of basic statistics about the seven filter data sets.	64
4.3	Computation time breakdown (in seconds) for each step in the proposed approach.	95
6.1	Timestamp of each trajectory error first detected.	121

Chapter 1

Introduction

The network topology and routing policy of an operation network are carefully designed so that the routers forward traffic along the chosen trajectories. In addition, different packet filters serving various functions are also carefully placed on routers to enforce the desired traffic reachability.

In the ideal case, routers should process traffic as intended so that the traffic always follows the chosen paths. However, routers are complex systems. They typically run an operating system (e.g., Cisco IOS and Juniper JUNOS), and a collection of protocol daemons which implement the various tasks associated with protocol operations. Like any complex software, routing software is prone to implementation bugs. The implementation bugs can affect routers in different ways. One class of bugs causes a router to crash or reboot. Fortunately, router crash or reboot is relatively easy for an operator to notice. Another class of bugs causes non-critical effects (e.g., a slow memory leak) that does not affect network services. However, a large number of router bugs, once triggered, can cause various traffic trajectory errors including forwarding error (i.e., traffic deviating from its intended forwarding paths), dropping error (i.e., traffic being mistakenly dropped) and filter-bypass error (i.e., unauthorized traffic bypassing packet filters). These traffic trajectory errors are serious problems because they may cause network applications to fail and create security loopholes for network intruders to exploit. Worse, traffic trajectory errors can be subtle and hard to detect during development or deployment. Note that static router configuration correctness checking tools [FR01, FB05] or control plane monitoring mechanisms [SG04] do not help here. This is because the bugs may exist even when routers are correctly configured by the operator, and the control plane (e.g., OSPF, BGP) of a buggy router may

continue to appear to be working correctly.

1.1 Defining Traffic Trajectory Errors

In this section, we define the three types of traffic trajectory errors that we address in this thesis.

- Packet dropping error - If a router drops legitimate packets that it should continue to forward along their trajectory, then we call it a packet dropping error. A packet could be dropped due to an implementation bug related to the forwarding table or the packet filter.
- Forwarding error - If the intended control state of a router indicates that a packet should be forwarded to its neighbor $N1$ but in reality neighbor $N2$ receives it, then we call it a forwarding error. When a forwarding error happens, the misforwarded packet could fall into a forwarding loop and never reach its destination, or the packet could take a different path but eventually still reach its destination. Whichever case happened, it is still an trajectory error because the packet deviated from its intended path.
- Filter bypass error - If a packet filter is supposed to drop a certain packet but mistakenly lets it through, we call it a filter bypass error.

All three types of errors can directly affect the traffic trajectory in the network. In the next section, we will further explain how these traffic trajectory errors happen in practice.

1.2 Traffic Trajectory Errors In The Wild

A recent study [CR08, KYCR09] manually classified the bugs found in Quagga [quab] and XORP [xor] open source routers. They found that more than 200 and 500 bugs respectively have been reported for Quagga since 2006 and for XORP since 2003. Cisco and Juniper are the current leaders in the IP router market. Cisco has extensively documented over

200 bugs in their products since 1995. Details of these bugs are publicly available [cisb]. Juniper also documents bugs in their products. Unfortunately, this information is not publicly available, only registered customers are allowed to access [Jund]. Here we cite several recently reported Quagga and Cisco router bugs that can cause traffic trajectory errors to illustrate two points:

1. Real-world router bugs could lead to a wide range of traffic trajectory errors.
2. These bugs may only affect a specific subset of data packets and may leave no gross evidence. Therefore, they are difficult to detect.

Multiple reported Quagga bugs can result in incorrect routing tables such as new routes being ignored (Quagga Bugzilla [quaa] bug ID: 298, 464, 518), expired routes being used (Quagga Bugzilla bug ID: 85, 134), incorrect routes being installed (Quagga Bugzilla bug ID: 238, 546), and routers stop adapting to topology change (Quagga Bugzilla bug ID: 107). For Cisco routers, multiple reported bugs can cause a network interface to drop all future packets (Cisco Advisory IDs [cisb]: cisco-sa-20080326-IPv4IPv6, cisco-sa-20090325-udp). Another bug may cause the firewall module of Cisco routers to stop forwarding traffic (Advisory ID: cisco-sa-20090819-fwsm). Another bug may change the forwarding table of a router (Advisory ID: cisco-sa-20080326-mvpn). Yet another bug may invalidate control-plane access control lists (Advisory ID: cisco-sa-20080604-asa). Multiple bugs can stop access control list from working, so that unauthorized traffic can go through the affected routers (Advisory IDs: cisco-sa-20090923-acl, cisco-sa-20071017-fwsm, cisco-sa-20011114-gsr-acl, cisco-sa-20000803-grs-acl-bypass-dos). Another bug (Advisory ID: cisco-sa-20070412-wlc) could allow packet filters to be inserted so that some packets may be dropped silently.

According to Cisco advisory [cisb] and Quagga Bugzilla [quaa], the reported Cisco and Quagga router bugs exist in multiple versions of Cisco IOS and Quagga routing software, thus, many deployed routers may be affected by those bugs. Worse, there are likely many more bugs yet to be discovered.

1.3 The Need For Detecting Traffic Trajectory Errors

Eliminating router bugs during development is hard in practice. Ideally, all router bugs should be discovered through rigorous testings and design verifications during the development stage and should be corrected prior to deployment. Unfortunately, router hardware and software are highly complex systems, so no vendor can test all network designs, configurations and traffic patterns that can exist in the real world. That is, realistically routers may never be bug-free. Therefore, many bugs will remain undiscovered in deployed routers. Although it helps to have vendors provide patches for the subset of bugs that they have discovered, it is doubtful that network operators always keep their routers up-to-date. Network operators must cope with these bugs when they are eventually triggered in the field. Therefore, it would be very beneficial for the network operator to have the ability to detect traffic trajectory errors quickly and efficiently.

In this thesis, we consider the traffic trajectory error detection problem in the context of a single autonomous system (e.g., one ISP, one campus network, one enterprise network, etc.). To address this problem, two classes of trajectory error detection techniques have been proposed by other researchers. The first class of techniques (e.g., [AKWK04] [HK00] [Per88] [WAAR06] [ZGC03]) relies on routers sending some form of packet arrival acknowledgments towards the packet source to confirm that a packet is making correct progress in the network. The second class of techniques (e.g., WATCHERS [BCP⁺98, HAB00], Fatih [MCMS05, MCMS06], SATS [LWK06], Trajectory Sampling [DG00], etc) relies on nodes to collect some form of network behavioral evidence and then processes such evidence to detect trajectory errors. More detailed discussions of these trajectory error detection techniques can be found in Section 2.2.

Although the two classes of techniques employ different approaches to detecting trajectory errors, we find that they share two common building blocks:

- **Tracking and maintaining routers' control states:** The control states of routers include the forwarding tables, packet filters and so on. The control states of routers determine the routers' behaviors, i.e., how routers should process packets. Knowing the

intended behaviors of routers is essential for correct trajectory error detection. The network control states that governs the various network functions can be obtained from control protocol messages, network operators, as well as router configuration files.

- **Collecting the actual traffic trajectories:** To obtain the actual trajectories of the traffic, we need to monitor how traffic flows through the network. The first class of techniques relies on the acknowledgments to learn the actual traffic trajectories, while the second class of techniques uses either counters to record the traffic statistics or the packet sampling techniques to monitor the trajectories of packets.

Once we obtain the intended control states and the actual traffic trajectories, traffic trajectory errors can then be detected by comparing the observed traffic trajectories against the intended trajectories according to the control states. If the observed traffic trajectories contradict the intended trajectories, a trajectory error is detected. Detected trajectory errors may be cross-validated by active probing facilities such as Cisco IPSLA [ips] to minimize false detections.

1.4 Challenges of Efficient Traffic Trajectory Error Detection

Although the basic idea of the trajectory error detection is straight-forward, it is actually challenging to design and implement an efficient and scalable trajectory error detection system, especially when the network is composed of a large number of routers and high-speed links. Specifically, to enable efficient and scalable traffic trajectory error detection, the following challenges need to be addressed first:

- **Efficiency of trajectory monitoring:** The first challenge we have to address is how to efficiently observe the network-wide traffic trajectories while incurring as little monitoring overhead as possible. To monitor how the traffic flows through a network, the existing techniques need to enable the traffic monitoring function on *all* the network interfaces in a network. Ideally, each interface should monitor all traffic

that is going through it. However, monitoring all traffic at full rate will incur a high monitoring and reporting overhead on routers and the network, so in practice only a certain fraction of traffic is sampled for each monitoring period. Specifically, during each monitoring period, all monitoring devices deterministically choose a certain subset of the packets, typically by a packet header hashing technique [MND05], to be sampled by all interfaces. Different subsets of packets are then monitored during different monitoring periods.¹ Once the actual traffic trajectories are obtained, traffic trajectory errors can then be detected by comparing the observed traffic trajectories against the intended trajectories according to the obtained control states. Although the traffic sampling can help reduce the monitoring overhead, existing approaches still require concurrent monitoring of all the network interfaces. In this thesis, we propose a novel technique to improve the efficiency of the trajectory monitoring by only monitoring a subset of interfaces during each monitoring period. The proposed technique is generic and can be adopted by multiple trajectory error detection systems to improve their efficiency.

- **Efficiency of maintaining control states:** Secondly, the detector needs to maintain control states of multiple routers to know their intended behaviors. Maintaining the control states of multiple routers in the network usually requires significant amount of memory. Efficient data structure for maintaining multiple forwarding tables has been studied by Fu and Rexford [FR08a]. How to efficiently store multiple packet filters has not been studied yet. Due to the complexities of the network services, each packet filter may be large and complex as well. For example, recent studies have shown that a complex packet filter on modern routers or firewalls can have as many as 50,000 rules [ZWG07]. Therefore, a large amount of memory is required

¹Trajectory errors may be persistent or not. Persistent errors are guaranteed to be eventually detected as long as the packet sampling method eventually covers the affected packets. On the other hand, rare, temporary trajectory errors may be detectable with a certain probability depending on the sampling method, the affected packets, and the duration of the error. The rest of this thesis assumes we are dealing with persistent errors.

to hold a large number of complex packet filters. Given the limited memory on the detector, it is critical for the detector to employ efficient data structures to store the multiple packet filters.

1.5 Contributions

This thesis makes the following contributions to enable more efficient and scalable traffic trajectory error detection:

1. **Router group monitoring:** We propose the router group monitoring technique, which only monitors the periphery interfaces of a group of routers. Because only a subset of interfaces in the network are monitored for each monitoring period, the monitoring overhead can be significantly reduced. Router group monitoring introduces a new spatial dimension to traffic trajectory error detection. That is, in addition to the dimension of varying the packet sampling rate to adjust the monitoring overhead, a new dimension to be considered is which network interfaces are to be monitored. To study whether the router group monitoring can be effective in practice, we perform extensive simulation based experiments on a large number of real network topologies. Our experiments show that the vast majority of the traffic trajectory errors within a router group can still be detected by only monitoring the periphery interfaces of the router group. To better understand what can affect the effectiveness of different router groups, we explore different factors. We show that the router group size, the average router degree inside a group, and the number of exits leaving the group are the key factors that influence a router group's detection rate. Based on the identified factors, we develop an analytical model for quickly and accurately estimating the detection rates of different router groups so that we do not need to use computationally expensive simulation to calculate the detection rate of a router group. This model makes it possible to identify effective router groups very efficiently. We propose a novel algorithm to select a set of router groups that can achieve guaranteed error detection and low monitoring overhead. Furthermore, this approach achieves

faster error detection than other monitoring algorithms. Finally, by only monitoring a group of routers at a time, the computation overhead for generating the necessary router states and for processing the behavioral evidence is reduced. We show that the efficiency of trajectory error detection based on Trajectory Sampling or Fatih can be significantly improved by applying the router group monitoring idea to them.

2. Efficient shared data structure for multiple packet filters:

To efficiently store multiple packet filters at the detector, we propose to use a shared data structure based on the HyperCuts decision tree, which is widely adopted by commercial routers. We first extend the original HyperCuts decision tree data structure and the tree construction algorithm to support multiple packet filters on a shared HyperCuts decision tree. We then experimentally show that naively classifying packet filters into shared HyperCuts decision trees may significantly increase memory consumptions and search time. To help decide which subset of packet filters should share a HyperCuts decision tree, we first identify a number of important factors that collectively impact the efficiency of the resulting shared HyperCuts decision tree. Based on the identified factors, we then propose to use machine learning techniques to predict whether any pair of packet filters should share a tree. Given the pair-wise prediction matrix, a greedy heuristic algorithm is used to classify packet filters into a number of shared HyperCuts decision trees. Our evaluation shows that the false positive rate of the pair-wise prediction algorithm is low. Though the false negative rate of the pair-wise prediction is relatively higher, we show that the classification algorithm can help alleviate the high false negative problem. Our experiments using both real packet filters and synthetic packet filters show that the resulting shared HyperCuts decision trees require considerably less memory while having the same or a slightly higher average height than the separate trees. In addition, the shared HyperCuts decision trees enable concurrent lookup of multiple packet filters sharing the same tree.

3. **Prototype detector compatible with Juniper JUNOS:**

We have implemented a complete prototype system based on the above two novel techniques. In addition, as opposed to some existing error detection techniques (e.g., Fatih, WATCHERS, Trajectory Sampling, etc) that require new router features, our prototype does not require any router modification and is completely compatible with Juniper's JUNOS [juna] version 8.5 running on FreeBSD (aka Olive [oli]). Our prototype leverages widely available traffic flow monitoring capabilities in routers (e.g. NetFlow [sne], Flexible NetFlow [fle], IPFIX [ipf]) to distributedly collect trajectories of packets in the network. As we will show in Section 5, to detect traffic trajectory errors, it is sufficient to only selectively implement a small set of key functions of a full-fledged router on our prototype, which makes our prototype implementation simpler and less error prone. Consequently, the detector can efficiently detect real world traffic trajectory errors. We conduct micro-benchmark experiments to show that the prototype can efficiently compute and maintain the routers' control states on demand, and efficiently monitor actual traffic trajectories to detect traffic trajectory errors. We also demonstrate the overall behavior of the complete system working on a realistically emulated Internet2 backbone network in Emulab.

1.6 **Thesis Outline**

The rest of this thesis is organized as follows. In Chapter 2, we discuss the assumptions we make, and explain the previous work and why they fall short. In Chapter 3, we present our router group monitoring approach and demonstrate its benefits. In Chapter 4, we present our efficient shared data structure for representing multiple packet filters. In Chapter 5, we present the design of our prototype system. We evaluate the performance of our system and demonstrate its capability in Chapter 6 using the Emulab testbed. Finally, we conclude in Chapter 7.

Chapter 2

Background

2.1 Important Assumptions

- **Single autonomous system:** We consider the problem in the context of a single autonomous system. That is, the network operator knows the intended configuration (e.g., link weights, packet filters) of each router.
- **Trusted control plane:** Control plane security is an orthogonal problem that this thesis will not focus on. We assume that the control plane of a network is trusted. Therefore, all information contained in control protocol messages (e.g., OSPF LSAs, BGP updates, IGMP messages, etc.), in direct configuration commands from network controllers (e.g. SNMP SET commands), and in router configuration files from the operator are assumed to be correct. While control plane security may not be a solved problem, many mechanisms do exist for securing the control plane and for detecting control plane misbehaviors (e.g., [HKD07, HARD09, SG04, Per00, Kum93, SMGLA97, ZH99, BHBR01, PST⁺02, Che97, HPT97, HJP03, SRS⁺04, RMR07, CR08, KYCR09, MB96, WWV⁺97, PSS⁺01, WEO04, KLMS00, Whi03]). For example, [SG04] can be used to monitor the link weights of the OSPF protocol. If the weight of any link is changed without the knowledge of the operator, the operator will be alerted. While accountability protocols such as PeerReview ([HARD09] [HKD07]) can help detect the misbehaving BGP routers.
- **Observability of trajectory errors:** A fundamental requirement for traffic trajectory error detection in *any* approach is that the evidence of a trajectory error must be observable by good evidence-collecting nodes. This thesis does not address tra-

jectory errors that are not observable. For example, if one router mistakenly drops a packet destined to itself, then this trajectory error cannot be detected because it is not observable from outside. Another example is: a group of colluding routers that changes the forwarding path of a packet *within* the colluding group and reports forged forwarding evidence is generally not detectable. Fortunately, forging evidence that is globally consistent will be quite hard in practice. Trajectory errors exhibited by edge routers (e.g., ingress/egress routers dropping inbound/outbound packets) can be observed by dedicated evidence collecting nodes (such as commercial NetFlow-based network monitoring devices [flob]) that tap the edge connections.

2.2 Limitations of Existing Techniques

We separate the discussion of existing techniques into primary techniques, which aim directly at detecting traffic trajectory errors, and secondary techniques, which may indirectly catch traffic trajectory errors.

Primary techniques: There are two classes of primary techniques. The first class of techniques relies on nodes sending some form of packet arrival acknowledgments towards the source to confirm that a packet is making correct progress in the network. The second class of techniques relies on nodes to collect some form of network behavioral evidence and processes such evidence to detect trajectory errors.

The first common limitation of *all* existing techniques in these two classes is that they assume destination address-based packet forwarding is the only network function. Consequently, they will falsely detect many ordinary scenarios as malicious. For example, access control list (ACL) filtering will be detected as malicious packet drop; marking of a packet's Type-of-Service (TOS) byte for QoS differentiation will be detected as malicious packet modification; tunneling of packets for VPN services will be detected as malicious packet redirection. Note that our proposed approach is an instance of the second class of techniques, but our proposed approach is unique in that it is control state-aware. The second

limitation is that the existing techniques need to enable the traffic monitoring function on *all* the network interfaces in a network, which will incur a high monitoring and reporting overhead on routers and the network. In contrast, our approach uses a novel router group monitoring technique to improve the efficiency of the traffic trajectory monitoring by only monitoring a subset of interfaces during each monitoring period.

The following is a brief summary of the first class of techniques: Avramopoulos et. al. [AKWK04] use secure source routing and end-to-end acknowledgments to detect misbehaving forwarders; Herzberg and Kuttan [HK00] describe different techniques based on end-to-end, hop-by-hop, or intermediate node acknowledgments to detect misbehaving forwarders; Perlman [Per88] uses end-to-end acknowledgments to identify unreliable network paths; Availability Centric Routing [WAAR06] and Feedback Based Routing [ZGC03] also use end-to-end acknowledgments but monitor the reliability of inter-domain paths.

The following is a brief summary of the second class of techniques: WATCHERS [BCP⁺98, HAB00] maintains several packet counters at routers and uses inconsistencies found in these counters among different routers to detect forwarding errors; Packet Obituaries [AMCS04] uses accountability boxes on inter-AS border links to record traffic information and exchanges traffic information among accountability boxes to identify unreliable inter-domain paths; Stealth Probing [AR06] hides end-to-end probe traffic among regular data traffic inside a secure tunnel and monitors the end-to-end probe traffic to determine the reliability of a network path; Secure Traceroute [PS03] compares traffic information collected at the source router and different intermediate routers to discover which intermediate router is misbehaving; Awerbuch et. al. [AHNRR02] is similar to Secure Traceroute but uses a binary search strategy for locating the misbehaving router; SATS [LWK06], PepperProbing [GXT⁺08], and SaltProbing [GXT⁺08] extend the idea of Trajectory Sampling [DG00] to securely monitor network paths for faulty forwarding behavior; Mizrak et. al. [MCMS05, MCMS06, MSM08, Miz07] propose a system called Fatih that uses traffic information to detect trajectory errors, including packet loss, fabrication, modification, re-ordering, delaying, and also provide mechanisms to distinguish congestive packet losses

from malicious packet drops.

Finally, many of the above techniques require specialized router support such as packet-by-packet fingerprinting and logging that are not currently available in commercial IP routers. In contrast, our goal is to design a solution that leverages existing router support for NetFlow and does not need specialized router support.

Secondary techniques: There is a large number of statistical network anomaly detection techniques in the literature (e.g., [Den87, LX01, FSBK03, LCD04, LCD05, ibm, cisd]). These techniques monitor traffic aggregates and aim to identify network problems such as DoS attacks, network outages, etc. While these techniques are not designed to catch specific traffic trajectory errors, they could potentially detect a traffic trajectory error if it results in a significant impact on the network's overall traffic. However, the limitation is that a localized error is very likely to evade statistical detection. For example, a misbehaving router that changes the forwarding behavior of a single /28 destination address prefix is unlikely to cause a gross detectable anomaly. Therefore, while statistical anomaly detection techniques can potentially be helpful in the fight against traffic trajectory errors, their use is limited and so we consider these techniques secondary.

Chapter 3

Monitoring Routers In Group

To monitor the behavior of a router, the basic idea is to monitor the traffic being sent to and received from this router by its neighbors. For example, consider the illustrative network in Figure 3.1(a). To monitor one of the middle routers requires the monitoring of six neighbor router interfaces. Monitoring all four middle routers will thus require the monitoring of 24 interfaces. At the extreme, if every router is to be monitored, then every network interface needs to be monitored. The overall processing overhead in trajectory error monitoring depends on both the sampling rate and the number of concurrently monitored interfaces. However, an important observation is that to detect a traffic trajectory error, it is in general unnecessary to monitor all network interfaces concurrently because it is sufficient to have just one monitored interface detect the error. In other words, monitoring all interfaces concurrently is overkill for trajectory error detection. Take Figure 3.2 as an example, if all interfaces are monitored, then routers R_4 , R_5 , and R_6 can all detect the same forwarding error, which creates unnecessary monitoring overheads.

This observation leads to the following question: Compared to the straight-forward setting of monitoring all interfaces concurrently, is it possible to detect the same trajectory errors in fewer sampling periods (i.e. faster) on average, without increasing the overall processing overhead, by monitoring fewer interfaces concurrently but each at a higher packet sampling rate? Conversely, is it possible to maintain the same detection speed, but reduce the overall processing overhead by monitoring fewer interfaces at the same packet sampling rate?

This chapter studies these questions under a particular interface monitoring strategy we call *router group monitoring*. Suppose we model a network as a graph $G(V, E)$ where V

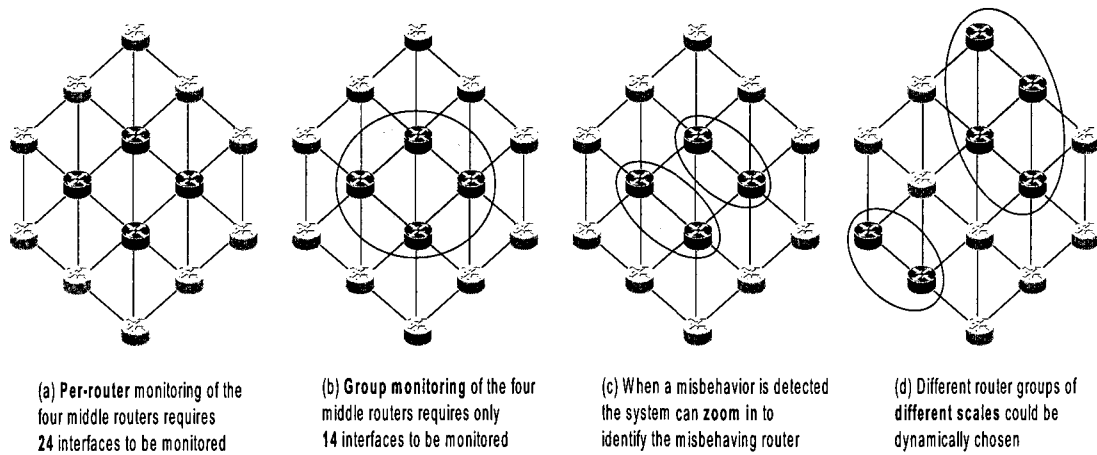


Figure 3.1 : Example illustrating router group monitoring. (a) Per-router monitoring is not the most efficient. (b) Group monitoring is more efficient and can detect most trajectory errors within the group. (c) Zooming in to a finer scale to identify the misbehaving router. (d) Two different router groups with different scales can be concurrently monitored.

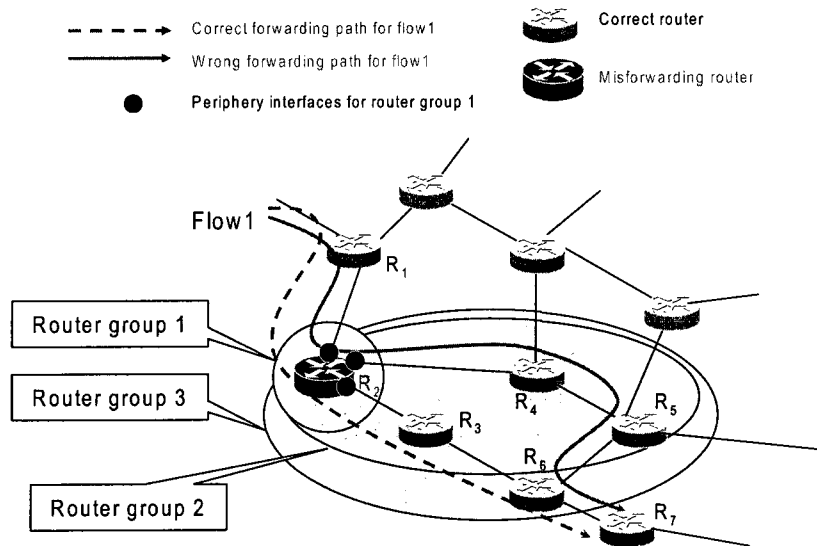


Figure 3.2 : Illustration of router group monitoring technique.

is the set of vertices (routers) and E is the set of edges (links). A router group RG_i is a set of *connected* vertices such that $RG_i \subset V$. When monitoring a router group RG_i , every

cut edge $(u, v) \in E$ with $u \in RG_i$ and $v \in V \setminus RG_i$ is monitored. We informally refer to these interfaces as the periphery interfaces of a router group. The overhead of monitoring a router group thus depends on the number of periphery interfaces and the packet sampling rate used. The potential overhead saving comes from not monitoring those edges (u, v) with $u, v \in RG_i$.

The idea of router group monitoring is illustrated in Figure 3.1(b). A subset of connected routers are chosen to form a router group for monitoring. A router group represents a network region that is being monitored by neighbor routers *outside* of the region. The router group monitoring provides a trade-off between overhead and trajectory error detection time. In the example, to monitor the four middle routers as a group requires only 14 interfaces to be monitored. Yet, most trajectory errors exhibited by the four routers can already be detected. For simplicity, consider the transit traffic that enters and then leaves the monitored region (to consider non-transit traffic simply requires additional Net-Flow agents not shown in the figure to monitor the ingress and egress links as discussed in section 2.1). Packets that are maliciously dropped, fabricated, tunneled to a third-party, let-through against ACL policies, TOS-marked, etc. can be detected from outside the region. Furthermore, a mis-forwarded packet can be detected if it leaves the region on an unexpected interface. For example, Figure 3.2 illustrates how the router group monitoring approach can detect a mis-forwarding error. Router group 1 is a singleton router group. To monitor router group 1, we only need to monitor the three periphery interfaces represented by the black circles. By monitoring the router group 1, we can detect the forwarding error immediately because the flow 1 is leaving the group from a wrong periphery interface. When monitoring the router group 2, we can still detect this forwarding error. However, monitoring the router group 3 will not detect this specific error because it has been self-corrected inside the group. This tells us that a router group may not detect all errors originated from the inside of the group.

Once a trajectory error is detected in a router group, the group can be divided up into finer scale groups to “zoom in” on the specific misbehaving router as shown in Fig-

ure 3.1(c). Multiple router groups can be concurrently monitored as long as the total overall processing overhead is below the desired ceiling. Note that concurrently monitored router groups RG_i and RG_j need not be disjoint. Figure 3.1(d) shows that two router groups are monitored concurrently. It is possible to choose router groups that guarantee to detect all persistent trajectory errors. A sufficient condition is presented in Section 3.3. This result is intuitive because one can always choose N router groups where N is the number of routers and each group corresponds to a unique router in the network; this strategy simply degenerates into the monitoring of all network interfaces.

However, to determine whether router group monitoring improves efficiency, a number of questions must be addressed. First, how likely is a trajectory error inside a router group detectable at the periphery interfaces? Second, what factors affect the detection rate of a router group and how can we efficiently identify router groups that have high detection rates? Third, how can we choose a set of router groups that can guarantee error detection while achieving a low monitoring overhead? This thesis systematically addresses each of these questions and show that router group monitoring has significant efficiency benefits for trajectory error detection.

In summary, in this chapter, we have made the following important contributions:

- We propose router group monitoring, which introduces a new spatial dimension to traffic trajectory error detection. That is, in addition to the dimension of varying the packet sampling rate to adjust the monitoring overhead, a new dimension to be considered is which network interfaces are to be monitored.
- To show that router group monitoring can be effective in practice, we analyze a large number of real network topologies by brute-force computations and show that effective router groups with high trajectory error detection rates exist in all cases.
- We show that the router group size, the average router degree inside a group, and the number of exits leaving the group are the key factors that influence a router group's detection rate. We develop an analytical model for quickly and accurately estimating

the detection rates of different router groups. This model makes it possible to identify effective router groups efficiently.

- We propose an algorithm to select a set of router groups that can achieve guaranteed error detection and low monitoring overhead. We show that applying this algorithm to select router groups to be monitored can significantly improve the efficiency of trajectory error detection based on Trajectory Sampling or Fatih.

The rest of this chapter is organized as follows. In Section 3.1, we study the effectiveness of router group monitoring in real topologies. In Section 3.2, we derive an analytical model for predicting the effectiveness of a router group. In Section 3.3, we formulate the router group selection problem and present an efficient heuristic algorithm for router group selection. In Section 3.4, we show the benefits of applying router group selection to Trajectory Sampling and Fatih. We discuss the related work in Section 3.5.

3.1 Effectiveness of Router Group Monitoring in Practice

A trajectory error represents a deviation from the intended network path and thus can potentially be detected at many interfaces in the network. Router group monitoring is a way to exploit this observation. Specifically, even if the trajectory of a packet starts to deviate from its intended path at a router inside a router group, the error may still be observable at the periphery interfaces of the router group. The effectiveness of the router group monitoring on detecting the three types of trajectory errors is discussed as follows:

- Dropping error - A dropping error simply drops all packets in the affected flow. Because a packet that is simply dropped in the middle of its trajectory will never leave the router group, by consistently observing packets missing from the intended exiting periphery interface, the error is easily detected. Thus, this thesis will not focus on dropping errors. Please note that a packet could be dropped due to a bug in the forwarding table implementation or in the packet filter implementation.

- Filter-bypass error - A filter-bypass error causes a flow to bypass a packet filter that should drop it. When a filter-bypass error occurs inside a router group, whether it will be detected by monitoring the periphery interfaces depends on the distribution of packet filters inside the group. If the flow encounters another packet filter that is designed to drop it as well before it leaves the group, then the specific filter-bypass error will not be detected. On the other hand, if the flow leaves the group, then a periphery interface will see the unexpected flow so that the error will be detected.
- Forwarding error - A forwarding error misforwards a flow to a wrong next-hop. A forwarding error can lead to two possible outcomes:
 - 1) Forwarding loop error: If a forwarding loop keeps a packet inside the router group, the packet will never leave the router group and can be detected just like a dropping error. If the forwarding loop takes the packet outside of the router group, if the exiting periphery interface is wrong, the error is detected. On the other hand, if the exiting periphery interface happens to be correct, the error is not detected by this router group.
 - 2) Detour error (no loop is formed): If the detour takes the packet outside of the router group via an incorrect exiting periphery interface, the error is detected. On the other hand, if the exiting periphery interface happens to be correct, the error is not detected by this router group.

Therefore, a router group does not guarantee the detection of all errors that start inside the group. Different router groups can also have different error detection rates. Ultimately, multiple router groups must be chosen carefully to guarantee the detection of all trajectory errors and achieve low monitoring overhead. Our evaluation will show that the router group monitoring approach is effective in detecting all types of trajectory errors.

A router at which an error occurs is called a *misbehaving router*. The misbehaving router's erroneous action such as dropping traffic, misforwarding traffic and allowing traffic

to bypass filters is called a trajectory error. More formally, a misbehaving router is said to have one trajectory error with respect to a flow i denoted as F_i if the error affects all packets belonging to F_i . We perform a series of empirical experiments to understand the impact of router group monitoring on trajectory error detection.

3.1.1 Methodologies

3.1.1.1 Static Analysis Methodology

We first consider the case where only one forwarding error exists inside a router group. Given a router group and a forwarding error inside the group, whether the forwarding error will be detected by monitoring the periphery interfaces of the router group can be decided using the following static analysis approach: starting from the misbehaving router, a hop-by-hop forwarding table lookup is used to decide the exiting interface where the mis-forwarded packet leaves the router group. If the exiting interface is the same as the original correct interface, then this error cannot be detected by using this router group. Otherwise, it can be detected because either the packet leaves from a wrong interface or a routing loop is formed. Similarly, if we want to know the overall effectiveness of one router group in detecting single forwarding error, we can calculate the *detection rate* of the router group as follows: for each router inside the group and for each possible destination in the network and for each possible wrong next hop interface for each destination, we introduce one forwarding error. Then a hop-by-hop forwarding table lookup is performed to decide whether the forwarding error can be detected. Thus, the detection rate can be calculated by dividing the number of detected errors by the number of total errors. Basically, given a network with N nodes and a router group with $|RG|$ nodes, $O(|RG| \times N \times (d - 1))$ errors will be analyzed, where d is the average node degree and accordingly $d - 1$ is the average number of wrong next hop interfaces. Because $|RG| = O(N)$ and $d = O(N)$, the complexity of exhaustively calculating the detection rate of one router group is $O(N^3)$ in the worse case.

Next, we consider the case where multiple forwarding errors exist in the router group.

When two forwarding errors are independent from each other (i.e., they affect different flows), the detection rate for these errors is the same as in the single error case. On the other hand, if multiple forwarding errors do affect the same flow, we call them “dependent forwarding errors”. We only study the detection rate of multiple dependent forwarding errors. Given a network with N nodes and a router group with $|RG|$ nodes, in order to analyze K dependent forwarding errors ($K \leq |RG|$), K distinct routers from the group will be selected, each of which will exhibit one forwarding error affecting the same flow. Each selected misbehaving router will mis-forward the flow to one wrong next hop interface. Similarly, a hop-by-hop forwarding table lookup is used to test whether the mis-forwarded packet can leave the router group from the original correct interface. The complexity of exhaustively analyzing all possible multiple forwarding errors is $O(C(|RG|, K) \times N \times (d-1)^K)$, where $C(|RG|, K) = |RG|! / K! (|RG| - K)!$. Suppose $K = 2$, then the worse case complexity is already $O(N^5)$.

For each network topology, we randomly choose router groups with different sizes and introduce forwarding errors as described above. We then can calculate an *average detection rate* for all the router groups. In Section 3.1.2, for each topology, we calculate average detection rates for router groups with different sizes. For each router group size, we choose up to 500 random router groups in order to limit the computation time. We implement our analysis tool using Matlab scripts.

3.1.1.2 Topologies

To show the real-world detection performance of router group monitoring, we conduct forwarding error detection rate experiments using a large number of real network topologies, including Internet2, TEIN2 (Trans-Eurasia Information Network), iLight (Indiana’s Optical Network), GEANT (European research network), SUNET (Swedish University Network), Sprint North America backbone network composed of only Sprint global IP nodes, and six Rocketfuel [SMW02] topologies. Table 3.1 summarizes the basic properties of each topology. For those topologies whose link weights are not available, we set all link weights as 1

Topologies:	# of nodes	# of edges	Degree	Link weight?
Internet2	9	13	(2, 2.9, 4)	Yes
TEIN2	11	11	(1, 2, 7)	No
iLight	19	21	(1, 2.2, 4)	No
GEANT	22	37	(2, 3.4, 9)	Yes
SUNET	25	28	(1, 2.2, 4)	No
Sprint (US)	28	46	(1, 3.2, 9)	No
RF-1	79	147	(1, 3.7, 12)	Yes
RF-2	87	161	(1, 3.7, 11)	Yes
RF-3	104	151	(1, 2.9, 18)	Yes
RF-4	138	372	(1, 5.4, 20)	Yes
RF-5	161	328	(1, 4.1, 29)	Yes
RF-6	315	972	(1, 6.2, 45)	Yes

Table 3.1 : Summary of 12 real network topologies used in our experiment. The three numbers in the Degree column are (minimum degree, average degree and maximum degree).

to compute their routing tables.

In addition to real network topologies, we have also conducted the same experiments on some representative synthetic topologies, such as power-law topologies (PLRG [ACL00] and INET [WJ02]), Hierarchical topologies (Transit-stub [ZCB96]) and random graphs. The results obtained from synthetic topologies are similar to those based on real network topologies shown in Section 3.1.2.

3.1.2 Detection Rate of Forwarding Errors

3.1.2.1 Single Forwarding Error

In this section, we first consider the simple case where only *one* single forwarding error exists inside a router group. Whether or not one forwarding error inside the router group can be detected depends on the network topology and routing. To illustrate, Figure 3.3(a) shows a router group with 5 routers, and the topology inside the router group has a cycle. There are two potential paths P_1 and P_2 between periphery interfaces $I f_1$ and $I f_3$. Let's assume path P_1 is the correct path for a particular flow F. Flow F should enter the router

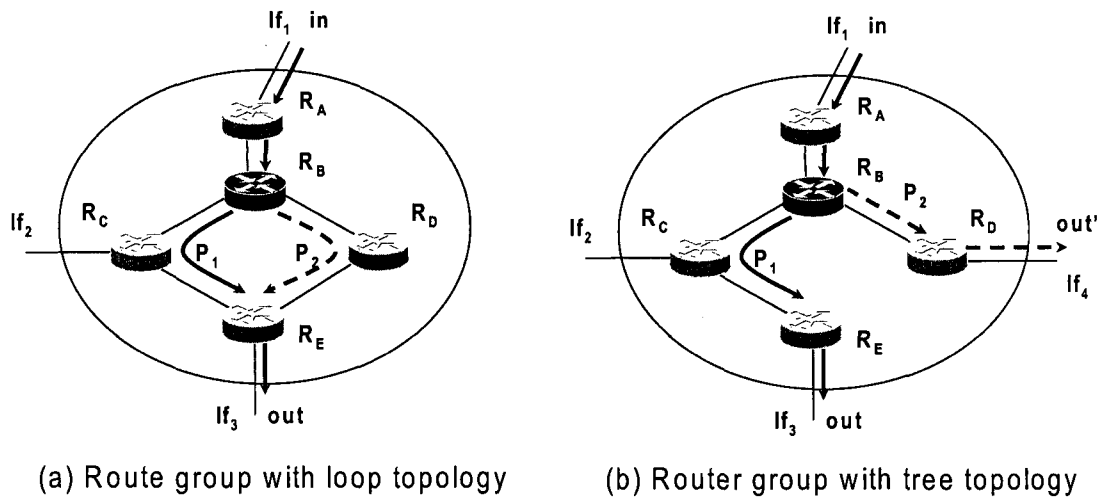


Figure 3.3 : The impact of router group topology on misforwarding detection.

group at the interface If_1 and leave at interface If_3 , following path P_1 inside the router group. However, if the router R_B has a forwarding error, it may forward the flow to router R_D as opposed to router R_C . The flow F will take path P_2 inside the router group, but it still leaves the group at interface If_3 . In this case, we cannot detect router R_B 's forwarding error by only monitoring the periphery interfaces. Generally, given a router group, if there are more than one paths between an ingress interface and an egress interface, it is possible that some forwarding errors inside a particular router group cannot be detected from the periphery interfaces. Note that the same forwarding error may be detectable by using a different router group.

In contrast, as shown in Figure 3.3(b), if a group of routers are connected in a tree topology, there is only one path between each ingress interface and egress interface. If the router R_B misforwards the flow F to the wrong path P_2 , the flow F will either leave the router group at the wrong interface If_4 , or be stuck between R_B and R_D . Therefore, in a tree topology router group, any single forwarding error is guaranteed to be detected by monitoring the periphery interfaces. Also, if a network has a full-mesh topology with all links having equal link weight, a misforwarding inside any router group is guaranteed

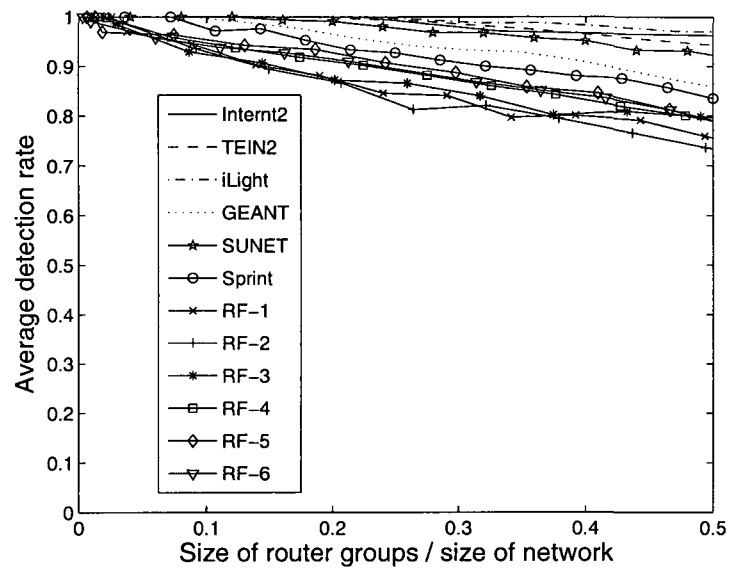


Figure 3.4 : Average detection rate of one forwarding error.

to be detected. This is because all nodes have a one-hop path to any destination in the network, and so any misforwarding will result in the packet leaving the group from the wrong interface.

Generalizing these observations, intuitively, router groups in networks with tree-like topologies or full-mesh-like topologies will tend to have excellent error detection performance.

Now we continue to study how effectively the router group monitoring approach can detect single forwarding error in real network topologies. Figure 3.4 shows the results. We can make two observations from this graph. First, as the router group size increases, the fraction of detectable mis-forwarding cases decreases but only slowly. When the group size increases to 50% of the network size, the detection rates are still as high as 80% for most of the topologies. These results based on real topologies demonstrate that router group monitoring can be highly effective in practice.

3.1.2.2 Multiple Forwarding Errors

Next, we consider the case where multiple dependent forwarding errors exist in the network. It is hard to predict the detection rate of multiple dependent forwarding errors because they can interact with each other. For example, after one router forwards a flow to a wrong path, the second misbehaving router on the wrong path might forward the flow back to the correct path. On the other hand, if the first misbehaving router fails to direct the flow to a wrong exiting interface, the second misbehaving router may increase the chance of the flow leaving from a wrong exiting interface by mis-forwarding it again. Therefore, the overall detection rate when having multiple dependent errors depends both on the network topology and the locations of the errors.

To better understand the detection rate for multiple dependent forwarding errors, we conduct the static analysis on the same set of real topologies. Specifically, we introduce 2 and 3 dependent forwarding errors on distinct routers inside each router group. The results are shown in Figure 3.5 and Figure 3.6. We can see that some topologies have higher detection rates than the 1-error case, while the other topologies have lower detection rates. Results based on synthetic topologies also confirm that the detection rate of two dependent errors is not consistently better or worse than the one error case. However, it is worth noting that even multiple dependent errors co-exist and even if we use 50% of the nodes in the network as the router group, for most of the networks (except RF-3) we have studied, the average detection rate is still higher than 65%. Another interesting observation is that when the number of dependent errors increases from 2 to 3, the detection rates for all topologies also increase.

3.1.2.3 Relation Between 1-Error Detection Rate and Multi-Error Detection Rate

In this section, we ask the question: If a router group is effective in detecting one forwarding error, will it also be effective in detecting multiple forwarding errors?

To answer the above question, we first define the overlap ratio metric as the percentage of the 10% router groups with highest 1-error detection rates that also belong to the 10%

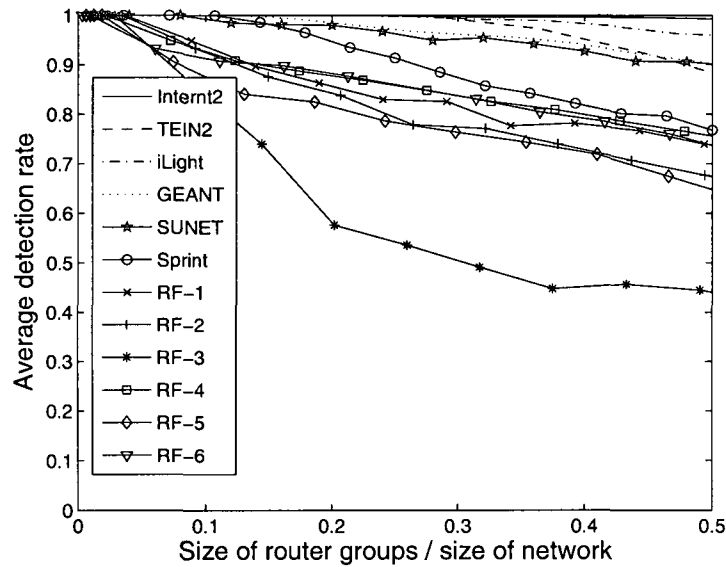


Figure 3.5 : Average detection rate of two dependent forwarding errors.

router groups with the highest 2 or 3-error detection rates. Figure 3.7 shows the result for all topologies. All topologies have high overlap ratios. Let us take the RF-6 topology as an example, the result shows that for the 10% router groups having highest 1-error detection rates, 89% of them are also among the 10% router groups having highest 2-error detection rates, and 88% of them are also among the 10% router groups having the highest 3-error detection rates.

In the rest of this thesis, we use the 1-error detection rate to characterize the effectiveness of a router group.

3.1.3 Detection Rate of Packet Filtering Errors

There are two types of packet filtering errors. The first type packet filtering error drops packets mistakenly, which is equivalent to the packet dropping error. If a packet filter drops a packet by mistake, the system can detect that the packet is missing from a periphery interface. Therefore, the first type of packet filtering error can be easily detected. The

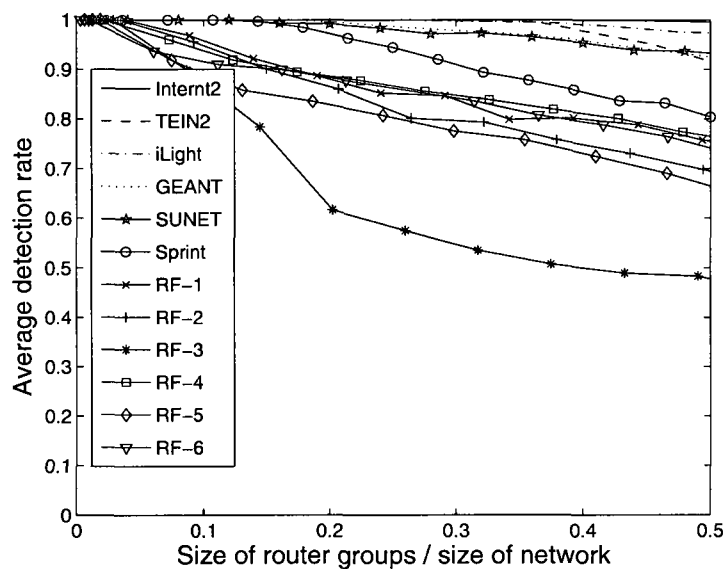


Figure 3.6 : Average detection rate of three dependent forwarding errors.

second type of packet filtering error fails to drop certain packets that are supposed to be dropped. We call the second type of packet filtering error as the filter-bypass error. When a filter-bypass error occurs inside a router group, whether it will be detected by monitoring the periphery interfaces depends on the distribution of packet filters inside the group. If the flow encounters another packet filter that is designed to drop it as well before it leaves the group, then the specific filter-bypass error will not be detected. On the other hand, if the flow leaves the group, then a periphery interface will see the unexpected flow so that the error will be detected. Since it is hard to obtain sensitive filter configurations used in real networks and the state-of-the-art synthetic filter generators (e.g., using [TT05a]) cannot capture certain critical characteristics we need (e.g., filter rules placement throughout the network), we decide to conduct the detection performance study on a particular kind of filter: Unicast Reverse Path Forwarding (uRPF). uRPF is a simple technique available on most Cisco and Juniper routers to block bogus packets. It assume symmetrical routing. Basically, if the reply to a packet would not go out the interface this packet came in from,

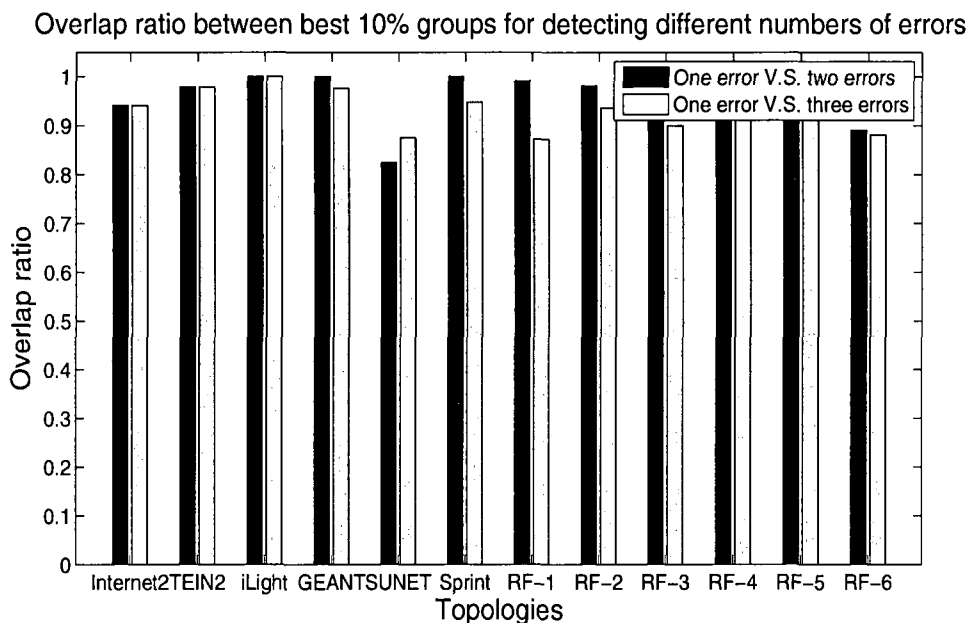


Figure 3.7 : Overlap ratio between the 10% router groups with highest 1-error detection rates and the 10% router groups with the highest 2 or 3-error detection rates.

then this is a bogus packet and should be dropped. For the seven real network topologies, we randomly introduce one forwarding error and two uRPF errors into the network, and then compute the detection rate for random router groups with up to 8 routers. The results are shown in Figure 3.8. As can be seen, the detection rate of uRPF filter errors is very high for all the network topologies. Even for groups with 8 routers, we can still detect 99.9% of uRPF errors in all the networks.

3.2 Analytical Model for Error Detection Rate

As we have shown in Section 3.1, router group monitoring can be highly effective in detecting trajectory errors in real topologies. However, for router group monitoring to be practical, those effective router groups with high trajectory error detection rates must be identified more efficiently than using the exhaustive hop-by-hop analysis approach.

One straight-forward way to avoid exhaustive analysis is to use sampling. For example,

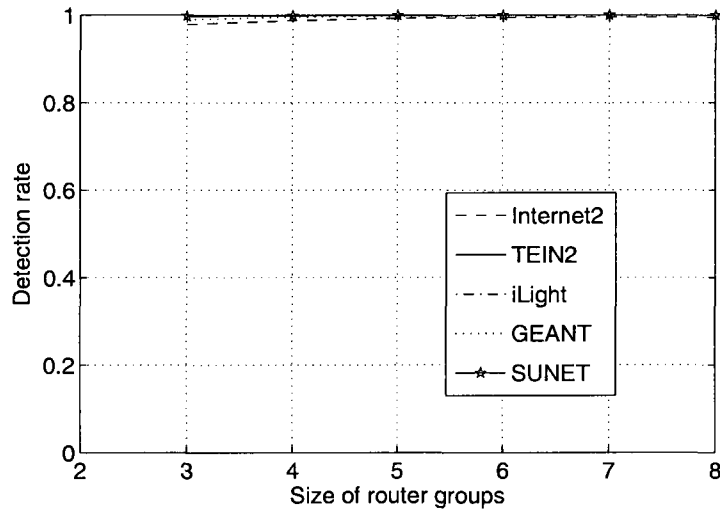


Figure 3.8 : The detection performance for uRPF errors.

given a router group, instead of analyzing all errors, we only analyze a small subset of randomly selected errors to estimate the overall detection rate of the router group. Another approach is to develop an analytical model for quickly estimating the detection rates of different router groups. The analytical model should require much less computation than the static analysis approach. In this section, we will first present our analytical model, which only depends on some simple structural and routing metrics of router groups, and then we will compare the prediction accuracy of both the sampling approach and the analytic approach in Section 3.2.2.

3.2.1 Contributing Factors of Error Detection Rate

Three major contributing factors affecting the forwarding error detection rate have been identified as follows:

Router group size: As shown in Figure 3.4, the size of a router group is an important factor affecting its detection rate. Specifically, the average detection rate decreases with the increase of router group sizes. Given a router group, its size is easy to calculate. It is

also not surprising that the size of a router group is important to its error detection rate. In a singleton router group with only one router, any error will be detected immediately. On the other hand, given a larger router group, a mis-forwarded packet is more likely to be self-corrected, i.e., it might fall back to its original routing path and leaves the router group from the original correct interface, thus the trajectory error might not be detected by this particular router group.

Number of exiting interfaces: Given a destination dst outside of the router group, a periphery interface If_i is called an *exiting interface* for dst if the interface's host router uses If_i as its direct next hop interface to route to dst . The router is called an exiting router accordingly. Given a particular destination, we can count how many periphery interfaces are exiting interfaces by scanning routing tables of routers having at least one periphery interface. The average number of exiting interfaces can be determined across all possible destinations. Intuitively, this factor characterizes how “diverse” the routing paths from inside the router group to a particular destination outside are. Please note that this metric is not the same thing as the number of periphery interfaces. One router group can have many periphery interfaces, but all the routers inside the group may only use a small number of periphery interfaces to route to any particular destination.

To illustrate why the number of exiting interfaces is important to a router group's error detection rate, Figure 3.9 (a) shows a router group with only one exiting interface If_1 with respect to the destination R_F . Since If_1 is the only exiting interface to R_F , when a forwarding error occurs (say R_B), it will be self-corrected by the router group (i.e., mis-forwarded packets end up leaving from the only exiting interface) unless a routing loop is formed. On the other hand, Figure 3.9 (b) shows a router group with two exiting interfaces (If_1 and If_2) for destination R_F , then a mis-forwarded packet is more likely to leave from the wrong exiting interface (If_2 in this example), allowing the error to be detected.

Connectivity of a router group: Given a router group, its connectivity is related to many topological characteristics of this group, such as average node degree, average outgoing degree (i.e., for each node, how many of its edges are connecting itself to nodes

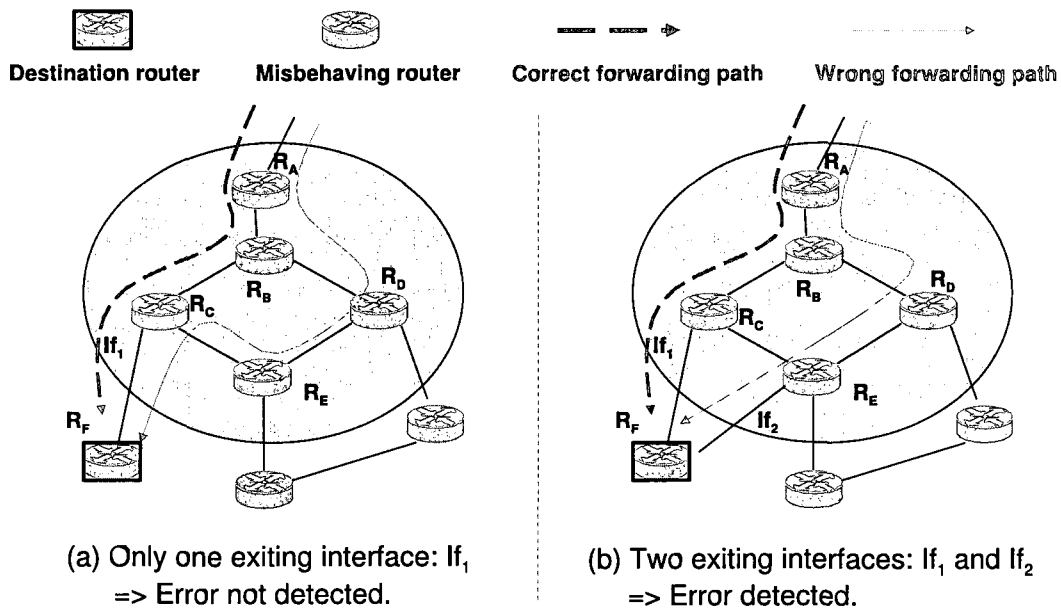


Figure 3.9 : Illustration of how the number of exiting interfaces impacts the error detection rate.

outside of the group), average internal degree (i.e., for each node, how many of its edges are connecting itself to other nodes inside the group). All these metrics are very easy to calculate. Intuitively, the connectivity can impact how likely a mis-forwarded packet will be self-corrected inside the group and how likely a forwarding loop will be formed.

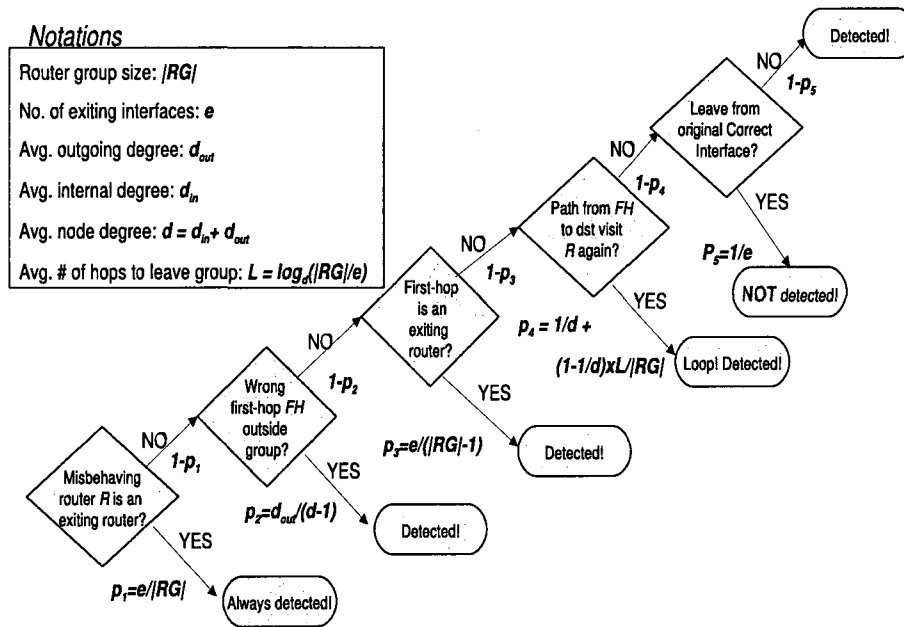
To illustrate why connectivity can impact forwarding error detection rate, Figure 3.3(a) shows a router group with 5 routers, and the topology inside the router group has a cycle. There are two potential paths P_1 and P_2 between periphery interfaces If_1 and If_3 . Let's assume path P_1 is the correct path for a particular flow F . Flow F should enter the router group at the interface If_1 and leave at interface If_3 , following path P_1 inside the router group. However, if the router R_B has a forwarding error, it may forward the flow to router R_D as opposed to router R_C . The flow F will take path P_2 inside the router group, but it still leaves the group at interface If_3 . In this case, this router group cannot detect router R_B 's forwarding error. Generally, given a router group, if there are more than one paths between an ingress interface and an egress interface, it is possible that some forwarding errors inside

a particular router group cannot be detected from the periphery interfaces. Note that the same forwarding error may be detectable by using a different router group. In contrast, as shown in Figure 3.3(b), if a group of routers are connected in a tree topology, there is only one path between each ingress interface and egress interface. If the router R_B misforwards the flow F to the wrong path P_2 , the flow F will either leave the router group at the wrong interface If_4 , or be stuck between R_B and R_D (assuming R_B consistently misforwards the packet to R_D). Therefore, in a tree topology router group, any single forwarding error is guaranteed to be detected by monitoring the periphery interfaces because there are no redundant routing paths for a misforwarded packet to go back to the original correct path.. Also, if a network has a full-mesh topology with all links having equal link weight, a misforwarding inside any router group is guaranteed to be detected. This is because all nodes have a one-hop path to any destination in the network, and so any misforwarding will result in the packet leaving the group from the wrong interface.

Generalizing these observations, intuitively, router groups in networks with tree-like topologies or full-mesh-like topologies will tend to have excellent error detection performance.

3.2.2 Analytical Model

Based on the three important factors identified above, we have developed an analytical model for accurately and quickly estimating the 1-error detection rate of a router group. We first define some notations we will use in our discussion. A router group's size is denoted as $|RG|$. The average number of exiting interfaces is e , which is also the number of exiting routers. The average outgoing degree and internal degree are d_{out} and d_{in} respectively. The average node degree $d = d_{out} + d_{in}$. In deriving the model, we assume that each router has an equal chance to be the misbehaving one, and the misbehaving router will forward the affected flow to one random incorrect next-hop. This assumption is made to make sure that the errors analyzed in our model do not have a biased distribution. We also assume that any two routers inside a router group are equally likely to have a link connecting them.



$$\text{Detection rate} = p_1 \times 1 + (1-p_1) \times (p_2 + (1-p_2) \times (p_3 + (1-p_3) \times (p_4 + (1-p_4) \times (1-p_5))))$$

Figure 3.10 : Analytical formula for estimating error detection rate.

In addition, we assume that all links have equal weight and the correct trajectories follow shortest path routing. These assumptions are made to facilitate our model derivation. In a real network, these assumptions may not always accurately hold. However, our evaluation using 12 real network topologies in Section 3.2.3 shows that the derived model is robust and it can accurately estimate the detection rates of router groups even if those real topologies have different connectivity and non-uniform links weights.

We denote the misforwarding router as R_m . To accurately model the error detection rate of a router group, the first thing to note is that if R_m is an exiting router with respect to destination dst and is misforwarding packets destined to dst , then the error is guaranteed to be detected. Recall that an exiting router for dst is supposed to forward packets destined to dst directly out of the router group using its exiting interface. If it fails to forward the packet using its own exiting interface and assuming this is persistent, then the misforwarded

packets will not leave the router group on the correct interface. Therefore, the forwarding error by an exiting router can always be detected. The probability that a router inside a router group is an exiting router is $p_1 = e/|RG|$.

However, if R_m is not an exiting router, its misforwarded packets may or may not leave the group from the correct exiting interface. When the non-exiting router R_m misforwards packets, it has the probability $p_2 = d_{out}/(d - 1)$ to misforward packets directly out of the group using one of its outgoing edges, where $d - 1$ is the number of all possible wrong next hops. In this case, the error will be caught by the device that is monitoring the corresponding periphery interface because the packets are observed from incorrect interfaces. On the other hand, R_m could misforward to a wrong next hop (also the first hop router) FH inside the router group. Since we assume only R_m in the router group is misbehaving, FH is a well-behaved router. Now we have two possibilities. The first possibility is that FH is an exiting router. The probability that FH is an exiting router is $p_3 = e/(|RG| - 1)$, where $|RG| - 1$ is the number of correct routers inside the router group. If this is the case, then FH will use its own exiting interface to route the packets out of the group. These packets therefore leave the group from an incorrect interface and will be caught because the correct trajectory follows the shortest path implies that FH does not lie on the correct trajectory. The other possibility is that FH is a non-exiting router. We model the length of the path $Path_{FH}$ from FH to its exiting router as $L = \log_d(|RG|/e)$, where $|RG|/e$ is simply the average number of nodes using one particular exiting interfaces. If $Path_{FH}$ does not contain R_m , then the probability of $Path_{FH}$ leaves from the same exiting interface as R_m should have used is modeled as $p_5 = 1/e$, in which case the error cannot be detected by this router group. On the other hand, if $Path_{FH}$ does contain R_m , then a loop is formed, which will cause the error to be detected since the packet is missing from its expected exiting interface. We estimate the probability of $Path_{FH}$ containing R_m as $p_4 = 1/d + (1 - 1/d) \times L/|RG|$, where $1/d$ estimates the probability of FH sending the packet directly back to R_m forming a 1-hop loop and $L/|RG|$ estimates the probability of a path of length L contains a node R_m out of $|RG|$ possible nodes in total.

Figure 3.10 gives a summary of the model and the final analytical formula for estimating detection rates.

3.2.3 Prediction Accuracy of Using Model vs. Sampling

We now evaluate the accuracy of both model-based and sampling-based detection rate prediction as follows: Up to 100 router groups of each size are randomly chosen from each topology. We first use static analysis to calculate the exact detection rate for each chosen router group. Then we use our model to predict the detection rate for each router group and record the computation time required. For the sampling based approach, we sample different percentages (up to 50%) of errors and then predict the overall detection rate by analyzing only the sampled errors. We also record the computation time used for different sampling percentages. Figure 3.11 compares the average prediction errors (defined as the absolute difference between the predicted detection rate and the correct detection rate) of both approaches when they use the same amount of computation time. As can be seen, first of all, the model's average prediction error is smaller than 0.05 on most topologies. Therefore, the model successfully captures the important characteristics of the error detection. Second of all, given the same amount of computation time, the model can predict the detection rates more accurately for most topologies. On iLight and RF-2, the sampling based approach works only slightly better than the model based approach. Then for the ten topologies where our model works better than sampling, we study how much more time is needed to generate results as accurate as the model. Our results show that the sampling based approach generally needs a few times more computation time to have the same accuracy as the model-based approach. For some topologies such as Sprint and GEANT, the sampling based approach needs 9 and 10 times more computation time to get the same prediction accuracy as the model.

The computation required by the model for computing the detection rate is significantly reduced compared against with the static analysis approach. Figure 3.12 shows the computation speedup comparison between the model based approach and the static analysis based

approach. For all topologies except TEIN2, the speedup is over 20 times when a router group contains 50% of the nodes in the network. For some large router groups in the large topologies, the speedup is up to 153 times. For example, given a desktop computer with an Intel Pentium 4 3.0 GHz CPU, 9 hours of computation time is used to compute the detection rates of 1000 random router groups in RF-6 topology by analyzing all errors inside each group, while it only costs 5 minutes of computation time for our analytical model. As expected, the computation saving of using our model increases when the network becomes larger.

Another useful property of our model is that the pair-wise ranking order among router groups is mostly preserved, which is very important to our router group selection algorithm in Section 3.3, where we use predicted detection rates to help select the most effective router groups. Specifically, for each pair of router groups, we will predict which one has a larger detection rate using our model and then validate the results using the detection rate calculated by the static analysis approach. Figure 3.13 shows the percentage of router group pairs whose order is preserved by the model. For example, the model correctly predicts the ranking order for 89.2% of router group pairs in the Sprint topology.

3.3 Router Group Selection Algorithm

We have demonstrated that router group monitoring is effective in detecting traffic trajectory errors. We have also proposed a model to predict the detection rates of router groups. The next problem is to design an algorithm to choose a suitable set of router groups for the system to monitor for each monitoring period.

As explained in Section 1, existing traffic trajectory monitoring algorithms monitor different subsets of packets during different monitoring periods. If during each monitoring period, $x\%$ of packets are monitored, then $\frac{100}{x}$ monitoring periods are needed to cover all traffic. Similarly, router group monitoring can also be performed period by period. However, in order to reduce monitoring overhead, in router group monitoring, only up to M interfaces are monitored during each monitoring period, where M is no larger than the

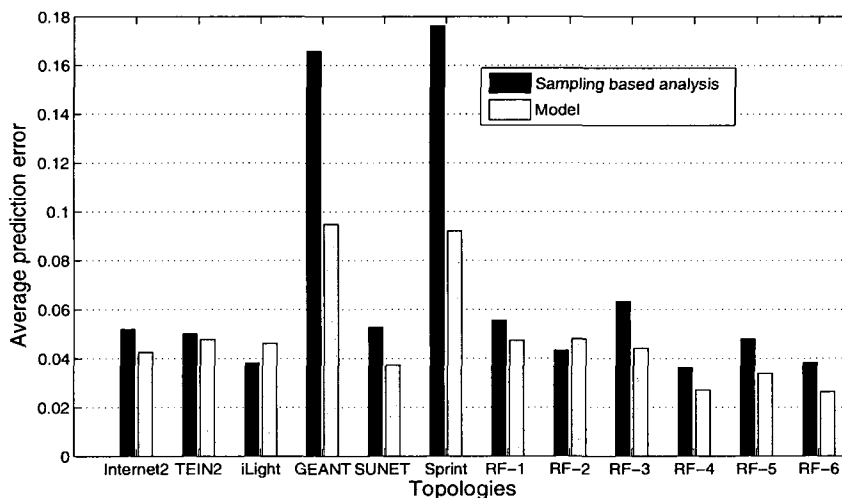


Figure 3.11 : Prediction errors comparison of model-based and sampling-based approaches.

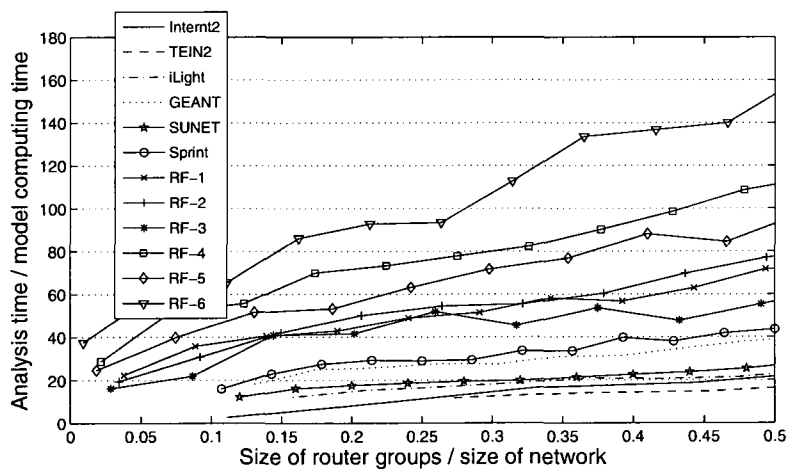


Figure 3.12 : Computational speedup of computing error detection rates using model versus computing error detection rates using full simulation approach.

total number of interfaces of the network and the M interfaces are periphery interfaces of the set of monitored router groups.

A good router group selection algorithm should (1) provide complete trajectory error

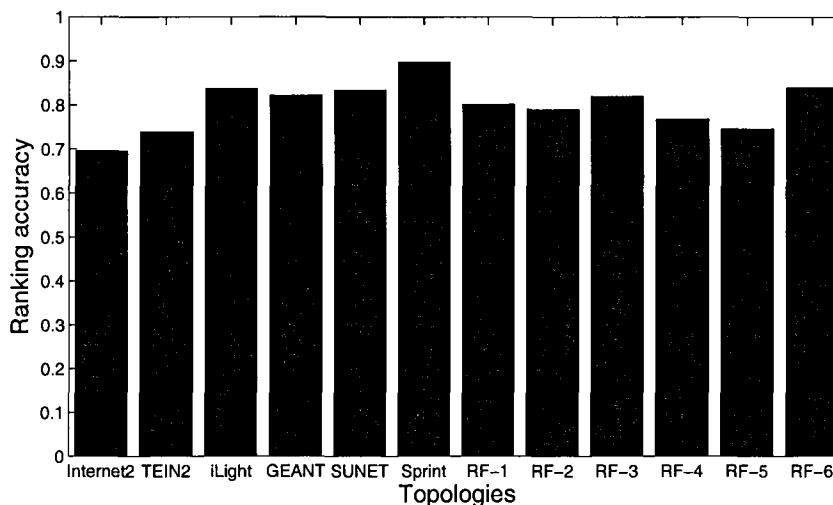


Figure 3.13 : The model accurately preserves the ranking order among pairs of router groups.

detection coverage, which is the *correctness* requirement elaborated in Section 3.3.1, and (2) detect errors as quickly as possible, which is essentially the *optimality* requirement discussed in Section 3.3.2.

3.3.1 Correctness of Router Group Selection Algorithm

As we explained, when using router group monitoring, some interfaces (M) are monitored during each monitoring period. Thus, the first concern of the router group selection algorithm is whether it can guarantee complete trajectory error detection coverage. One straight-forward way to satisfy this requirement is to treat each single router as a router group and then always include all the singleton router groups for monitoring. This is however unnecessary. We give a more general sufficient condition as follows.

Lemma 1. *To guarantee that all observable trajectory errors¹ are eventually detected, it*

¹A fundamental requirement for trajectory error detection in *any* approach is that the evidence of an trajectory error must be observable by other monitoring nodes. This thesis does not address errors that are

is sufficient to select a set of router groups such that every router interface f_{ij} on a node v_i connecting to a node v_j is an end of a cut edge $(v_i, v_j) \in E$ of a selected router group RG , with $v_i \in RG$ and $v_j \in V \setminus RG$. Intuitively, f_{ij} is a periphery interface of a router group RG facing outward.

Every error, whether it is a mis-forwarding, or a packet dropping or a filter-bypass exhibits itself in one of two ways: (1) a packet that should have been observed on an interface is not observed; (2) a packet that should not have been observed on an interface is observed. If a router interface is a periphery interface of a router group RG facing outward, then that interface's behavior is monitored when router group RG is monitored. Therefore, any error involving that interface will be caught.

3.3.2 Optimality of Router Group Selection Algorithm

Given the sufficient condition, we can now easily tell whether a set of router groups can provide complete error coverage. The next question is: how should we select router groups to monitor during each monitoring period so that we can not only achieve complete error coverage but also only iteratively monitor the *smallest* number of monitoring periods? This is the *optimality* requirement of the router group selection problem.

Minimizing the total number of monitoring periods while providing complete error coverage is a hard problem. The reason can be intuitively explained as follows. Suppose each interface f_{ij} is involved in some number of errors. When the interface f_{ij} is inside (i.e. not on the periphery of) a router group RG_k , those errors involving f_{ij} can be detected with some probability $w(RG_k, f_{ij}) \in [0, 1]$. Therefore, once RG_k is selected for monitoring, the usefulness of monitoring other router groups that also contain f_{ij} will decrease accordingly. This interdependence makes it hard to determine an optimal selection of router groups.

not observable. For example, if one router mistakenly drops a packet destined to itself, then this error cannot be detected because it is not observable from outside.

3.3.2.1 Definition of the monitoring problem

We now formalize the router group selection problem as follows.

Notations:

- Let $G = (V, E)$ be a graph, where V is a set of nodes

$$V = \{v_1, v_2, \dots, v_n\}$$

and E is a set of edges $E \subseteq V \times V$.

- Each node v_i is associated with a set of *interfaces*

$$F_{v_i} = \{f_{ij} : (v_i, v_j) \in E\}$$

The set of all the interfaces in the graph $G = (V, E)$ is $F = \bigcup_{v_i \in V} F_{v_i}$ ². For any subset $A \subseteq V$,

$$F_A = \{f_{ij} : (v_i, v_j) \in E \wedge (v_i \in A) \wedge (v_j \notin A)\}$$

- Given function α , for any subset $A \subseteq V$ the monitoring weight function is defined as³: for each $f_{ij} \in F$,

$$w(A, f_{ij}) = \begin{cases} 1 & \text{if } f_{ij} \in F_A \\ \alpha(A, i, j) \in [0, 1] & \text{if } v_i \in A \wedge v_j \in A \\ 0 & \text{if } v_i \notin A \wedge v_j \notin A \end{cases}$$

- A *Monitoring* is defined as a multiset of sets of subsets of V ⁴:

$$\begin{aligned} \mathcal{A} &= \{\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_m\} \\ &= \{\{A_{11}, \dots, A_{1n_1}\}, \{A_{21}, \dots, A_{2n_2}\}, \dots, \\ &\quad \{A_{m1}, \dots, A_{mn_m}\}\} \quad (A_{ij} \subseteq V) \end{aligned}$$

where $m \geq 1$ and $n_i \geq 1$.

²The set of interfaces is decided by the set E

³Intuitively, the weight $w(A, f_{ij})$ means that the errors involved with the interface f_{ij} can be detected using group A with probability of $w(A, f_{ij})$. Trajectory errors on periphery interfaces can always be immediately detected, while errors on other interfaces may or may not be detected.

⁴(1) Nodes in A_{ij} may not necessarily be connected. (2) A_{ij} and A_{ik} may have overlapped nodes.

Optimization objective:

Find the smallest *Monitoring* \mathcal{A} such that for any \mathcal{A}' we have $|\mathcal{A}| \leq |\mathcal{A}'|$, where \mathcal{A} and \mathcal{A}' satisfy the following two constraints:

- Given a *Monitoring* $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_m\}$, for each $1 \leq i \leq m$, $\mathcal{A}_i \in \mathcal{A}$,

$$|\bigcup_{A \in \mathcal{A}_i} F_A| \leq M$$

Where $M \geq 1$.

- Given a *Monitoring* $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_m\}$ and a constant $0 \leq \beta \leq 1$, for each $f_{ij} \in F$ ($1 \leq i, j \leq |V|$),⁵:

$$1 - \prod_{1 \leq k \leq m} \prod_{A \in \mathcal{A}_k} (1 - w(A, f_{ij})) \geq \beta$$

We generally want to find the smallest *Monitoring* given $\beta = 1$ and a small constant M .

In the above problem formulation, during monitoring period i , a set of router groups $A_i = A_{i1}, A_{i2}, \dots, A_{in_i}$ are monitored concurrently, where A_{ij} and A_{ik} could overlap with each other. We have studied a special case of the above problem, where for any $j, k \in [1, n_i]$ and $j \neq k$, A_{ij} and A_{ik} always have no overlap. We have proved that as long as $M < |F|$, the above special case is a NP hard optimization problem [Kre92]. Please refer to Section 3.3.2.2 for the complete proof. We believe the general case is also NP hard and we are currently working on the proof.

3.3.2.2 The complexity of the monitoring problem

Definition 1 (optimization problems). An optimization problem \mathcal{P} is characterized by the following quadruple of objects $(I_{\mathcal{P}}, SOL_{\mathcal{P}}), m_{\mathcal{P}}, goal_{\mathcal{P}}$, where:

⁵Intuitively, this means that after m groups of monitoring, all the error involved on the interface f_{ij} have been detected with a probability of at least β .

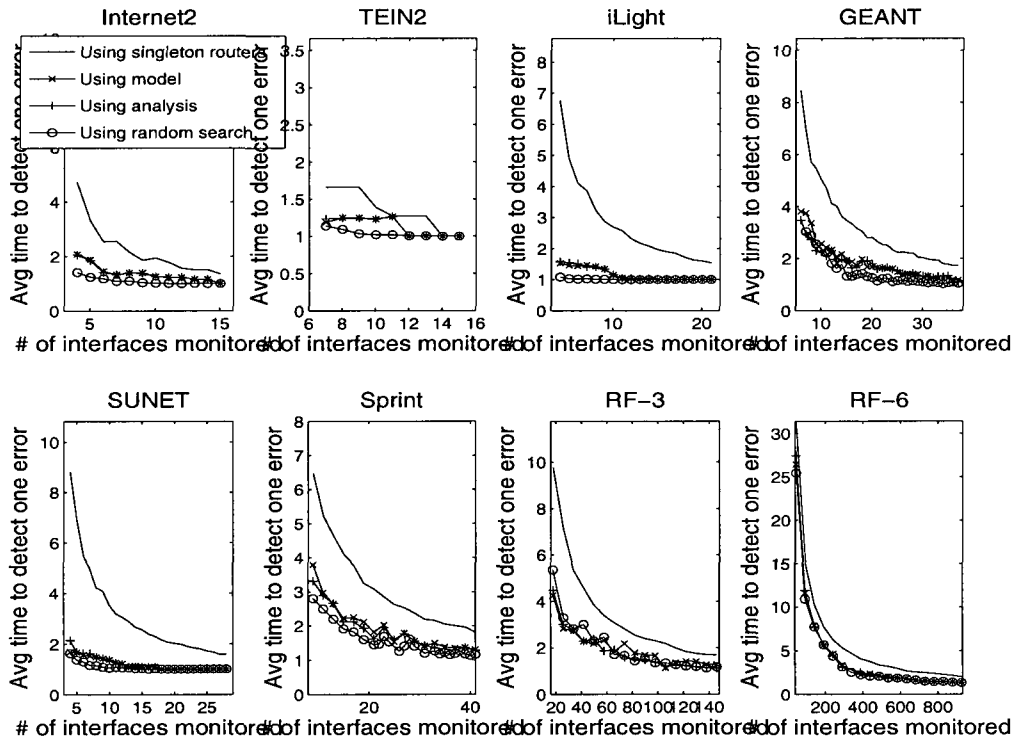


Figure 3.14 : Comparison of average error detection speeds of different router group selection approaches.

1. $I_{\mathcal{P}}$ is the set of instances of \mathcal{P} ;
2. $SOL_{\mathcal{P}}$ is a function that associates to any input instance $x \in I_{\mathcal{P}}$ the set of feasible solutions of x ;
3. $m_{\mathcal{P}}$ is the measure function, defined for pairs (x, y) such that $x \in I_{\mathcal{P}}$ and $y \in SOL_{\mathcal{P}}$. For every such pair (x, y) , $m_{\mathcal{P}}(x, y)$ provides a positive integer which is the value of the feasible solution y ;
4. $goal_{\mathcal{P}} \in \{MIN, MAX\}$ specifies whether \mathcal{P} is a maximization or a minimization problem.

It is worth noting that any optimization problem \mathcal{P} has an associated decision problem \mathcal{P}_D . In the case that \mathcal{P} is a minimization problem, \mathcal{P}_D asks, for some $k > 0$, for the existence of a feasible solution y of instance x with value $m(x, y) \leq k$. For any optimization problem \mathcal{P} , the corresponding decision problem \mathcal{P}_D is not harder than the problem \mathcal{P} . If an optimization problem \mathcal{P} has its associated decision problem \mathcal{P}_D be NP-hard, then \mathcal{P} is an *NP-hard optimization problem* [add the citation here].

For a graph $G = (V, E)$, given function $\alpha \in [0, 1]$, number $\beta \in [0, 1]$, and a positive integer M , the problem of finding an optimized β -complete M -monitoring \mathcal{A} is trivial if M is sufficiently big. This is because we can have $\mathcal{A} = \{\mathcal{A}_1\}$ and $\mathcal{A}_1 = V$, which leads to optimal solution with $|\mathcal{A}| = 1$. However, for a reasonably small M , it is not easy to find an efficient algorithm to find a solution. In what follows, we look at a special case of the problem, and prove that it belongs to the family of NP-hard optimization problem. We thus conclude that the optimal monitoring problem can be generally very complicated.

Consider a graph $G = (V, E)$, let $\beta = 1$, and function $\alpha(A, i, j) \equiv 0$ for any $A \subseteq V$, $1 \leq i, j \leq |V|$. If the graph and a number M satisfies that $|F_A| \leq M$ for all $A \subseteq V$ and $|F_{A_1} \cup \dots \cup F_{A_m}| > M$ for any $m \geq 2$, $A_i \subseteq V$, then each M -monitoring $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_t\}$ must satisfies $|\mathcal{A}_i| = 1$ ($1 \leq i \leq t$). Thus we can equivalently write any M -monitoring as $\mathcal{A} = \{A_1, A_2, \dots, A_t\}$, $A_i \subseteq V$. We denote this problem of finding an optimized β -complete M -monitoring of G in this special case as \mathcal{P} . We have:

Theorem 1. *The problem \mathcal{P} defined above is an NP-hard Optimization Problem.*

Proof. The problem \mathcal{P} has an associated decision problem \mathcal{P}_D : Is there a β -complete M -monitoring $\mathcal{A} = \{A_1, A_2, \dots, A_t\}$ of G , such that $t \leq k$?

To show \mathcal{P} is a NP-hard optimization problem, we only need to show \mathcal{P}_D is NP-hard. The decision problem \mathcal{P}_D can be equivalently restated as:

For an integer k , find k subsets of V such that every edge is a cut edge for at least one of the subsets.

We show that 2^k -colorability problem \mathcal{P}_C can be Turing reduce to \mathcal{P}_D in polynomial time.

A polynomial reduction has two parts: converting the problem and converting the solution. Converting the problem is a no-op, where the graph $G = (V, E)$ is kept unchanged. Conversion of the solution is as follows.

Suppose you have subsets A_0, A_1, \dots, A_{k-1} of V satisfy that every edge is a cut edge for at least one of the subsets. Then color each vertex $v \in V$ with color $b_{k-1}b_{k-2} \cdots b_1b_0$, where b_i is the i -th bit of the color written in binary notation, and $b_i = 1$ iff $v \in A_i$. Then for every edge (v, u) , for some i , A_i contains exactly one of u and v . So the colors of u and v differ at bit i . Therefore this is a 2^k -coloring of the given graph.

Suppose contrariwise that a given graph is 2^k -colorable. Then write out the colors in binary notation as above. For each j , form A_j so that it contains a vertex v_i iff its color has $b_i = 1$. Then for any edge (u, v) , their colors differ at some bit i , and so A_i contains exactly one of u and v .

The reduction takes $O(k|V|)$ time.

Since n -colorability is NP-complete for any $n \geq 3$, it follows that \mathcal{P}_D is NP-hard for any $k \geq 2$.

3.3.3 Heuristic Algorithm for Router Group Selection

Given the above definition, we now give a heuristic algorithm for selecting a set of router groups that achieves complete coverage, has bounded concurrent monitoring overhead, and in practice provides timely error detection.

Input of the algorithm: A positive integer M , a set of n router group candidates denoted as $RG_{candidates} = RG_1, RG_2, \dots, RG_n$, and the $w(RG_k, f_{ij})$ function defined in Section 3.3.2.1.

M is the maximum number of interfaces that the system can concurrently monitor and it should be determined by the operator based on resource constraints. We assume that the maximum degree of any router in the network is no larger than M . $RG_{candidates}$ should

contain a large number of diverse router groups of different sizes in order to provide enough opportunity for the selection algorithm to explore. We cannot include all possible router groups into $RG_{candidates}$ for large networks. Thus, we first randomly generate a number of router groups of each size, and then select up to K router groups from each size with the highest predicted detection rates. All singleton routers are always included in $RG_{candidates}$, which is important for guaranteeing that the selection algorithm eventually terminates. The $w(RG_k, f_{ij})$ function specifies that if RG_k is monitored, then the errors involving interface f_{ij} can be detected with a probability of $w(RG_k, f_{ij})$.

Output of the algorithm: Given M , function $w(\cdot)$ and $RG_{candidates}$, the output of the algorithm should be m sets of router groups, T_1, T_2, \dots, T_m , where $T_i \subseteq RG_{candidates}$. Then we can iteratively monitor all m sets of router groups one by one. If we can sample $x\%$ packets at each moment, then $\frac{100}{x}$ periods are needed for each T_i . That is, in total, $m \times \frac{100}{x}$ monitoring periods are needed to cover all traffic.

Algorithm's intuition: The main idea of our heuristic algorithm is to keep greedily selecting a set of router groups that have the potential to detect most uncovered errors to form a new set of router groups T_i until the sufficient condition is satisfied. We define an uncovered error function $E(f_{ij}) \in [0, 1]$ on each interface f_{ij} to represent the fraction of uncovered errors on f_{ij} at the current moment. At the beginning of the algorithm, none of interface f_{ij} 's errors have been covered by any selected router group, so $E(f_{ij}) = 1$. Once a router group RG_k containing $f_{i,j}$ has been selected for monitoring, we update $E(f_{ij})$ as follows: $E(f_{ij}) = E(f_{ij}) \times (1 - w(RG_k, f_{ij}))$.

Suppose $RG_{selected}$ is the set of selected router groups at this moment. Now we can define the selection weight of a router group RG_k as follows:

$$W(RG_k) = \begin{cases} 0 & \text{if } RG_k \in RG_{selected} \\ \sum_{f_{ij}} w(RG_k, f_{ij}) \times E(f_{ij}) & \text{if } RG_k \notin RG_{selected} \end{cases}$$

The router group selection algorithm is as follows:

- 01: $RG_{candidates} = \{\text{All singleton routers}\}$;
- 02: FOR $m = 2 : |V| - 1$

03: Randomly generate up to T router groups containing
 m routers and $\leq M$ periphery interfaces
and then select up to $K \leq T$ router groups with highest
predicted detection rates and put into $RG_{candidates}^m$;

04: $RG_{candidates} = RG_{candidates} \cup RG_{candidates}^m$;

05: END-FOR

06: $RG_{selected} = \{ \}$;

07: $A = \{ \}$;

08: $E(f_{ij}) = 1$, for $\forall f_{ij} \in F$;

09: $period = 1$;

10: WHILE($\sum E(f_{ij}) > 0$)

11: AvailableIFs = M ;

12: $A_{period} = \{ \}$;

13: WHILE AvailableIFs > 0 AND $\sum E(f_{ij}) > 0$

14: Find $RG_i \in RG_{candidates}$ with largest $W(RG_i)$
and with \leq AvailableIFs periphery interfaces,
if multiple choices exist, pick the largest group,
if no such choice exists, break the WHILE loop;

15: $RG_{candidates} = RG_{candidates} \setminus \{RG_i\}$;

16: $RG_{selected} = RG_{selected} \cup \{RG_i\}$;

17: $A_{period} = A_{period} \cup \{RG_i\}$;

18: $\forall f_{ij}, E(f_{ij}) = (1 - w(RG_i, f_{ij})) \times E(f_{ij})$;

19: Update $W(RG_j)$ for $\forall RG_j \in RG_{candidates}$;

20: AvailableIFs -= # periphery interfaces of RG_i ;

21: END-WHILE

22: $A = A \cup A_{period}$;

23: $period = period + 1$;

24: END-WHILE

25: RETURN A ;

Algorithm termination and correctness: Since we assume M is no smaller than the largest router degree in the network, each router is eligible to form a singleton router group while not violating the resource constraint. Since $RG_{candidates}$ includes all singleton router

groups, the selection algorithm can always return the singleton router groups. Therefore, the algorithm is guaranteed to terminate and return a set of router groups that has complete coverage.

3.3.4 Performance of Heuristic Router Group Selection Algorithm

In this section, we evaluate the performance of our heuristic algorithm. In the experiments, we use $K = 10$ to initialize $RG_{candidates}$. We study the performance of the algorithm using various M , as long as M is no smaller than the maximum degree of the network topology.

We use two different approaches of estimating function $w(RG_k, f_{ij})$. The first approach is based on static analysis, so we can accurately know the $\alpha(RG_k, i, j)$ function. The second approach is based on our detection rate model. Given a router group RG_k , suppose its predicted detection rate is $detection_k \in [0, 1]$ and suppose the router group RG_k contains $p_k \in [0, 1]$ fraction of periphery interfaces and accordingly $(1 - p_k)$ fraction of non-periphery interfaces. If we assume all internal non-periphery interfaces in RG_k have the same $\alpha(k)$ values, then we have $detection_k = p_k \times 1 + (1 - p_k) \times \alpha(k)$, i.e., $\alpha(k) = (detection_k - p_k) / (1 - p_k)$.

As a baseline for comparison, we also include the performance of singleton router based selection algorithm, whose $RG_{candidates}$ only contains all singleton router groups. In order to estimate how close our heuristic algorithm is to the real optimal group selection, we also compare with a bounded random search based approach. Specifically, given a topology and its $RG_{candidates}$, we will randomly select a multiset of sets of router groups for monitoring and then we can compute a corresponding average detection speed by introducing 10,000 random forwarding errors uniformly distributed across all nodes for all possible destinations. We repeat this random group selection process 10,000 times and keep the best detection speed we found. Please note that performing 10,000 random search is very expensive. For example, given the RF-6 topology, 66 hours of computation time is used to finish on a desktop computer with an Intel Pentium 4 3.0 GHz CPU. On the other hand, it only costs 16 minutes of computation time for our algorithm.

For each topology, we introduce up to 10,000 forwarding errors uniformly distributed across all routers for all possible destinations one by one. We then statically analyze to see how many monitoring periods it will take for each approach to detect each introduced error. We then present the average detection speed of all four approaches on different topologies in Figure 3.14. We omit the results for RF-1, RF-2, RF-5 as they are qualitatively similar to those of RF-3. The results for RF-4 is qualitatively similar to those of RF-6 and are also omitted. As we can observe, first of all, the detection speeds of the approach based on the model predicted detection rates are very close to the one using static analysis across all topologies. Secondly, the detection speeds of our approach are also very close to the bounded random search based approach, though our approach requires much less computation time. For some topologies such as RF-3, our heuristic algorithm is better for some M . This indicates our heuristic algorithm is effective in quickly selecting a good set of router groups for monitoring. Thirdly, our algorithm outperforms the singleton router groups based approach for all topologies. Especially when M is a small value, our approach can detect an error a few times faster than the singleton router group based approach. This performance gain comes from the fact that we are covering much more routers at any moment, though both approaches monitor the same number of interfaces and have the same overhead.

3.3.5 Discussion

In our problem formulation, we assume no constraints on which routers can be used for monitoring, that is, all routers are assumed to be homogeneously powerful. However, routers in real networks might be very heterogeneous. For example, some routers may even not have the monitoring capability. Fortunately, we can always use standalone passive traffic monitoring devices (e.g., [flob]) to tap on the corresponding network links to perform the monitoring function. In addition, some low-end routers might only afford up to a certain sampling rate due to resource constraints in hardware or software. In this case, we can either use standalone passive traffic monitoring devices for monitoring or we need to

carefully set the sampling rate on all periphery interfaces to not to exceed the the required resource constraints on the slowest monitoring device. If certain routers need to be taken offline for the scheduled maintenance, then the operator should plan ahead and calculate a new set of router groups for monitoring according to the specific topology change. If topology change is caused by other dynamic network events such as link failures, it can be learned from the dynamic routing protocol messages such as OSPF LSAs. To quickly respond to the dynamic topology change, different sets of router groups with respect to different potential network events should be computed in advance as well. How to incrementally update the set of monitored router group to efficiently accommodate unexpected dynamic events is one of our future work.

3.4 Applications of Router Group Monitoring

In this section, we show how the router group monitoring technique can improve the efficiency of trajectory error detection based on Trajectory Sampling and Fatih. The basic Trajectory Sampling algorithm monitors all interfaces in the network and samples the same subset of packets at the same time. Then, information about sampled packets is sent to a centralized collector for analysis. The basic Fatih algorithm, on the other hand, monitors all interfaces that are used in forwarding packets, although as we shall see this is nearly the same as monitoring all interfaces in practice. Fatih also samples the same subset of packets at the same time. The fingerprints of the sampled traffic belonging to each network path will be exchanged among the monitors along that path for analysis.

The router group monitoring technique can be used to select a subset of network interfaces to be monitored under Trajectory Sampling or Fatih. This translates into reduced monitoring overhead and/or faster trajectory error detection without sacrificing the completeness of coverage.

3.4.1 Applying to Trajectory Sampling

In Trajectory Sampling, all network interfaces in the network will sample the same subset of packets (say, 1% of all traffic) during the same monitoring period. Different subsets of packets will be sampled for different monitoring periods to achieve complete coverage.

3.4.1.1 Scenario One: Improve Detection Speed While Keeping the Reporting Traffic Overhead Constant

In this scenario, we want to keep the reporting overhead (i.e., how many messages are sent to the collector per period) constant so that we do not overwhelm the collector.

Suppose we can vary the sampling rate in a small range from 1% to 5%. Can router group monitoring improve the trajectory error detection speed while keeping the reporting overhead constant? To maintain the same reporting overhead, when we increase the sampling rate m times, we decrease the number of concurrently monitored interfaces by m times accordingly. The overall reporting overhead is maintained at the same level as sampling all interfaces in the network with a 1% rate.

Figure 3.15 shows the result. If we use a 5% sampling rate and allow the concurrent monitoring of 20% of the interfaces, the detection speedup over baseline Trajectory Sampling (i.e., sampling 1% on all interfaces) is at least 2 times and for some topologies the detection speedups are more than 4 times. The detection speedup comes from the fact that when we increase the sampling rate, we can rotate the set of monitored router groups more quickly. For example, if we use a 5% sampling rate, we only need to monitor each set of router groups 20 periods then we can rotate to a new set of router groups that can detect another set of errors. Specifically, taking SUNET as an example. If we assume that each monitoring period lasts one minute and the router group monitoring approach monitors 20% of all the interfaces with a sampling rate of 5%, then it will take the router group monitoring approach 25 minutes to detect all errors, while it will take 105 minutes for the original Trajectory Sampling.

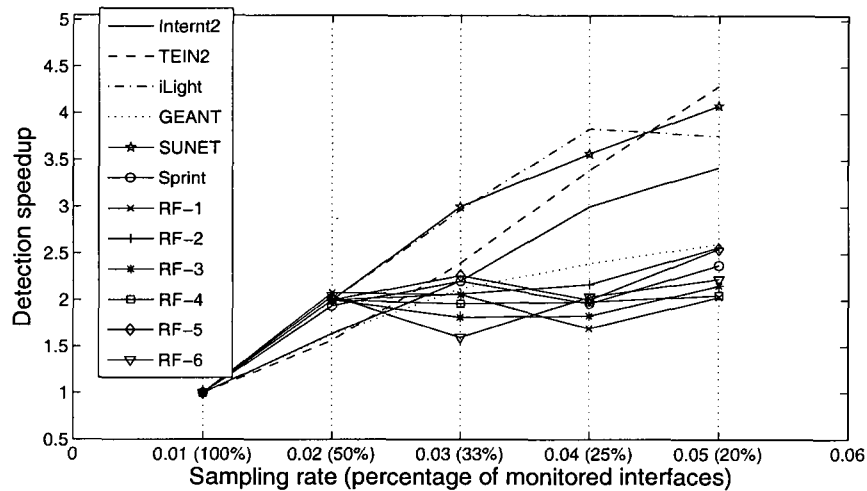


Figure 3.15 : Detection speedup when varying the sampling rate and the maximum number of interfaces concurrently monitored.

3.4.1.2 Scenario Two: Reduce Reporting Overhead While Keeping Same Detection Speed

In this scenario, we assume a fixed 1% sampling rate. Then we want to study what fraction of interfaces we have to monitor to keep the same detection speed as monitoring all interfaces simultaneously. Figure 3.16 shows the result. As can be seen, for certain topologies such as iLight, concurrent monitoring of 33% of the interfaces are enough to provide the same detection speed as baseline Trajectory Sampling. For most of the topologies, monitoring roughly 50% of the interfaces concurrently is enough to detect errors as quickly as baseline Trajectory Sampling. Specifically, taking RF-6 as an example. Assuming that each interface forwards 13,000 active flows per second on average [spr]. Given a 5% sampling rate, each interface can sample 650 active flows per second on average. Because each NetFlow record is 64 bytes, each interface will generate 332.8 Kbps of traffic. Since there are a total of 1944 interfaces in RF-6 topology, 646.9 Mbps of reporting traffic will be generated. On the other hand, the router group monitoring approach will only generate about 329.9 Mbps of reporting traffic while having the same detection speed.

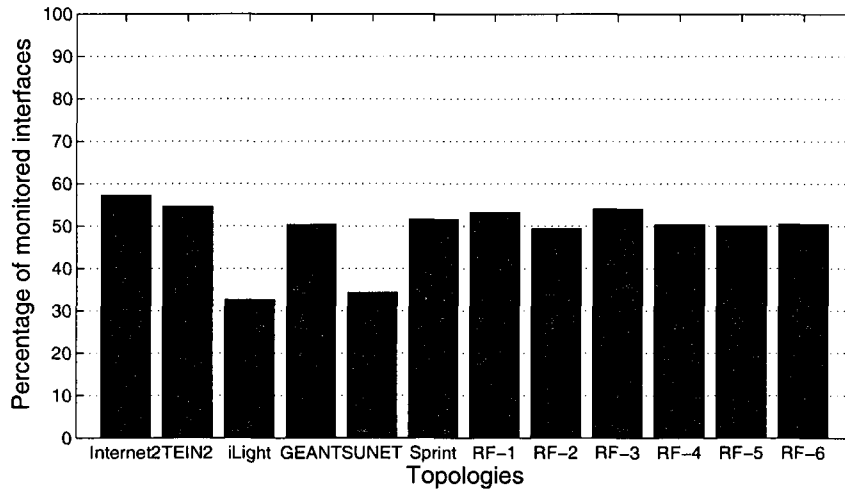


Figure 3.16 : Percentage of monitored interfaces required to achieve the same detection speed as the original Trajectory Sampling.

3.4.2 Applying to Fatih

In Fatih, each router r_i needs to maintain certain traffic information for *each* 3-path-segment containing itself. A 3-path-segment is a subpath with length 3. The traffic information Fatih maintains for each path segment is the fingerprints (e.g., hash values of the packets) of all the packets r_i forwarded along the monitored path segment. Periodically, router r_i exchanges the fingerprints information with other routers on the same 3-path-segment. Because all 3-path segments r_i, r_j, r_k are monitored, then if r_j dropped or misforwarded packets, r_i and r_k can detect this error when they exchange traffic information.

For the purpose of trajectory error detection, we can use the router group monitoring technique to reduce the monitoring overhead by only having each periphery router to maintain information about what traffic it will forward to other periphery routers in the same router group. Therefore, while in baseline Fatih each router keeps a set of information for each path segment, in contrast, with router group monitoring, only periphery routers need to maintain information for other periphery routers.

We evaluate the benefits of applying router group monitoring to Fatih. First of all, we compare the number of interfaces monitored with and without router group monitoring while keeping the detection speed the same in Figure 3.17. Since Fatih needs to monitor every 3-path segments, for many topologies, all interfaces need to be monitored. The interfaces will not be monitored if the corresponding link is not used or only used in an end-to-end path with length 2. Applying router group monitoring allows much fewer interfaces to be monitored while having the same detection speed and detection accuracy.

Next we evaluate the fingerprints communication overhead saving after using router group monitoring. In this experiment, we assume the same amount of traffic is sent between each pair of nodes in the network, and we study the fingerprint exchanging overhead with and without router group monitoring. The results are shown in Figure 3.18. The communication overhead of the baseline Fatih is normalized to 100 units. As can be observed, applying router group monitoring can reduce dramatically the fingerprint communication overhead for all topologies. For certain topologies, the overhead reduction is more than 80%. To understand the absolute reporting overhead reduction, we first take RF-6 as an example. Following the same assumption in Section 3.4.1, we assume that ten packets on average will be sampled for each flow. If each hash value is 8 bytes, then a total of 808 Mbps of traffic will be sent to the collector by all links. By employing router group monitoring approach, the reporting overhead can be reduced from 808 Mbps to 266 Mbps while keeping the same detection speed.

3.5 Related Work

Network measurement and monitoring are important for many network management applications. However, measurement and monitoring often incur high overhead. Therefore, a constant theme in many related research is to improve the efficiency of measurement and monitoring techniques. The goal of our technique is to specifically improve the efficiency of trajectory error detection. In the following, we discuss some previous work on improving the efficiency of monitoring and measurement for other important applications.

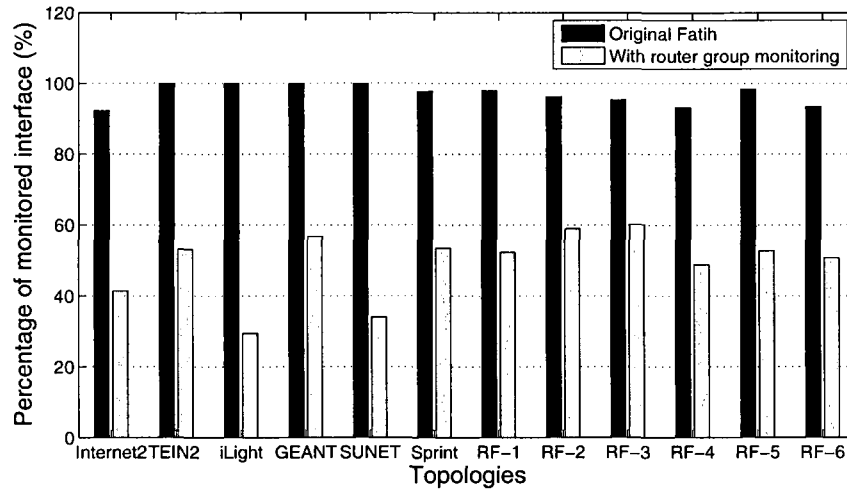


Figure 3.17 : Router group monitoring helps Faith reduce the number of monitored interfaces.

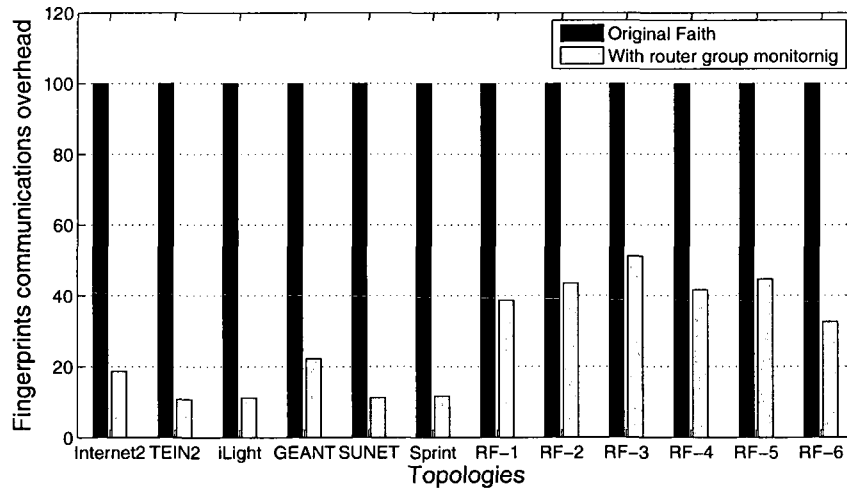


Figure 3.18 : Router group monitoring helps Faith reduce communication overheads.

WATCHERS [BCP⁺98, HAB00] maintains several packet counters at routers and uses inconsistencies found in these counters among different routers to detect forwarding errors. Because it only uses course-grained counters, it is only capable of detecting dropping er-

rors. Sekar *et al* [SRW⁺08] presented a new flow monitoring system, cSamp. cSamp can improve the flow monitoring coverage by enabling routers to coordinate and to sample different flows. Their goal is however not to identify the trajectory errors in flows. Therefore, cSamp can only tell which flows are in the network, but it does not know the actual trajectories of the monitored flows. Lee *et al* [LWK06] presented a secure split assignment trajectory sampling (SATS) technique. The idea is to enhance trajectory sampling by letting each pair of routers to sample different subsets of packets to improve monitoring coverage. However, SATS cannot detect forwarding error unless a forwarding loop is formed causing some packet loss. In addition, it only detects packet dropping error with a certain probability. On the other hand, our approach can detect both forwarding and dropping errors. Like our router group monitoring technique, cSamp and SATS also introduce a spatial dimension to their solution in the sense that different parts of the network perform different heterogeneous tasks to improve the overall efficiency of traffic monitoring. At the interface traffic sampling level, Estan *et al* [EKMV04] proposed a set of efficient techniques to adapt the NetFlow sampling rate in order to better control resources consumption. Kompella and Estan [KE05] proposed an efficient flow measurement solution called Flow Slices to control and reduce CPU usage, memory usage and reporting bandwidth of flow measurements. Our router group monitoring technique currently assumes packet level monitoring. Applying these flow based monitoring techniques can potentially improve the efficiency of trajectory error detection further. These efficient interface-level sampling techniques are orthogonal and complementary to our work.

Router group monitoring could also be viewed as a sort of traffic monitor placement technique because it identifies interfaces that need or need not be monitored for trajectory error detection. However, our technique is only designed for the trajectory error detection problem. The selected interfaces always form a boundary around a group of routers, so we can track how the traffic flows through the network. The more general monitor placement problem has been extensively studied for various problem settings. However, all these monitor placement techniques aim to sample more flows, instead of learning the actual spa-

tial trajectories of flows. That is, they are designed to sample different flows at different locations while our approach is designed to sample the same flow at different locations so that we can infer their complete trajectories. Horton and Lopez-Ortiz [HLO03] addressed the monitor placement problem in an active monitoring infrastructure to efficiently measure delays and detect link failures. Suh *et al* [SGKT05] used optimization techniques to place monitors and set the sampling rates in order to maximize the fraction of IP flows being monitored. They first find the links that should be monitored and then run another optimization algorithm to set sampling rates. Chaudet *et al* [CFR05] not only studied the tap device placement problem for passive monitoring but also the beacon placement problem for active monitoring. Their goal is to minimize the number of tap devices used for passive monitoring and to find the optimal locations for placing the beacons. Similarly, Cantieni *et al* [CIB⁺06] proposed mechanisms to optimally select links to be monitored and select sampling rates in order to achieve specific measurement tasks with high accuracy and low overhead. Jackson *et al* [JMS⁺07] studied the monitor placement problem using the current Internet topology. Their goal is to choose a set of locations to maximize the chance of covering all possible communication pairs in the Internet. Note that in general, how to optimally choose interfaces to monitor for trajectory error detection is still an open problem. Zang *et al* [ZN05] investigates the problem of deploying NetFlow with optimized coverage and cost in an IP network. It aims to solve the Optimal NetFlow Location Problem (ONLP) for a given coverage ratio. However, it only samples flows at fixed points instead of monitoring their actual spatial trajectories.

Chapter 4

Constructing Shared HyperCuts Decision Trees for Multiple Packet Filters

As explained in the introduction, the detector needs to maintain control states (e.g., forwarding tables and packet filters) of multiple routers. To hold the control states of multiple routers, a large amount of memory is required. Therefore, one research challenge is how to efficiently store the control states of multiple routers in the detector. Efficient data structure for maintaining multiple forwarding tables has been proposed by Fu and Rexford [FR08a]. In this thesis, we propose to efficiently store multiple packet filters using a shared data structure based on the HyperCuts decision tree [SBVW03a], which is widely adopted by commercial routers and firewalls such as Nevis Networks [nev], Cisco Systems [cisc] and NewBridge Networks [new]. We experimentally show that naively classifying packet filters into shared HyperCuts decision trees may significantly increase memory consumptions and heights of trees. To help decide which subset of packet filters should share a HyperCuts decision tree, we first identify a number of important factors that collectively impact the efficiency of the resulting shared HyperCuts decision tree. Based on the identified factors, we then propose to use machine learning techniques to predict whether any pair of packet filters should share a tree. Given the pair-wise prediction matrix, a greedy heuristic algorithm is used to classify packet filters into a number of shared HyperCuts decision trees. Our experiments using both real packet filters and synthetic filters show that our shared HyperCuts decision trees require considerably less memory while having the same or a slightly higher average height than the separate trees. In addition, the shared HyperCuts decision trees enable concurrent lookup of multiple packet filters sharing the same tree.

The rest of the chapter is organized as follows: In Section 4.1, we briefly introduce

the HyperCuts data structure and then extend the original HyperCuts data structure to support multiple packet filters. Section 4.2 first uses a simple experiment to show that naively clustering packet filters to shared HyperCuts decision trees may result in significantly increased memory consumption and heights of trees. Section 4.3 presents our approach of clustering packet filters into multiple shared HyperCuts decision trees. The idea is to first identify some important factors that can affect the efficiency of the constructed shared HyperCuts decision tree. Based on the identified factors, we then leverage machine learning techniques to predict which pairs of packet filters should share a tree. Given the pair-wise prediction, a heuristic clustering algorithm is used to cluster all packet filters into a number of shared HyperCuts decision trees. We evaluate the accuracy of the pair-wise prediction and the performance of the constructed shared trees in Section 4.4. Another application of using the proposed shared decision trees is discussed in Section 4.5. We discuss related work in Section 4.6.

4.1 Background

4.1.1 Packet Filter Notations

Informally, a packet filter of size n is a list of n ordered rules $\{R_1, R_2, \dots, R_n\}$ that collectively define a packet classification policy. Each rule R_i is composed of two parts: a combination of D values, one for each selected packet header field, and an associated action. The most commonly used five packet header fields are: source IP address, destination IP address, source port, destination port, and protocol type. Each of the D values specified in R_i could be a single value or an interval of values or the special value ANY used to specify all possible legitimate values for that field. Typical actions associated with a rule include permit, deny, marking the ToS bit, etc. A packet P is considered to match the rule R_i if all the D header fields of P match the corresponding values in R_i . If P matches more than one rule, then the rule with the smallest index in the packet filter is returned. The associated action of the returned rule will be performed on P accordingly.

Rule ID	Source IP	Destination IP	Source port	Destination port	Protocol	Action
R_0	104.253.26.143/31	151.217.12.0/23	ANY	1489	TCP	<i>act0</i>
R_1	103.11.193.196/31	151.193.40.150/32	ANY	27000	TCP	<i>act0</i>
R_2	51.109.218.92/30	243.82.86.0/23	ANY	135	TCP	<i>act1</i>
R_3	133.202.88.44/30	78.87.20.226/31	ANY	[1300-1349]	TCP	<i>act2</i>
R_4	137.180.89.7/32	243.82.125.14/32	ANY	6789	TCP	<i>act1</i>
R_5	201.130.210.90/31	6.92.31.0/25	ANY	1533	TCP	<i>act0</i>
R_6	119.10.210.90/31	6.92.31.0/25	ANY	1526	UDP	<i>act0</i>
R_7	119.67.166.172/31	151.143.84.75/32	ANY	1521	TCP	<i>act3</i>
R_8	71.252.162.33/32	151.166.64.162/32	ANY	[1300-1349]	TCP	<i>act4</i>
R_9	209.137.112.252/31	151.248.122.158/32	ANY	[61200-61209]	TCP	<i>act2</i>

Table 4.1 : A simple packet filter example with 10 rules defined on five packet header fields.

A simple packet filter with 10 rules defined on five packet header fields is shown in Table 4.1.

4.1.2 The HyperCuts Data Structure and Algorithm

Decision trees have been shown to be a powerful data structure for performing packet classification by using geometric cutting [Tay05]. Several different variants of decision tree based packet classification algorithms (e.g., [Woo00] [GM99] [SBVW03a]) have been proposed. HyperCuts [SBVW03a] is considered to be one of the most efficient decision tree based packet classification algorithms. In this section, we will briefly introduce the HyperCuts data structure and algorithm. A more detailed discussion can be found in [SBVW03a].

A HyperCuts decision tree is composed of two types of nodes: internal nodes and leaf nodes. Each leaf node contains less than *BucketSize* number of rules, where *BucketSize* is a small constant (e.g., 4). The small number of rules stored in a leaf node will be linearly traversed to find the matched rule with the smallest index in the original packet filter. By contrast, an internal node contains more than *BucketSize* rules, so rules stored in the internal node have to further split to its child nodes.

The HyperCuts decision tree is efficient because it splits rules in internal nodes using the

information from multiple packet header fields. In contrast to HyperCuts, HiCuts [GM99] only splits rules on one packet header field at a time. In order to decide which subset of packet header fields to use to split rules on an internal node, the HyperCuts algorithm will first count the number of unique elements on each field for all rules stored on the node. Let us take the 10 rules in Table 4.1 as an example, the number of unique elements in all five fields is 10, 10, 1, 9, 2 respectively. The HyperCuts algorithm will then consider the set of fields for which the number of unique elements is *greater than the mean* of the number of unique elements for all the fields. For example, given a node holding the 10 rules in Table 4.1, the three fields of source IP, destination IP and destination port should be considered for cutting. After determining which set of fields to cut, the HyperCuts algorithm uses several heuristics to decide how many cuts should be performed on each field. More detailed discussions of those heuristics can be found in [SBVW03a]. However, it is worth noting that the number of child nodes that an internal node can be split into is limited by a factor of the number of rules stored in the node. The function is defined as $f(N) = spfac \times \sqrt{N}$, where N is the number of rules in the internal node and $spfac$ is a small constant with a default value of 2. This technique is used by both the HiCuts and the HyperCuts algorithms to reduce the memory consumption.

4.1.3 Extend the HyperCuts Data Structure and Algorithm

To allow multiple packet filters to share a HyperCuts tree, the original HyperCuts data structure and tree building algorithm need to be extended. Figure 4.1 (a) shows two separate HyperCuts trees, each of which only has one internal node (its root) and four leaf nodes. Figure 4.1 (b) shows the corresponding shared HyperCuts tree. As can be seen, the internal node on shared HyperCuts tree is the same as the one in the original HyperCuts tree. Each internal node only records the number of cuts performed on each field and a list of pointers to its child nodes. On the other hand, leaf nodes have to be slightly extended to support multiple packet filters sharing the tree. In the original HyperCuts tree, a leaf node is composed of a header (indicating the node is a leaf node) and a pointer to the set of rules

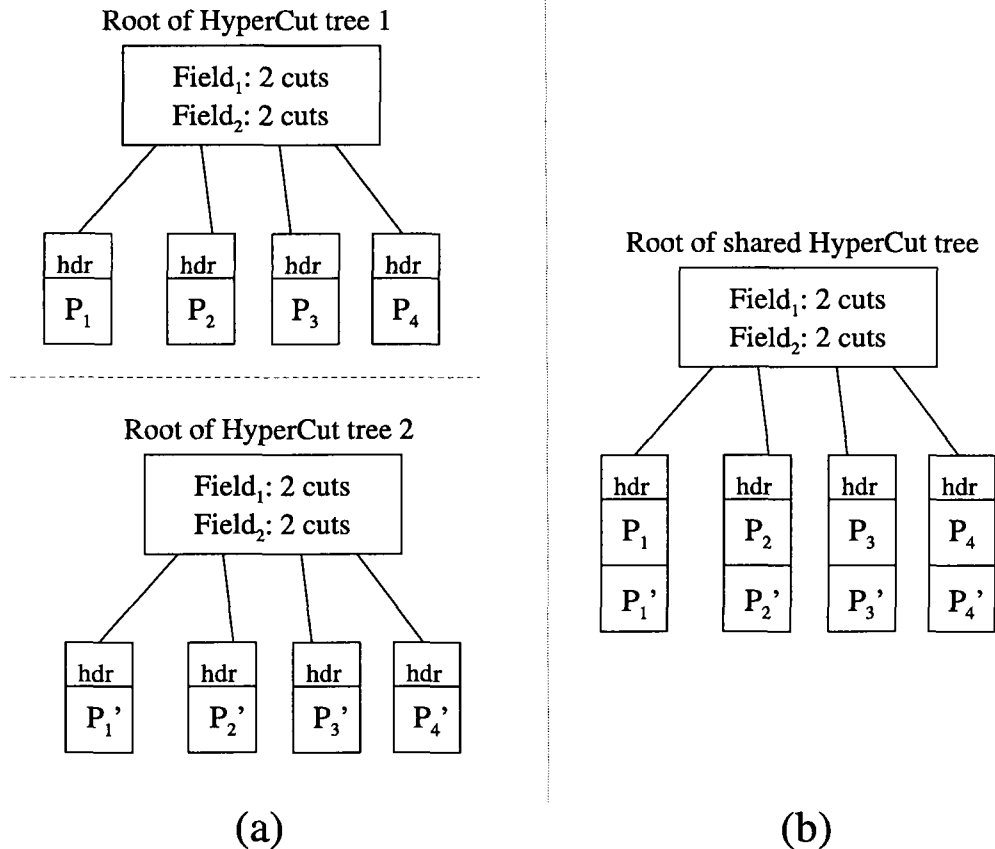


Figure 4.1 : Example of a shared HyperCuts tree: (a) Two separate HyperCuts trees. (b) The corresponding shared HyperCuts tree.

stored in this leaf node. In the shared HyperCuts tree of *two* packet filters, a leaf node is composed of the same header and *two* pointers, one for each packet filter. When a packet reaches a leaf node when searching the shared HyperCuts tree, since it knows which packet filter this packet is being matched, it will directly calculate which pointer it should access next. Therefore, the time to access a leaf node on the shared HyperCuts tree is still the same as in the original HyperCuts tree. In this simple example, by making the two packet filters share a tree, we saved one internal node and 4 headers of leaf nodes.

Now we continue to explain how we extend the original HyperCuts tree construction algorithm. The idea is to use a corresponding average value across all packet filters to

replace the value used in the original algorithm. For example, suppose that the two packet filters F_1 and F_2 are sharing a HyperCuts decision tree. Given an internal node on the shared tree, if the number of stored rules from each packet filter is N_1 and N_2 , then the number of child nodes this internal node can have is bounded by $spf_{ac} \times \sqrt{(N_1 + N_2)/2}$. Similarly, to decide the subset of fields for cutting on each internal node, we will first calculate the number of unique elements in each field on a per packet filter basis. Let us denote the number of unique elements for rules from F_1 and F_2 as u_{1j} and u_{2j} respectively, where $1 \leq j \leq D$. Then the number of unique elements on each field u_j for the current internal node is defined as $u_j = (u_{1j} + u_{2j})/2$. The rest of the algorithm is just the same as the original HyperCuts algorithm.

4.1.4 Efficiency Metrics of The HyperCuts Decision Tree

Given a constructed HyperCuts tree, we wish it consumes as little memory as possible. Thus, a natural metric of interest is **memory consumption**. In addition, we wish to do fast packet classification using the shared HyperCuts tree, so the tree search time (i.e., from the root to leaf nodes) is also important. We use the following two metrics to characterize the tree search time:

Average depth of leaf nodes: The depth of a leaf node is just the length of the shortest path from itself to the root. Assuming each leaf node has the same probability to be reached during a packet matching, then the average depth of all leaf nodes reflects on average how many internal nodes need to be accessed to terminate this tree search.

Height of the tree: This metric characterizes the largest number of internal nodes needed to be accessed for a packet to reach a leaf node. It corresponds to the worst case search time.

4.2 Challenges of Constructing Efficient Shared HyperCuts Decision Tree

To construct efficient shared HyperCuts decision trees, one key question to answer is: which subset of packet filters should share a HyperCuts decision tree so that the resulting shared tree is more efficient than a set of separate trees? In this section, we first introduce the filter data sets used in the thesis. We then experimentally show that naively letting multiple packet filters share a HyperCuts decision tree will significantly increase the memory consumption and height of the shared trees.

4.2.1 Filter Data Sets

We extracted a set of real packet filters from the configuration files of routers in a large-scale campus network [SRXM08] at Purdue University. We did not include the 260 packet filters that contain no more than *BucketSize* number of rules, because their corresponding HyperCuts decision trees just contain one root node. In our experiment throughout the thesis, we always set *BucketSize* as 4.

Because it is hard to obtain other real packet filters, a synthetic filter generator ClassBench [TT05b] is used to generate some synthetic filters. The ClassBench tool takes a parameter file as the input and then generates synthetic filters using the information stored in the input parameter file. We used three parameter files provided by ClassBench and they were originally generated from three real access control lists (ACLs) on Cisco routers. Given each parameter file, we generate two sets of 1,000 synthetic filters. The first set of 1,000 synthetic filters all contains 100 rules, while the size distribution of the second set of 1,000 synthetic filters follows an exponential distribution with the average value of 100. Please note that when generating synthetic filters with exponential size distribution, we also discard the filters containing no more than *BucketSize* rules.

Some basic statistics about the set of real packet filters and the six sets of synthetic filters are summarized in Table 4.2.

Data Set Name	Parameter File	Number of Filters	Size Distribution	Average Size	Minimum Size	Maximum Size
Purdue	N/A	140	N/A	21.5	5	763
Syn1-Exp	ACL1	1000	Exponential	98.21	5	1002
Syn1-100	ACL1	1000	Uniform size:100	100	100	100
Syn2-Exp	ACL3	1000	Exponential	101.9	5	910
Syn2-100	ACL3	1000	Uniform size:100	100	100	100
Syn3-Exp	ACL4	1000	Exponential	106.3	5	874
Syn3-100	ACL4	1000	Uniform size:100	100	100	100

Table 4.2 : Summary of basic statistics about the seven filter data sets.

4.2.2 Making Randomly Selected Packet Filters Share HyperCuts Trees?

In this section, we will use a simple experiment to show that extra care has to be taken in deciding which set of packet filters should share a tree. Naively making a set of randomly selected packet filters share a tree will significantly degrade the performance.

In our experiment, for each filter data set, we randomly choose n distinct filters, where n is a small number. Given the n selected filters, we first build a separate tree for each selected filter. Let us denote the memory consumption of the n trees as m_i , the average depths of leaf nodes of the n trees as d_i , and the heights of the n trees as h_i , where $1 \leq i \leq n$. Then we construct a shared HyperCuts decision tree to represent the selected n filters. Let us denote the memory of the shared tree, the average depth of leaf nodes in the shared tree and the height of the shared tree as m_{shared} , d_{shared} and h_{shared} . Now we can define the **memory consumption ratio** as $m_{shared} / \sum_{i=1}^n m_i$, the **average leaf depth ratio** as $d_{shared} / (\sum_{i=1}^n d_i / n)$, and the **tree height ratio** as $h_{shared} / (\sum_{i=1}^n h_i / n)$. The smaller the ratios are, the more benefits we obtain by making the n packet filters share a single HyperCuts tree. A ratio larger than 1 means that the shared tree has worse performance than n separate trees. Given each fixed n , we repeat the experiment 1000 times, i.e., we randomly select 1000 sets of n distinct filters for our experiment. We also vary n from 2 to 10.

Figure 4.2 (a) shows the average memory consumption ratio across 1000 runs for all 7 data sets. As can be seen, when the number of randomly selected filters increases, the mem-

ory consumption ratio becomes higher for all 7 data sets. This is because the more packet filters are randomly selected to share a tree, the harder it is to construct a HyperCuts tree suitable for all packet filters. When 10 randomly selected packet filters share a HyperCuts decision tree, it will consume 2 to 20 times more memory than simply using 10 separate trees. Figure 4.2 (b) shows the average of the average leaf depth ratios across 1000 runs. Similarly, the more packet filters are randomly selected to share a tree, the larger the ratios are. The tree height ratio results are very similar to the average leaf depth ratio results, so they are not shown here.

By comparing the memory consumption ratio and average leaf depth ratio, we can also observe that the average leaf depth ratio increases more rapidly with the increase of n than the memory consumption ratio does. The reason is that the sizes of all internal nodes in a HyperCuts tree are not the same. Please recall that the number of child nodes that an internal node can have is related to the number of rules stored in the node. Because those nodes closer to the root usually contain more rules, they accordingly have more child pointers (4 bytes for each pointer). Thus, internal nodes closer to the root are much larger than the internal nodes far from the root. This explains why a HyperCuts tree with doubled height consumes less than doubled memory.

4.3 Clustering Packet Filters to Construct Efficient Shared HyperCuts Decision Trees

As shown in Section 4.2, letting a set of randomly selected filters share a HyperCuts tree leads to increased memory consumption and average height of trees. In this section, we propose a novel approach to clustering packet filters to form efficient shared HyperCuts decision trees. In our approach, to help decide which subset of packet filters should share a tree, we first identify a number of important factors that collectively impact the efficiency of the resulting shared tree. Based on the identified factors, we then propose to use machine learning techniques to predict whether any pair of packet filters should share a HyperCuts decision tree. Given the pair-wise prediction on all possible pairs, a greedy heuristic algo-

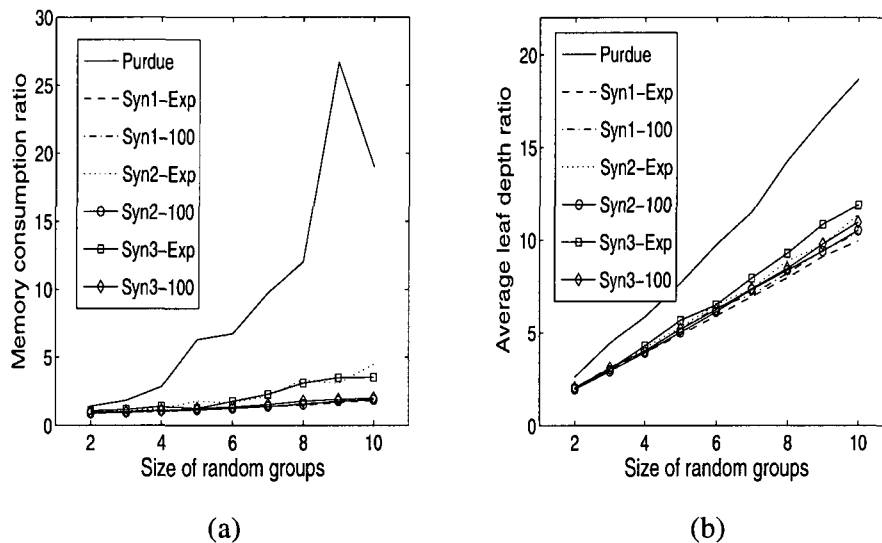


Figure 4.2 : (a) Memory consumption increases when randomly selected packet filters share a HyperCuts tree. (b) Average depths of leaf nodes increase when randomly selected packet filters share a HyperCuts tree.

rithm is used to classify packet filters into a number of shared HyperCuts decision trees.

4.3.1 Factors Affecting the Efficiency of the Shared Trees

In Section 4.3.1.1, we first present some important factors that can characterize each individual packet filter. We then study the relationships among those factors in Section 4.3.1.2. We found that some factors are highly correlated to each other. Finally, we show that the identified factors can all help decide whether any pair of packet filters should share a HyperCuts decision tree or not in Section 4.3.1.3.

4.3.1.1 Important Factors for Characterizing Individual Filter

According to our analysis, there are two different classes of factors that can characterize each individual packet filter:

Class-1 factors include some simple statistical properties of a packet filter itself. They

include the *size of the packet filter* and the *number of unique elements in each field*. To obtain the Class-1 factors, we do not need to build the HyperCuts decision tree for the packet filter. These factors are important because they are used in the HyperCuts tree construction algorithm. Thus, they can affect the structure of the final HyperCuts tree. However, only Class-1 factors are not enough to determine the structure or memory consumption of the final HyperCuts decision tree. Two packet filters with identical Class-1 factors may have very different tree structures. Therefore, we identify the second class of factors as follows. **Class-2 factors** represent the characteristics of the constructed HyperCuts decision tree. That is, the HyperCuts tree must be constructed to obtain the Class-2 factors of a packet filter. Because we want the final shared tree to have good performance, the *memory consumption of the tree*, the *average depth of leaf nodes* and the *height of the tree* are one part of the Class-2 factors. In addition, the *number of leaf nodes*, the *number of internal nodes* and the *total number of cuts on each field* are also included into the Class-2 factors, because they can more accurately reflect the actual structure and memory consumption of the HyperCuts tree. For example, the more nodes a tree has, the more memory it will generally consume. In addition, the total number of cuts performed on each field can reflect the relative importance of each field so it can impact the structure of the constructed tree.

4.3.1.2 Relationship Among Factors

The factors identified in Section 4.3.1.1 are not independent to each other. In fact, some factors are highly correlated to each other in the common case. We identify the following correlated pairs of factors:

Size V.S. Memory consumption: In the common case, the HyperCuts decision tree of a bigger packet filter tends to consume more memory. The reason is that the more rules a packet filter contains, the more times of splittings are needed to reduce the number of rules contained in nodes to *BucketSize*. Accordingly, more internal nodes and leaf nodes will be created, which will generally consume more memory. Please note that this is by no means a universal rule, i.e., a bigger packet filter could consume less memory than a

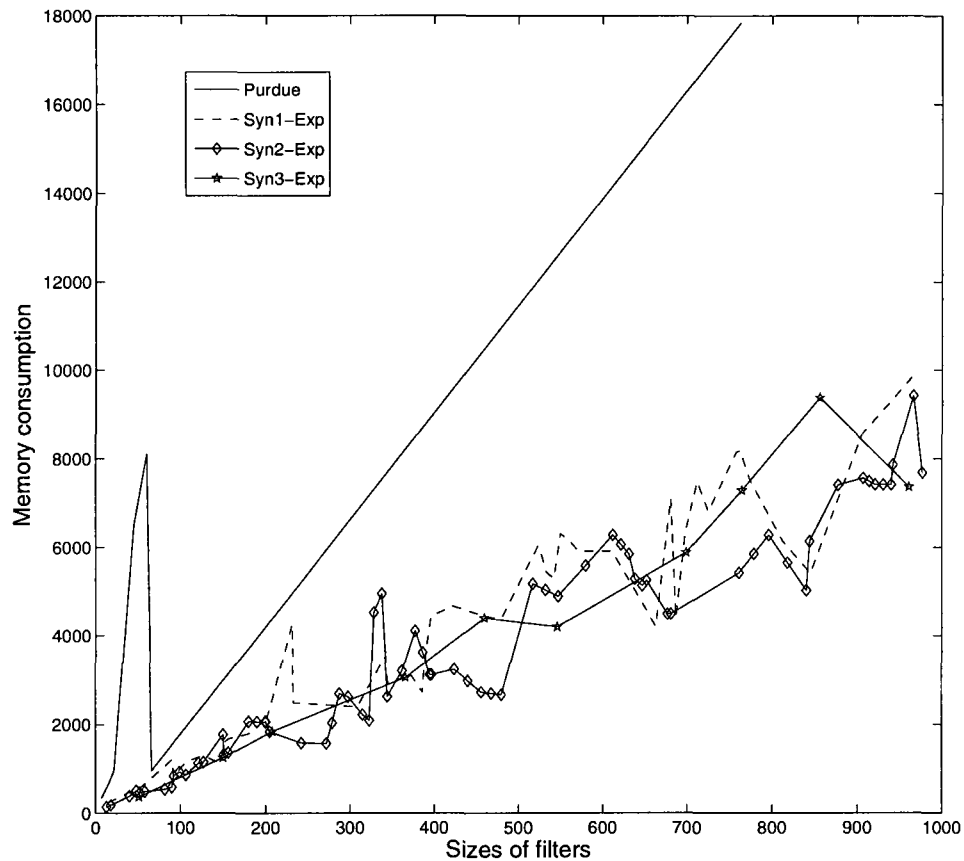


Figure 4.3 : Filter size V.S. memory consumption.

smaller packet filter does. For example, one bigger packet filter with more dissimilar rules may consume less memory than a smaller packet filter with similar rules. Figure 4.3 shows the relationship between the memory consumption and the packet filter size. As you can see, the memory consumption is generally increasing with the increase of sizes of packet filters.

The number of internal nodes V.S. The number of leaf nodes: Usually, the number of internal nodes on a tree is highly correlated with the number of leaf nodes. This is because

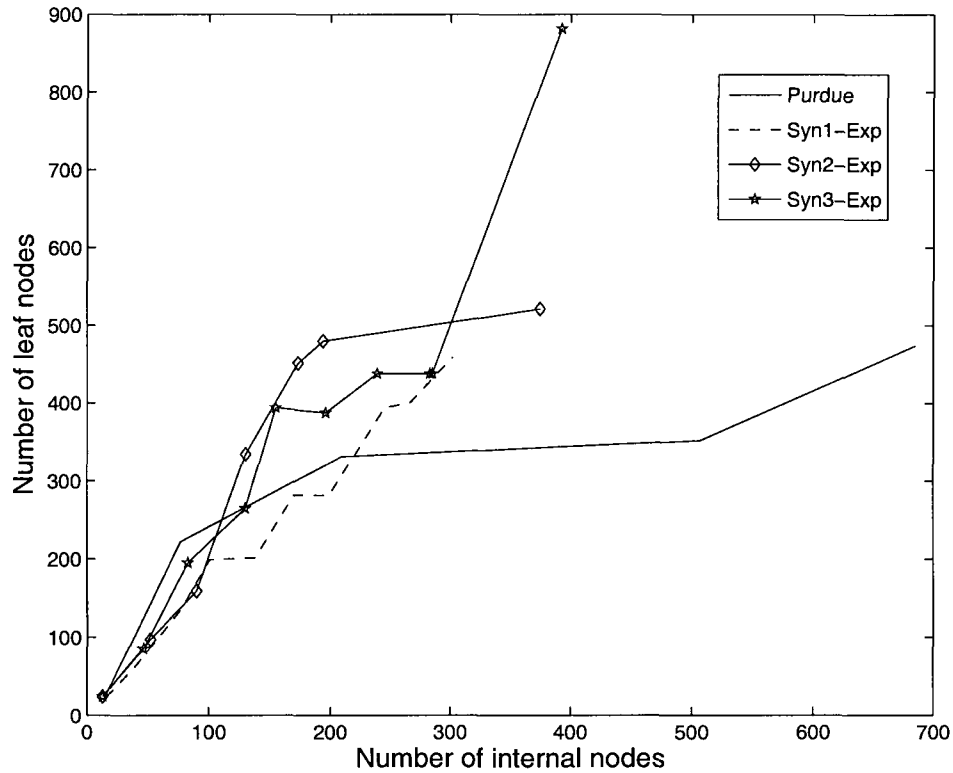


Figure 4.4 : Number of internal nodes V.S. number of leaf nodes.

that generally the more internal nodes a tree has, then the more distinct paths from the root to the bottom of the tree exist. Please note that each path will end at a leaf node, so more internal nodes will generally lead to more leaf nodes. Figure 4.4 shows the relationship between the number of internal nodes and the number of leaf nodes. As can be observed, the number of internal nodes generally increases with the increase of the number of leaf nodes.

Please note that a special tree such as a chain-like tree may have many internal nodes but very few number of leaf nodes. However, we consider those highly unbalanced trees as the non-common case.

Size V.S. The number of internal/leaf nodes: We have shown that the bigger a packet filter is, the more memory is generally consumed. The major reason is that the bigger the packet filter is, the more nodes the tree will generally have. Figure 4.5 shows the correlation between the filter sizes and the number of internal nodes. As can be seen, the number of internal nodes on a tree increases generally with the increase of the size of a packet filter. Since we have shown that the number of internal nodes is also correlated with the number of leaf nodes, the filter size is also correlated to the number of leaf nodes. Figure 4.6 shows the relationship between the filter sizes and the number of leaf nodes.

However, the number of nodes on a tree is just one important factor that can impact the memory consumption. The size of each internal node is also important. The size of an internal node depends on the number of children pointers it contains. The more children pointers a node contains, the bigger the node is. That is to say, not all internal nodes have the same size. Please note that some children pointers will be empty, so the number of children pointers on an internal node is usually smaller than the the number of children nodes.

Number of unique elements in each field V.S. Number of cuts in each field: The number of unique elements in a field is generally highly correlated with the number of cuts performed in the same field. The more unique elements there are in a field, the more rules are defined using this field, then the more important this field generally is. To quickly split rules in the packet filter, the fields with more unique elements will be more likely to be selected for cutting according to the HyperCuts algorithm. Therefore, the number of cuts in one field is generally well correlated with the number of unique elements in that field. Figure 4.7 shows the correlation between the number of unique elements in field 1 and the number of cuts in field 1.

Height of the tree V.S. The average depth of leaf nodes: The height of a tree is the maximum depth of all leaf nodes in the tree. In the common case, the higher a tree is, the larger the average depth of all leaf nodes is. However, the height of the tree is purely determined by the depth of the deepest leaf node, so the height of the tree is not as reliable as the average depth of all leaf nodes. One outlier leaf node with large depth will make the

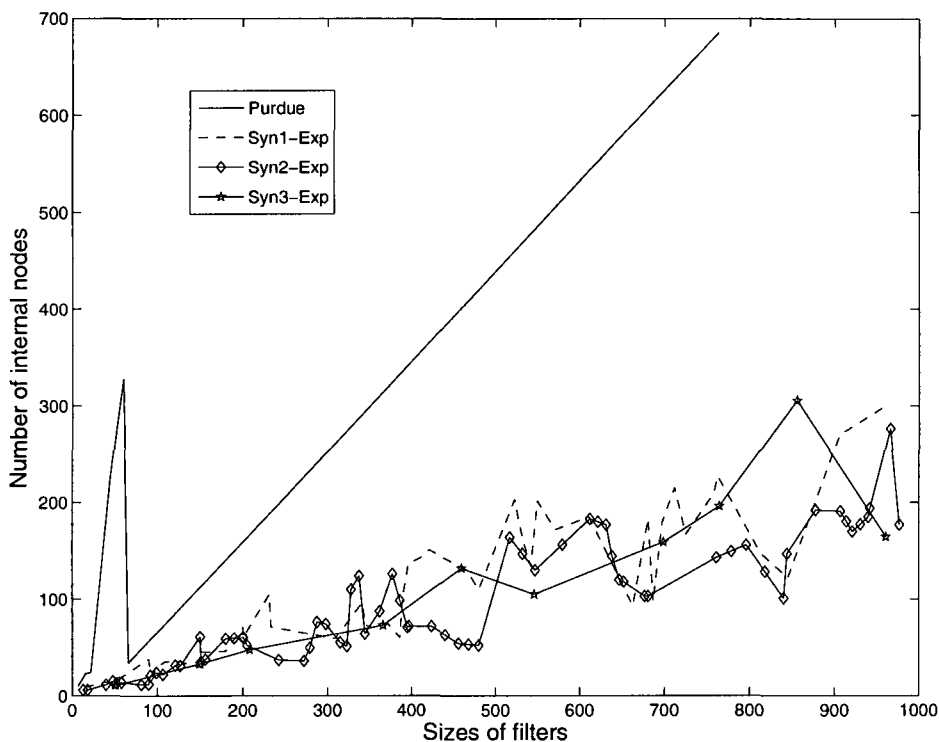


Figure 4.5 : Filter size V.S. number of internal nodes.

height of the tree big, however the average depth of all leaf nodes may be much smaller than the height. Figure 4.9 shows the correlation between the heights of trees and the average depths of trees.

Although we have shown that in the common case some factors are highly correlated with each other, it is worth noting that they are not identical factors, i.e., they can not be replaced by each other. Therefore, we still need to use all factors to more accurately characterize each packet filter. Another difference between the Class-1 factors and the Class-2 factors is that the Class-1 factors are much cheaper to calculate than the Class-2 factors.

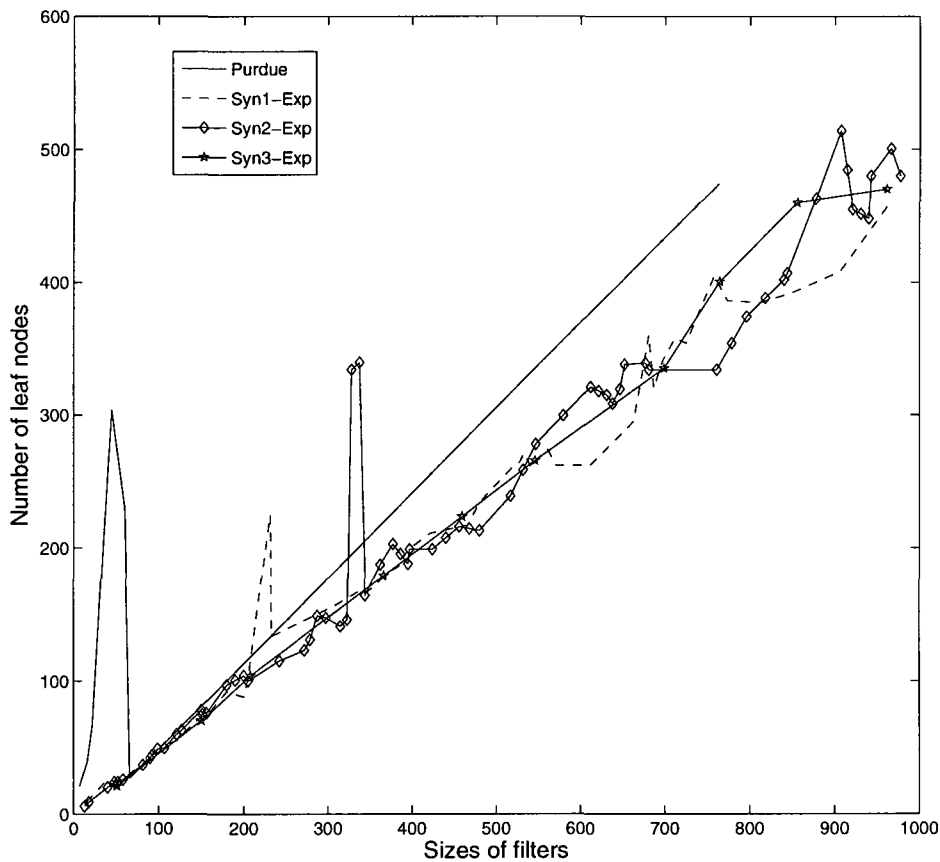


Figure 4.6 : Filter size V.S. number of leaf nodes.

4.3.1.3 Factors Relevant to the Goodness of Packet Filter Pairs

We have identified a set of factors that can characterize each individual filter. Now we continue to show that these factors are also important for determining whether a set of packet filters should share a tree or not. To make this problem simpler, in the following section, we will first study why these factors are important to decide whether any pair packet filters should share a tree or not.

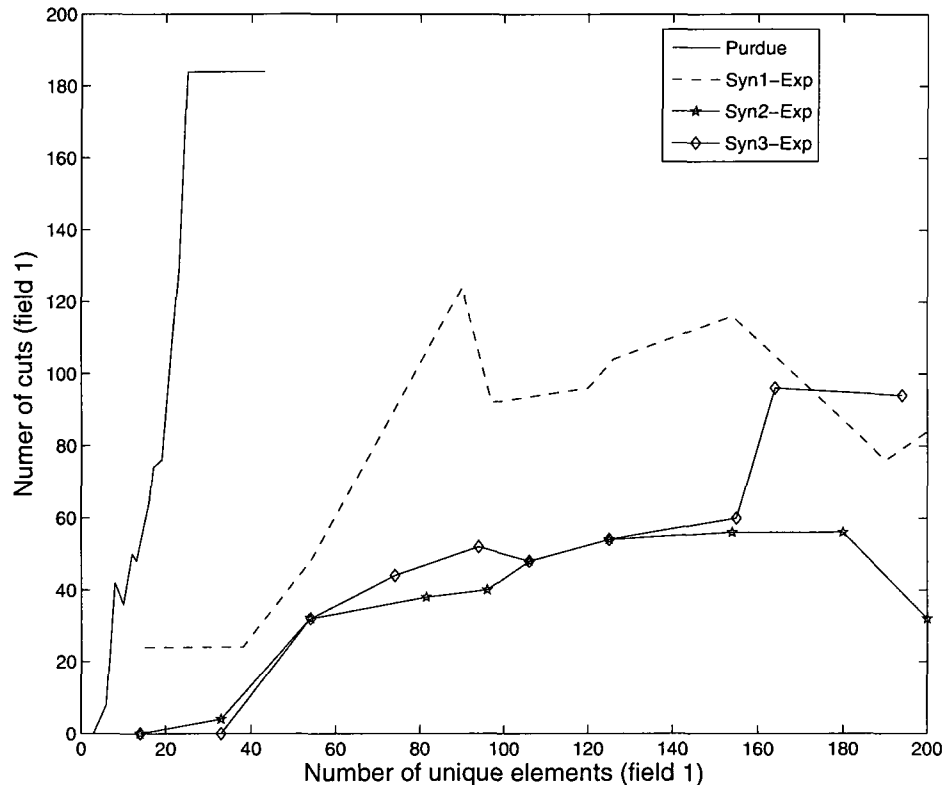


Figure 4.7 : Number of unique elements V.S. number of cuts.

Let us first define a pair of packet filters as a *good pair* if and only if the shared tree of the two packet filters consumes less memory than the two separate trees and the height of the shared tree is no bigger than the average height of the two separate trees. Since both the memory consumption and the height of the tree are important performance metrics, a factor should be considered to be an important one as long as it can affect at least one performance metric of the shared tree.

Size difference: Assume that the size difference of two packet filters is big, i.e., one packet filter F_1 is much smaller than the other packet filter F_2 . Let us denote the size of F_1 and F_2 as s_1 and s_2 , the decision tree memory consumption of F_1 , F_2 and the shared tree as

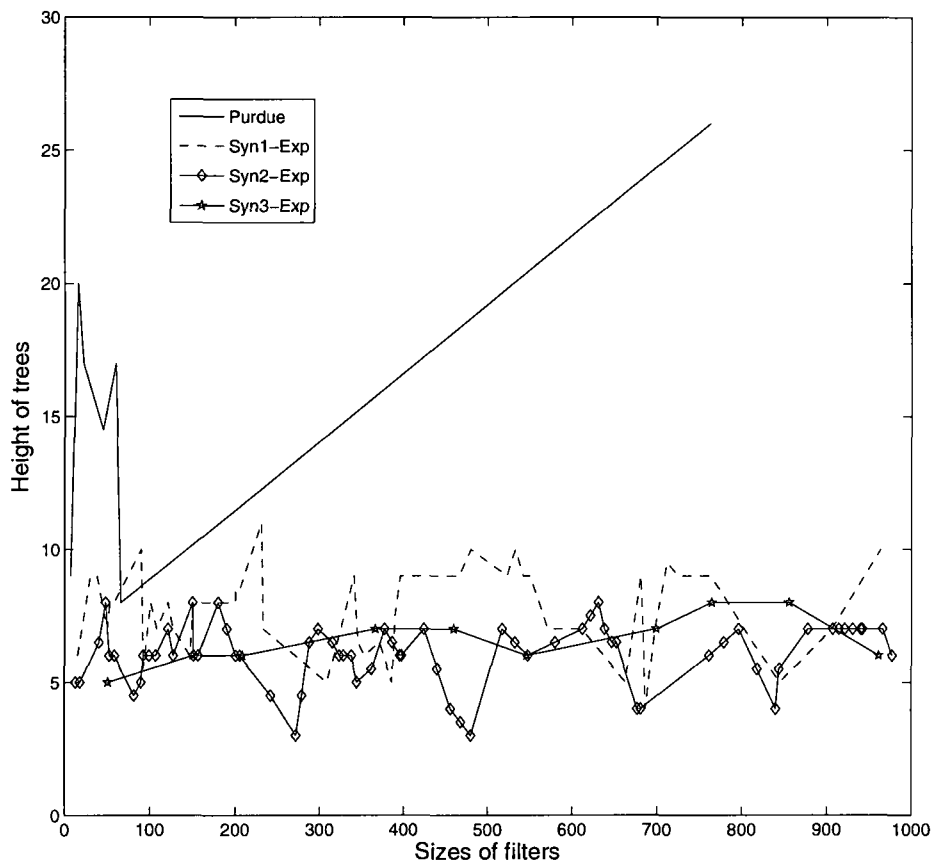


Figure 4.8 : Filter size V.S. height of trees.

m_1 , m_2 and m_s , the height of F_1 , F_2 and the shared tree as h_1 , h_2 and h_s . Recall that the average number of rules in internal nodes of the shared tree and the average number of unique elements on each field in internal nodes on the shared tree are determined by both packet filters. In addition, the number of children pointers an internal node can have is bounded by the average number of rules in the node. Therefore, the number of children pointers in the root node of the shared tree is generally smaller than the number of children pointers in the root node of the decision tree of F_2 . Consequently, it will take more times of

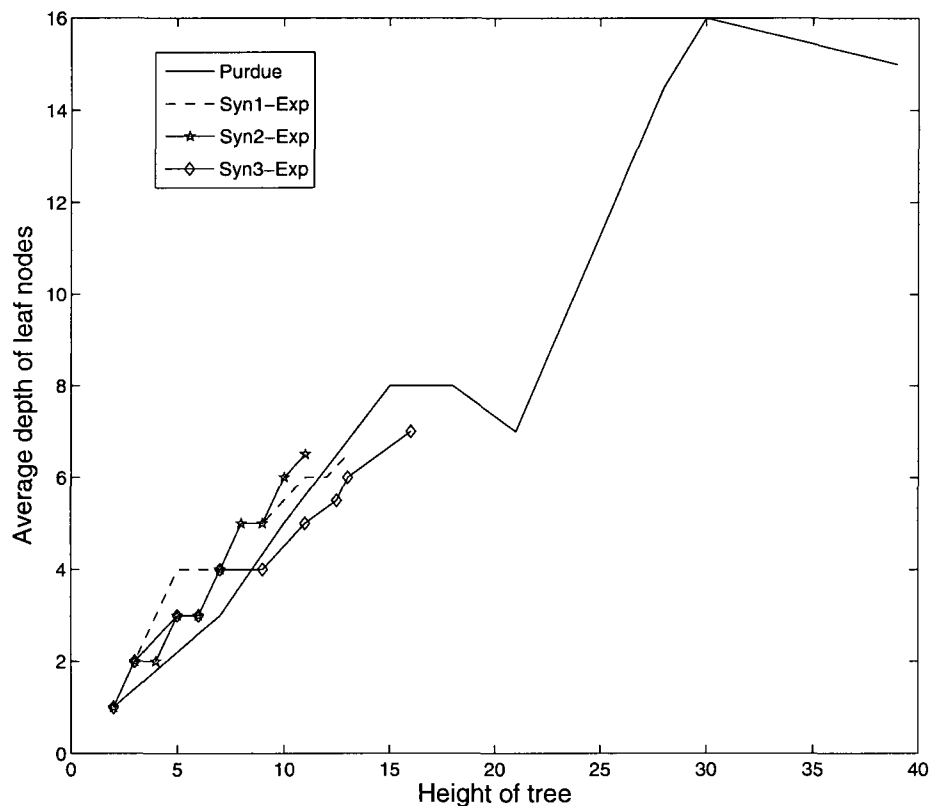


Figure 4.9 : Height of trees V.S. average depth of leaf nodes.

splitting to reduce the number of rules in nodes down to *BucketSize*. Thus, h_s should be larger than h_2 . h_1 should be similar to h_2 according to Figure 4.8. Therefore, after letting F_1 and F_2 shared a decision tree, the height of the shared tree will generally be bigger than $(h_1 + h_2)/2$.

As for the memory consumption, F_2 should have a larger number of unique elements on each field, so which fields to cut on each internal node will be mainly determined by F_2 . Therefore, the structure of the shared tree will be more similar to the decision tree of F_2 . Since the shared tree will be taller than decision tree of F_2 , more internal nodes and leaf

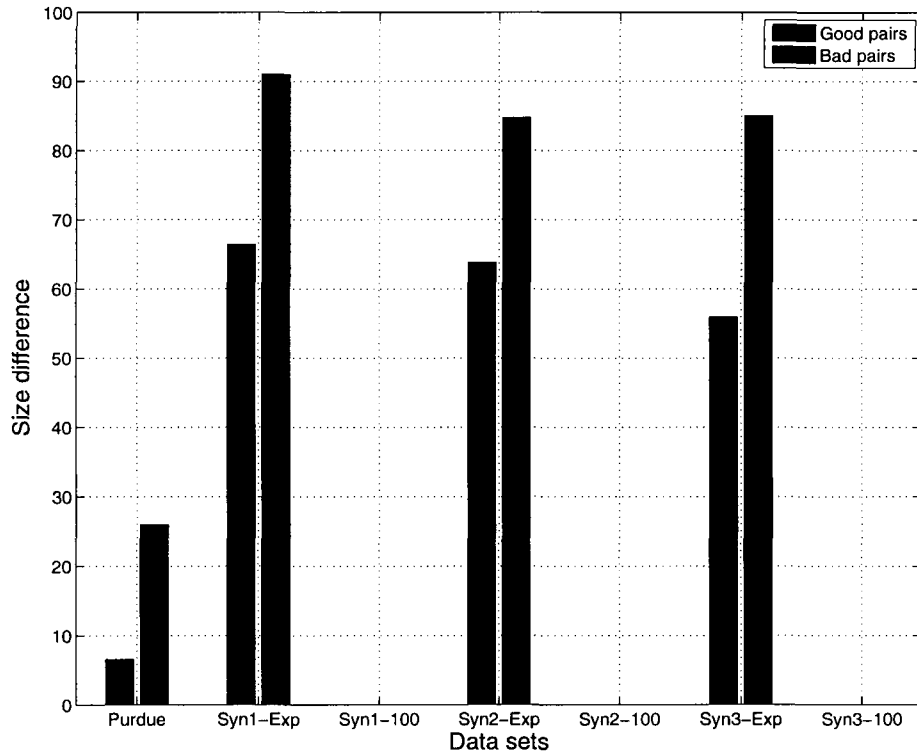


Figure 4.10 : Size differences: good pairs V.S. bad pairs.

nodes will be needed generally. On the other hand, the internal nodes from the decision tree of F_1 can be saved, so it is hard to say whether m_s will be smaller or bigger than $m_1 + m_2$.

Figure 4.10 compares the size difference of good pairs and bad pairs for all data sets. As you can see, good pairs tend to have smaller size difference as expected.

Number of internal/leaf nodes difference: Since there is a high correlation between the number of internal/leaf nodes and the packet filter sizes, if two packet filters differ greatly on the number of internal or leaf nodes, then their size difference should be big as well in general. Therefore, a big difference on the number of internal or leaf nodes usually implies that these two packet filters should not be sharing a HyperCuts decision tree. Figure 4.11

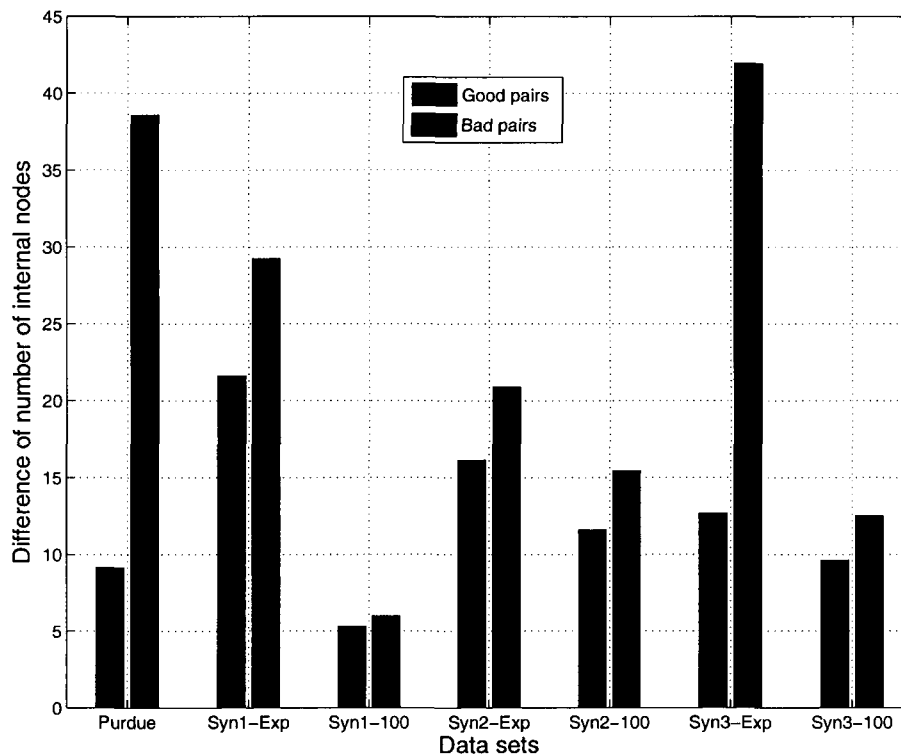


Figure 4.11 : Difference of number of internal nodes: good pairs V.S. bad pairs.

and Figure 4.12 compare the difference of the number of internal nodes and the difference of number of the leaf nodes between good pairs and bad pairs respectively. As expected, good pairs tend to have smaller differences.

Memory consumption difference: Since there is a strong correlation between memory consumption and packet filter sizes, if two packet filters differ greatly on memory consumption, then their size difference should be also big in general. Therefore, big memory consumption difference usually implies that these two packet filters should not share a HyperCuts decision tree in general. Figure 4.13 compares the memory difference of good pairs and bad pairs. As can be seen, good pairs tend to have closer memory consumptions.

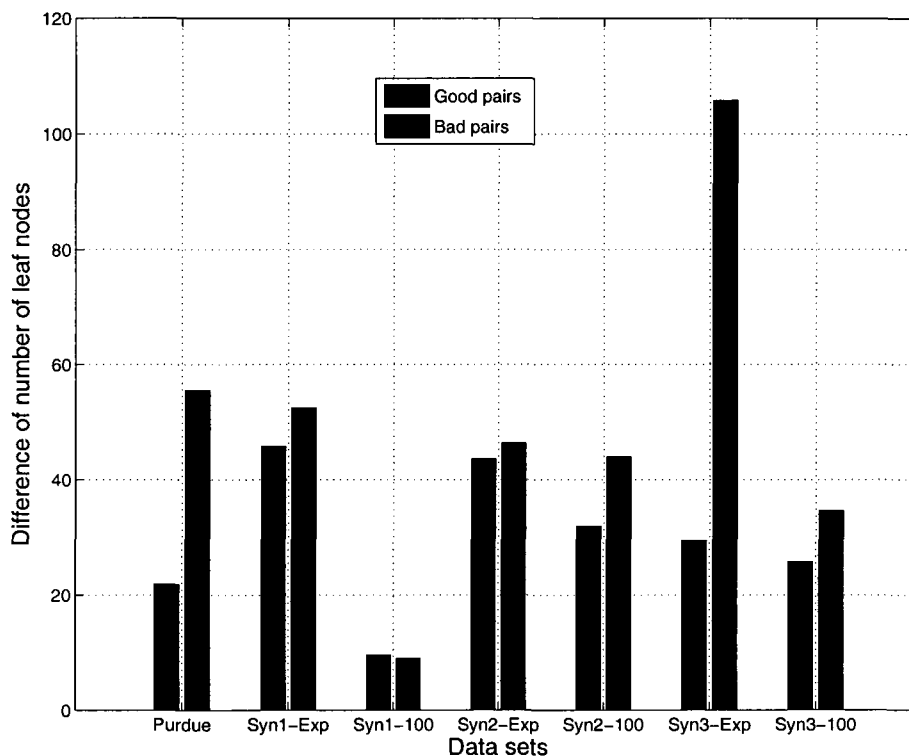


Figure 4.12 : Difference of number of leaf nodes: good pairs V.S. bad pairs.

Correlation of the vectors of the number of unique elements on each field: For each packet filter, we can calculate the number of unique elements on each field. The number of unique elements from all fields can form a vector. Given two packet filters F_1 and F_2 , we can calculate one vector for each filter, then we can calculate the linear correlation between the two vectors. If the correlation is high, then it means that they share the same subset of important fields (i.e., the fields with more unique elements). Therefore, the tree structures of the two packet filters should be similar. Consequently, it is easier to construct a shared tree for the two packet filters with good performance.

On the other hand, if the correlation between the two vectors is low, then it generally

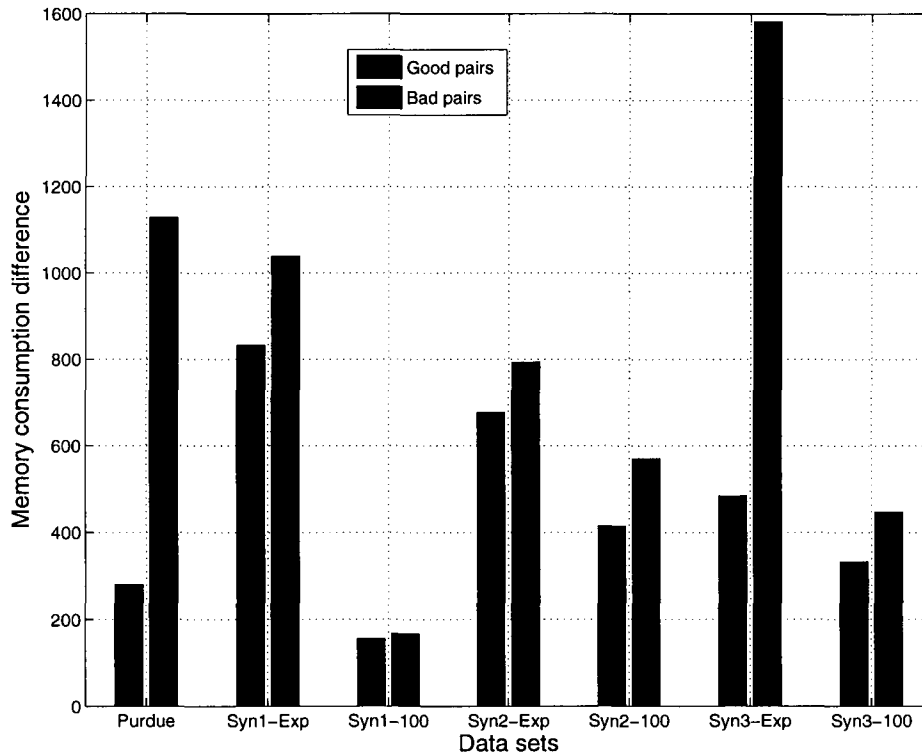


Figure 4.13 : Difference of memory consumption: good pairs V.S. bad pairs.

means that the two packet filters have different subsets of important fields. When the tree construction algorithm has to decide which fields to cut for an internal node, rules from both packet filters will be considered. Consequently, the structure of the shared decision tree will not be similar to either of the separate trees. This will generally lead to increased memory consumption and increased tree height. Figure 4.14 compares the average correlation rate of vectors from good pairs and bad pairs.

Correlation of vectors of the number of cuts on each field: Since there is a high correlation between the number of unique elements and the number of cuts on each field, if two packet filters have a low correlation for the vectors of the number of cuts on each field,

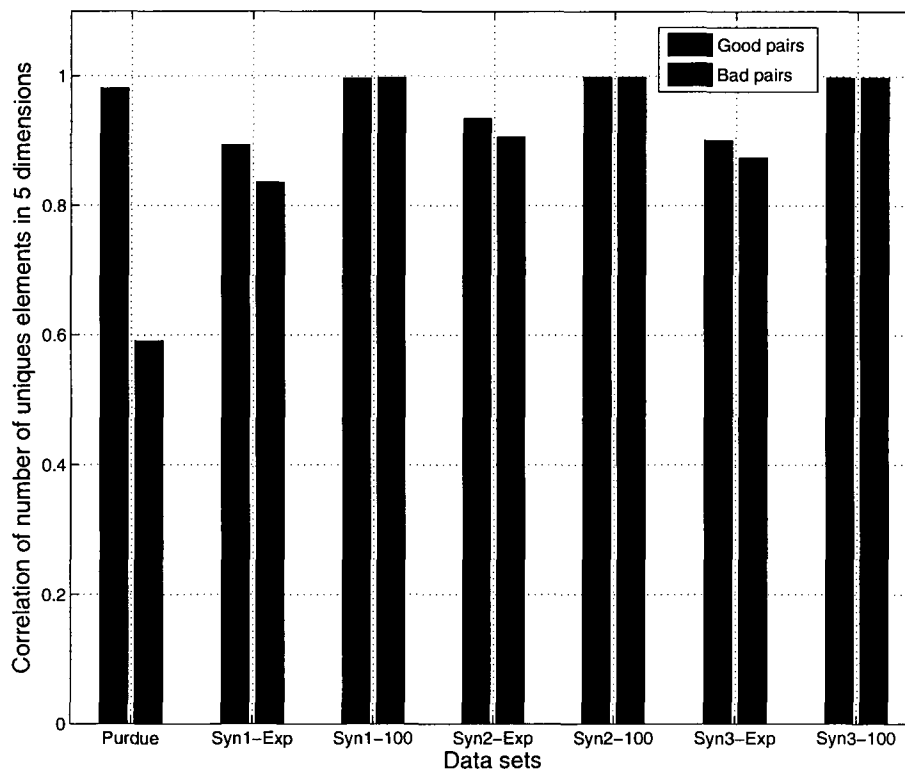


Figure 4.14 : Correlation of number of unique elements in all dimensions: good pairs V.S. bad pairs.

then their correlation for the vectors of the number of unique elements on each field will also be low generally. Therefore, a low correlation of number of cuts usually implies that these two packet filters should not be merged together in general. Figure 4.15 compares the average correlation rate of vectors from good pairs and bad pairs.

Height difference: Given two packet filters F_1 and F_2 , let us denote the heights of F_1 , F_2 and the shared tree as h_1 , h_2 and h_s respectively. Assume that h_1 is much larger than h_2 . h_s will be similar to h_1 . Thus, after making the two packet filters share a HyperCuts tree, the height of the shared tree will generally become larger than the average height of the

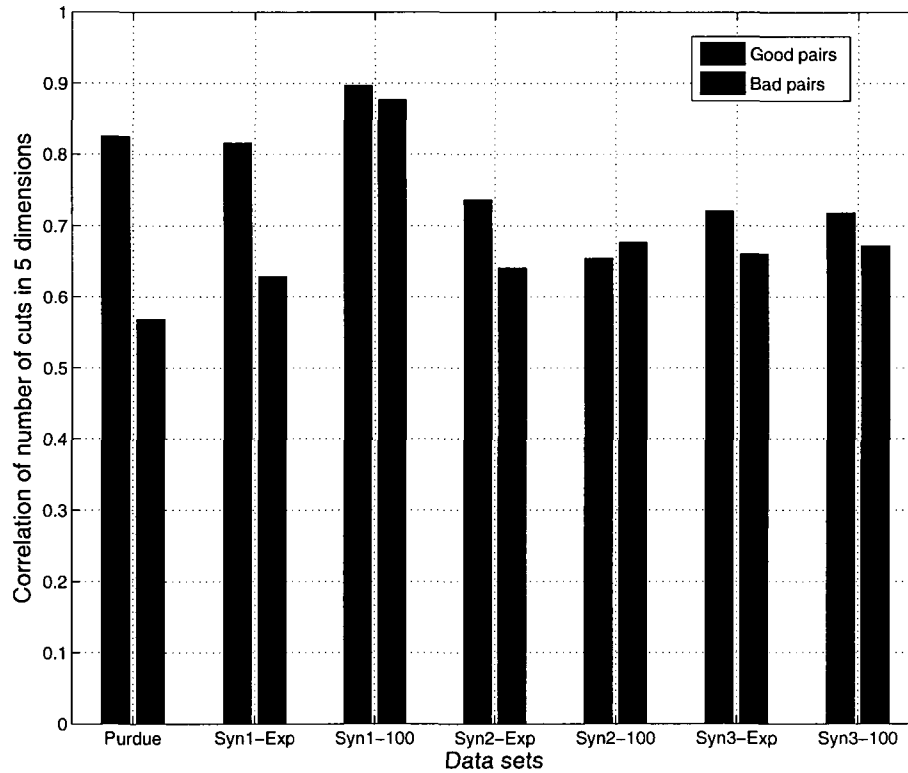


Figure 4.15 : Correlation of number of cuts on all dimensions: good pairs V.S. bad pairs.

two separate trees. Therefore, if the height difference of two packet filters is big, then they should not share a tree in general. Figure 4.16 compares the height difference of good pairs and bad pairs for all data sets.

Difference of average depth of leaf nodes: Since there is a correlation between the height of the tree and the average depth of leaf nodes, if the average depths of leaf nodes of the two packet filters are very different, then their height difference will also be large generally. Therefore, a big difference on average depth of leaf nodes generally implies that the two packet filters should not be sharing a decision tree. Figure 4.17 compares the average leaf node depth difference of good pairs and bad pairs for all data sets.

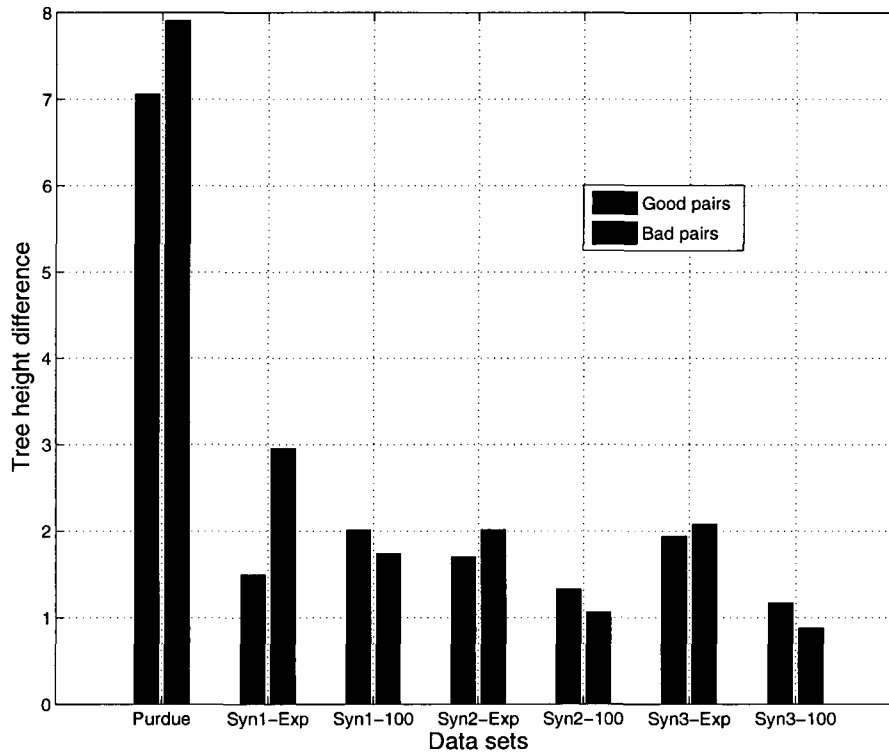


Figure 4.16 : Height difference: good pairs V.S. bad pairs.

4.3.2 Predicting Good Pairs of Packet Filters

Two packet filters are defined to be a “good” pair if their shared HyperCuts tree has decreased memory usage and decreased average depth of leaf nodes compared to the two separate HyperCuts trees. This problem is clearly a classification problem, i.e., we need to classify all pairs of packet filters into either good pairs or bad pairs. However, it is non-trivial to manually derive some effective rules for us to accurately decide whether a pair of packet filters should share a tree or not. Luckily, some effective supervised machine learning techniques [Mit97] can help perform this classification task. We will study a few representative supervised machine learning techniques in Section 4.4.

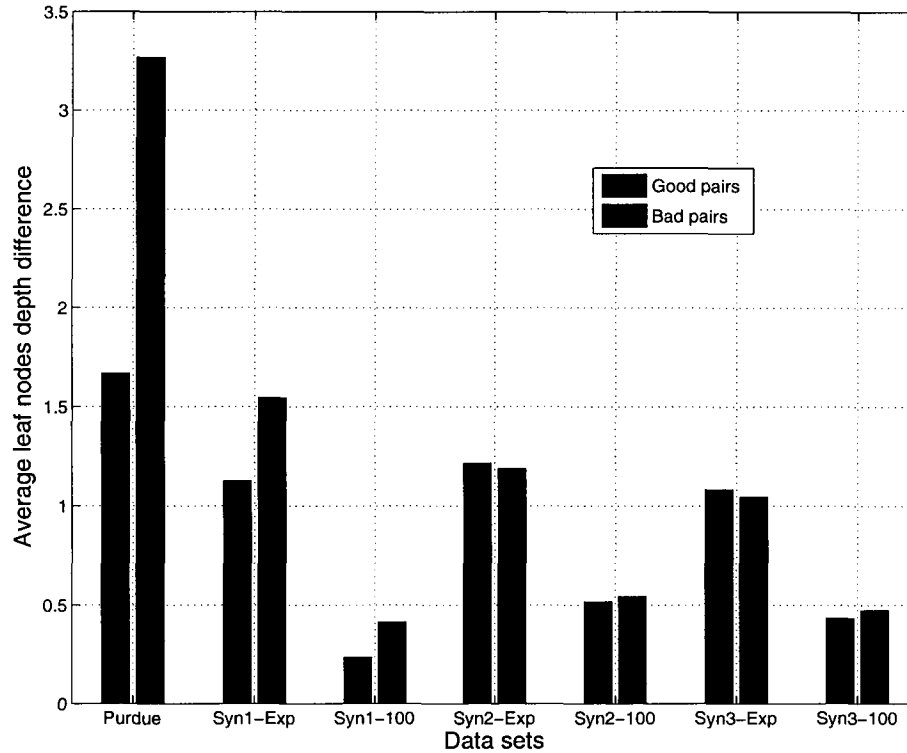


Figure 4.17 : Average leaf nodes depth difference: good pairs V.S. bad pairs.

To use machine learning techniques to predict whether a pair of filters is good, we need to first prepare some training data to train a model. Given a filter data set with N distinct packet filters, we can randomly select M pairs of filters out of all possible $N \times (N - 1)/2$ pairs as the training data. For each selected pair of filters, we can decide whether they are a good pair by constructing two separate trees and one shared tree. For each selected pair of filters, we can also calculate their factors. By feeding all these information to certain machine learning technique, a model can be learned to predict whether any new pair of packet filter is good or bad. We will evaluate the prediction accuracy of different machine learning techniques in Section 4.4.

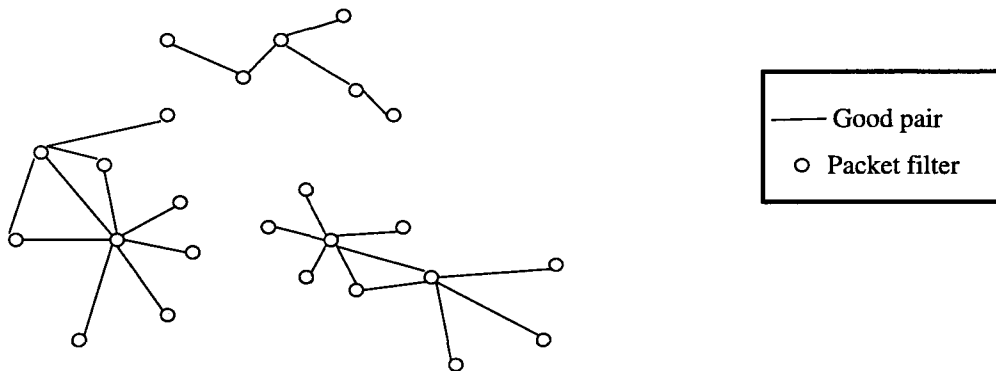


Figure 4.18 : Constructing a graph based on pair-wise prediction.

4.3.3 Clustering Packet Filters Based on Pair-wise Prediction

By using the model learned from a small amount of training pairs, we can now predict whether any pair of filters is good or not. Based on the pair-wise prediction for all possible pairs of all filters, an undirected graph G can be constructed as illustrated in Figure 4.18. In the graph G , each node represents a distinct packet filter. Two nodes in G are connected with an edge if and only if the two corresponding packet filters are predicted to be a good pair. Given the constructed graph G , the following clustering algorithm is proposed to determine which subset of packet filters should share a HyperCuts decision tree:

INPUT OF ALGORITHM: G and $\alpha \in [0, 1]$

OUTPUT OF ALGORITHM: A set of packet filter clusters: $S_{clusters}$

- 01: $S_{filters} = \{\text{All packet filters}\};$
- 02: $S_{clusters} = \{ \};$
- 03: WHILE ($|S_{filters}| > 0$)
- 04: $cluster_i = \{ \};$
- 05: Find $f_m \in S_{filters}$ who has most neighbors from $S_{filters}$ in G ;
- 06: $cluster_i = cluster_i \cup \{f_m\};$
- 07: $S_{filters} = S_{filters} \setminus \{f_m\};$
- 08: WHILE TRUE
- 09: Find $f_n \in S_{filters}$ with most neighbors from $cluster_i$ in G

if multiple choices exist, pick the one with largest degree in G ,
 let us assume f_n has k neighbors from $cluster_i$ in G ,

```

10: IF ( $k \geq \alpha \times |cluster_i|$ )
11:    $cluster_i = cluster_i \cup \{f_n\}$ ;
12:    $S_{filters} = S_{filters} \setminus \{f_n\}$ ;
13: ELSE break the WHILE loop;
14: END-IF
15: END-WHILE
16:  $S_{clusters} = S_{clusters} \cup \{cluster_i\}$ ;
17: END-WHILE
18: RETURN  $S_{clusters}$ ;

```

In the above algorithm, α is a constant value between 0 and 1. Intuitively, the higher the α value is, the more difficult that a packet filter can join an existing cluster. For example, if α is set to 0, then all packet filters in the same connected component in G will share a HyperCuts decision tree. On the other hand, if α is set to 1, then a set of packet filters will be clustered together if and only if the corresponding nodes in G form a clique. We will evaluate the performance of the clustering algorithm with different α values in Section 4.4.2.

4.4 Performance Evaluation

In Section 4.4.1, we first evaluate how accurately we can predict whether a pair of packet filters should share a tree. We then study the performance of the packet filter clustering algorithm in Section 4.4.2. Finally, we show the detailed breakdown of the time spent on each step of our approach in Section 4.4.3.

4.4.1 Accuracy of Predicting Good Pairs

As introduced in Section 4.3, we want to apply supervised machine learning techniques to address this classification problem. A supervised machine learning technique can automat-

ically learn a model from some training data. The training data consists of pairs of input vectors, and desired outputs. After a model is learned, it can then be used to predict an output value for any valid input vectors. We discuss how we define the input vectors, the output values and three classification techniques we studied in detail as follows.

4.4.1.1 Three Types of Input Vectors

Based on the two classes of factors introduced in Section 4.3.1, we can define three types of input vectors for each pair of packet filters. The first type of input vectors is composed of only the Class-1 factors from both filters. The second type of input vectors is composed of only the Class-2 factors of both filters. The third type of input vectors includes both the Class-1 and Class-2 factors of the two filters. We evaluate the impact of using different types of input vectors in Section 4.4.1.4.

4.4.1.2 Defining Output Values

The output of our classification problem should be a label indicating whether the input vectors correspond to a good pair or not. That is, there are only two possible output values: good or bad. In this section, we define two packet filters as a good pair if their shared HyperCuts tree's memory consumption ratio and average leaf depth ratio are both smaller than 1. That is, the shared HyperCuts tree must have decreased memory consumption and decreased average depth of leaf nodes compared against two separate HyperCuts trees. Please note that in the above definition, if we replace the average depth of leaf nodes with the height of the tree, the prediction accuracy is a little worse according to our study. The reason is that the heights of trees are determined by the leaf node with largest depth, so it is not as stable as the average depth of all the leaf nodes.

We studied the percentage of good pairs by examining 10,000 randomly selected pairs from each data set. The fractions of good pairs vary from 8% to 16% across all 7 data sets. Since the fractions of good pairs are relatively small, any classification technique that can accurately identify good pairs will be very useful in practice.

4.4.1.3 Three Classification Techniques

We studied three representative classification techniques including the decision tree (DT)¹, the generalized linear regression (GLR) [Mit97] and the naive Bayse classifier (NBC) [Mit97]. We plan to study more classification techniques such as the neural network in the future.

It is straightforward to apply the DT technique to perform classification here. For GLR technique, if we use the output values 1 and 0 to represent the good pair and the bad pair respectively in the training data, then given a new pair of filters, GLR will output a value between 0 and 1. In our experiment, if the returned value by GLR is larger than 0.5 then we predict the pair as good. Otherwise, the pair is predicted to be bad. As for NBC, we cannot directly feed the input vectors defined in Section 4.4.1.1 to NBC technique. NBC requires a set of features instead. In our experiment, we simply define a corresponding feature from each factor. For example, the size of the first packet filter in the pair is a factor. We can define its corresponding feature as follows: we first calculate the 10th percentile and 90th percentile of the sizes of the first packet filter from all good pairs in the training data. A pair of testing packet filters is then said to have this feature if the size of its first packet filter falls into the above 10th and 90th percentile range. After we convert factors into features, the NBC can be used directly to perform classification.

4.4.1.4 Accuracy of Pair-Wise Prediction

For each data set, we randomly select 10,000 pairs and then calculate both Class-1 and Class-2 factors for those selected pairs. We also need to determine whether each selected pair is good or bad. To evaluate the prediction accuracy using different types of input vectors, we randomly choose 1,000 pairs (i.e., 10%) out of the 10,000 pairs as the training data. We then use the rest 9,000 pairs as the testing data to test the prediction accuracy

¹To avoid ambiguity, we always use “HyperCuts decision tree” to refer the packet classification technique, while using “decision tree” or “DT” in this section to represent the machine learning techniques used

of the learned model. We repeat this experiment 10 times, each of which uses a different 1,000 pairs as the training data. Figure 4.20, 4.21 and 4.22 show the average false positive rate and the average false negative rate of the three classification techniques using different input vectors across 10 runs.

First of all, different types of input vectors have a significant impact on the false positive and false negative rate for all three techniques. Only using Class-1 factors as input gives the worst prediction accuracy for both DT and GLR. Including Class-2 factors in the input vectors help improve the performance of both DT and GLR. This is expected because Class-1 factors are relatively simple and they are not sufficient to predict the final HyperCuts decision tree. However, including more factors as input does not help NBC. Instead, when more and more factors are included as input, the performance of NBC is getting worse. The NBC technique assumes that all the input variables are independent to each other, while in our case, those input factors may not be completely independent. When having more and more dependent variables into the input vectors, the performance may get worse.

Secondly, among the three techniques we have studied, DT technique has the best overall performance. GLR does not work well because its linear model simply can not accurately capture the complex relationships among those factors. NBC falls short because it assumes that all factors are independent while they are actually not. If both Class-1 and Class-2 factors are used in the input vectors to train the decision tree, then the false positive rates will vary from 3% to 8%. In addition, the average false negative rate across the 7 data sets is 23%. A low false positive rate is important because it means that only a small percentage of bad pairs will be misclassified to be good ones. A 23% false negative rate means that 23% of all good pairs will be misclassified as bad pairs. Fortunately, the high false negative rate can be alleviated by the filter classification algorithm discussed in Section 4.3.3. Please recall that each misclassified good pair represents a missing edge in the graph as illustrated in Figure 4.19. Our study shows that the two packet filters on a missing edge are 1.7 hops away from each other on average, so it is still very likely that they will be classified into the same cluster by our classification algorithm. On the other hand, for those

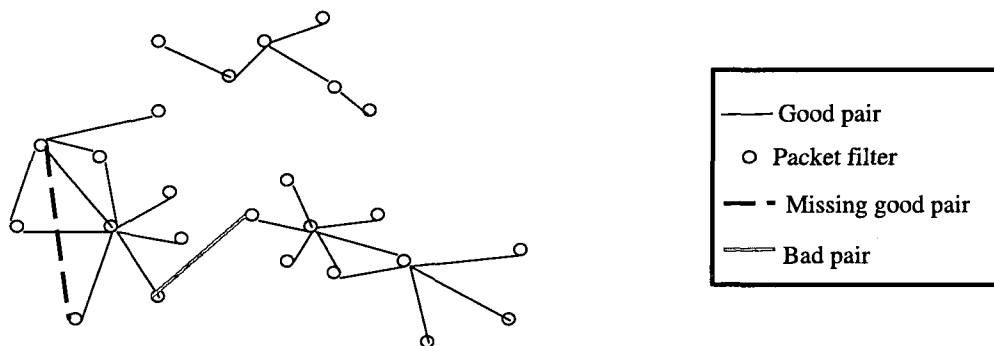


Figure 4.19 : The filter classification algorithm helps alleviate the high false negative rate of the pair-wise prediction.

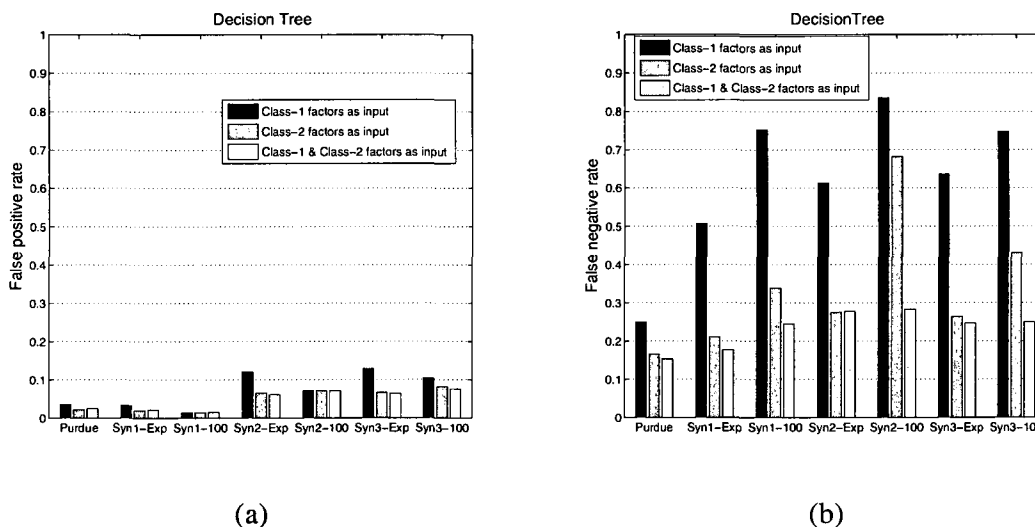


Figure 4.20 : Performance of decision tree technique: (a) false positive rate (b) false negative rate.

truly bad pairs, 70% of them are not even connected on the graph. The rest of the 30% of bad pairs are 3 hops away from each other on average, which makes it much harder for the classification algorithm to classify them into the same cluster.

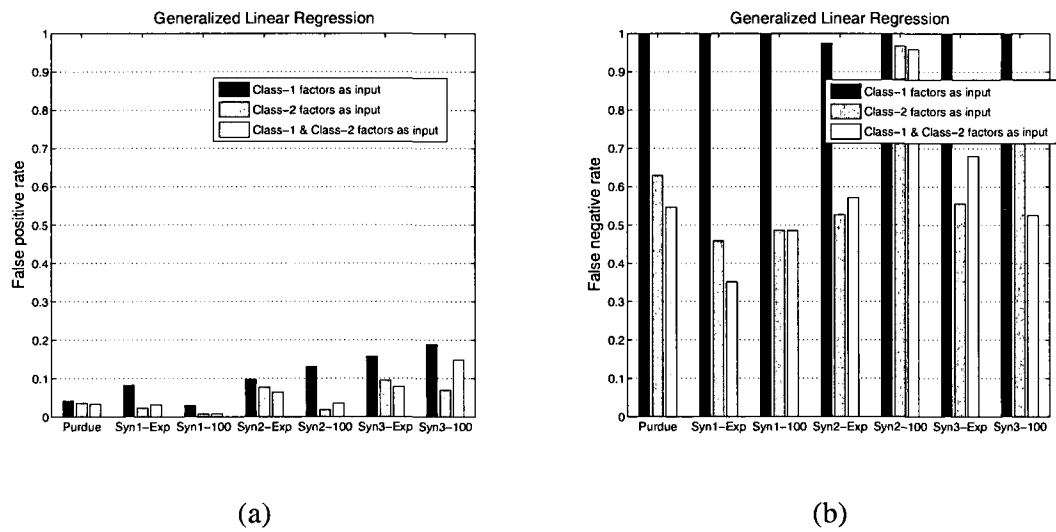


Figure 4.21 : Performance of generalized linear regression: (a) false positive rate (b) false negative rate.

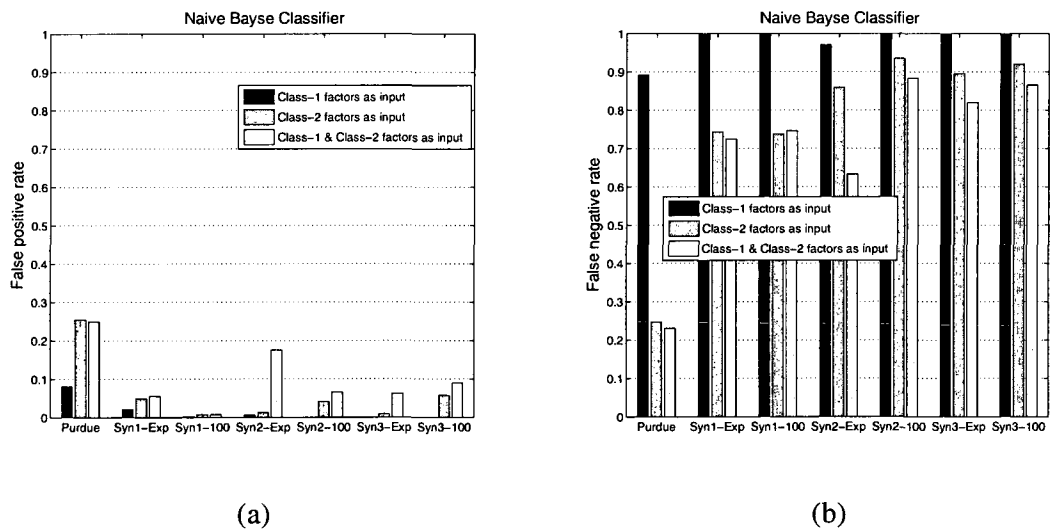


Figure 4.22 : Performance of naive Bayes classifier: (a) false positive rate (b) false negative rate.

4.4.2 Performance of The Filter Clustering Algorithm

Since we have shown that the DT technique using both Class-1 and Class-2 factors as input has the best prediction accuracy among the three techniques we studied, in this section we

will use DT to predict the goodness of all pairs of packet filters in a data set. Based on the pair-wise prediction provided by DT, we can construct a graph G for each filter data set. We can then apply our filter clustering algorithm to cluster nodes in G to decide which subset of packet filters should share a HyperCuts decision tree. DT is trained by using a training data set of 1,000 random pairs, and the results presented in this section are the average values across 10 runs. Recall that in addition to G , the proposed clustering heuristic algorithm also needs a constant α . In our experiment, we vary α from 0.25 to 1.

Figure 4.23 shows the performance of the final constructed shared trees for 140 Purdue filters. When $\alpha = 0.25$, the shared trees actually have much worse performance than the 140 separate trees. Please recall that a smaller α value means that a packet filter can more easily join an existing cluster. When a packet filter is mistakenly classified into a wrong filter cluster, the overall performance of the cluster will significantly degrade. When a larger α such as 0.5 is used, the performance becomes better. The overall memory saving is over 40%. In the meantime, the average height of the shared trees and the average depth of leaf nodes in shared trees slightly decrease.

Figure 4.25, Figure 4.26 and Figure 4.27 show the overall performance of the 6 synthetic data sets. As can be observed, when α increases, the memory consumption ratio generally increases while the average leaf depth ratio and tree height ratio decrease. If we fix α as 1, then we can reduce memory consumption over 20% on average while only increasing average leaf depth by 3% on average across all 6 synthetic data sets.

In our algorithm, the parameter α plays a vital role in determining the filter clustering results and also the performance of the constructed shared HyperCuts decision trees. However, determining the optimal α value for a specific packet filter data set is beyond the scope of this thesis. We will continue to study this problem as our future work.

In addition, the shared HyperCuts trees enable concurrent lookup of multiple packet filters sharing the same tree. Therefore, if multiple packet filters lie on a path inside a router group, then by using the shared HyperCuts trees, the total tree heights along the path may be reduced. Recall that the tree height represent the worse case tree traversal time. We

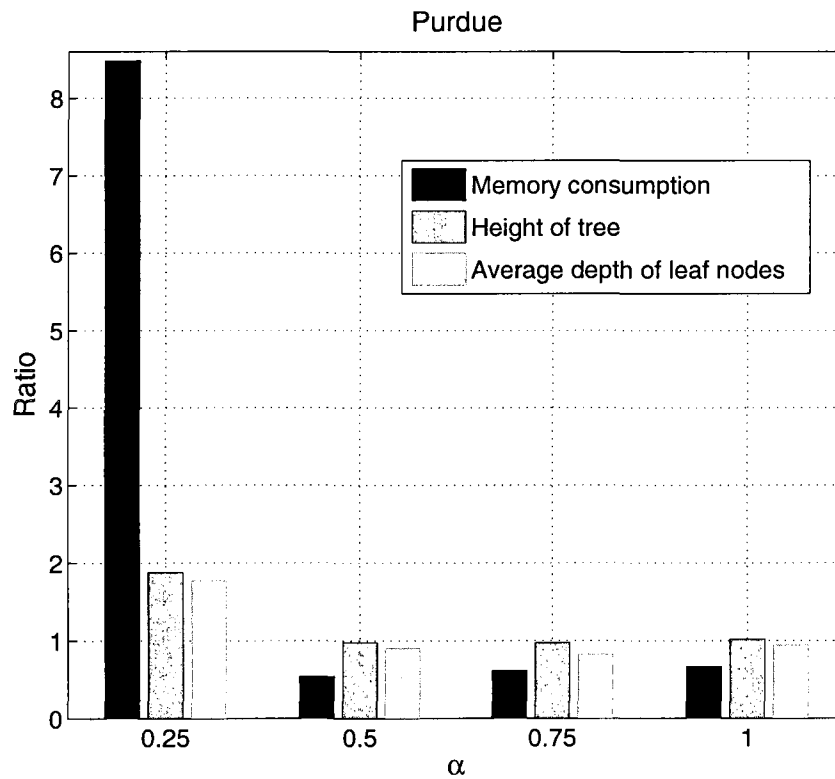


Figure 4.23 : Shared HyperCuts trees V.S. separate HyperCuts trees for Purdue data.

use the Purdue network in our experiment. For each random path in the Purdue network, we first determine how many filters are lying on the path. Then we compare the total heights of the separate HyperCuts trees and the shared HyperCuts trees. Figure 4.24 shows the result. As you can see from Figure 4.24, the shared HyperCuts trees can help reduce the worst case tree traversal time if multiple packet filters lie on the same path. When the number of packet filters lying on a path increases, the benefits of using the proposed shared HyperCuts tree also increase.

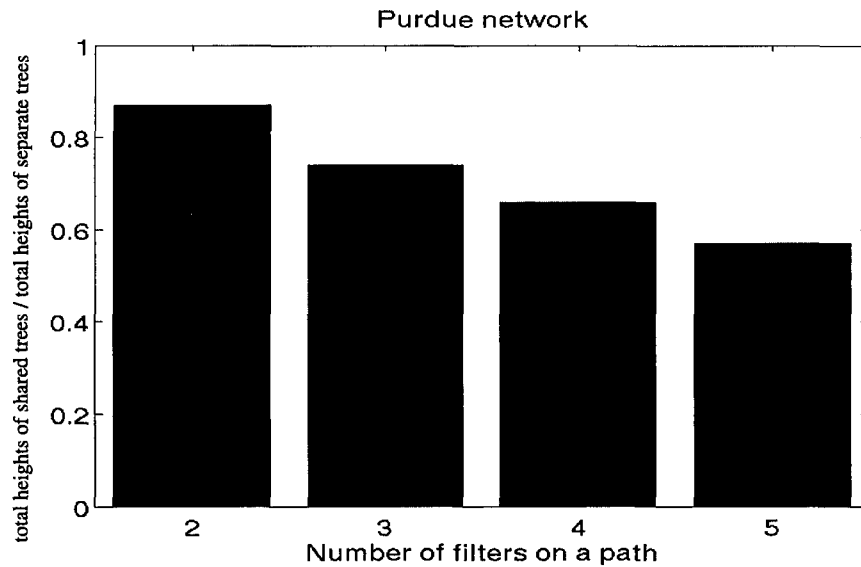


Figure 4.24 : The shared HyperCuts trees enable concurrent lookup of multiple packet filters sharing the same tree.

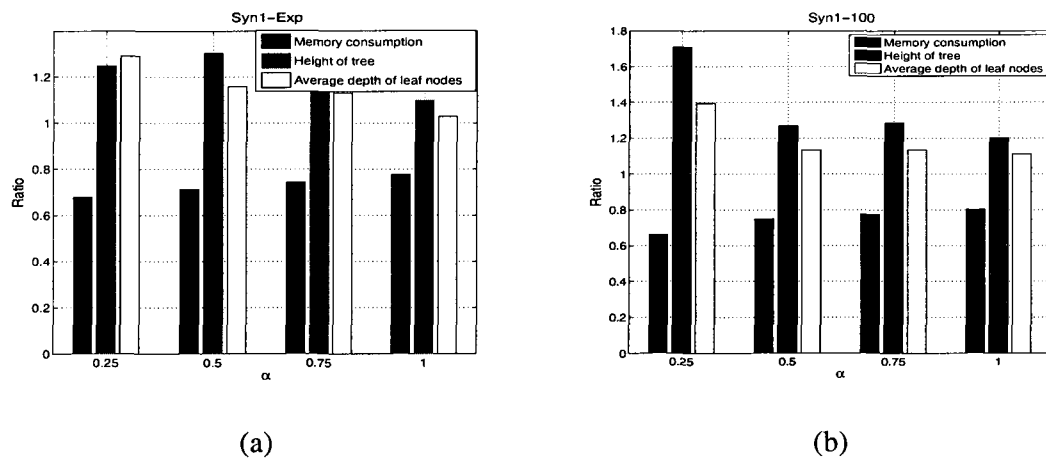


Figure 4.25 : Shared HyperCuts trees V.S. separate HyperCuts trees: (a) Syn1-Exp (b) Syn1-100.

4.4.3 Computation Time Breakdown

In this section, we want to study the computation time spent on each step in our approach. We break our approach into 7 steps: (1) calculating Class-1 factors, (2) calculating Class-2

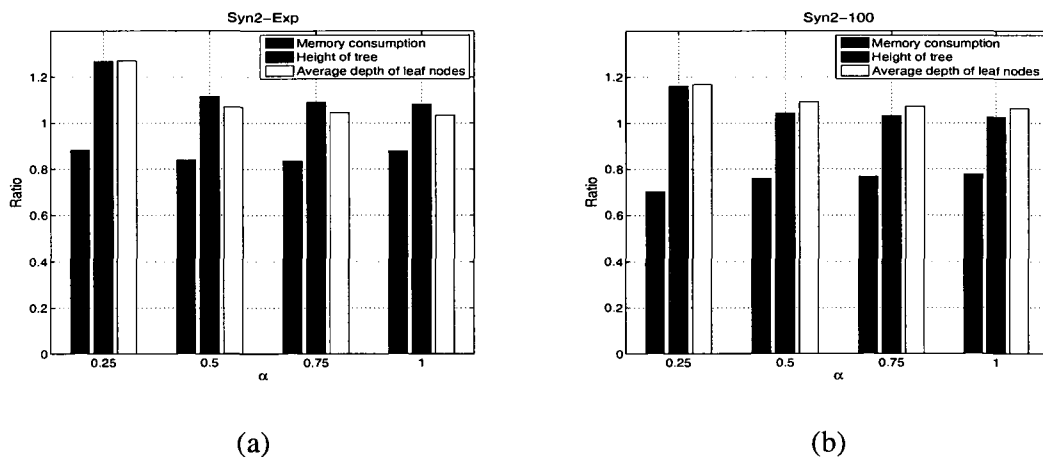


Figure 4.26 : Shared HyperCuts trees V.S. separate HyperCuts trees: (a) Syn2-Exp (b) Syn2-100.

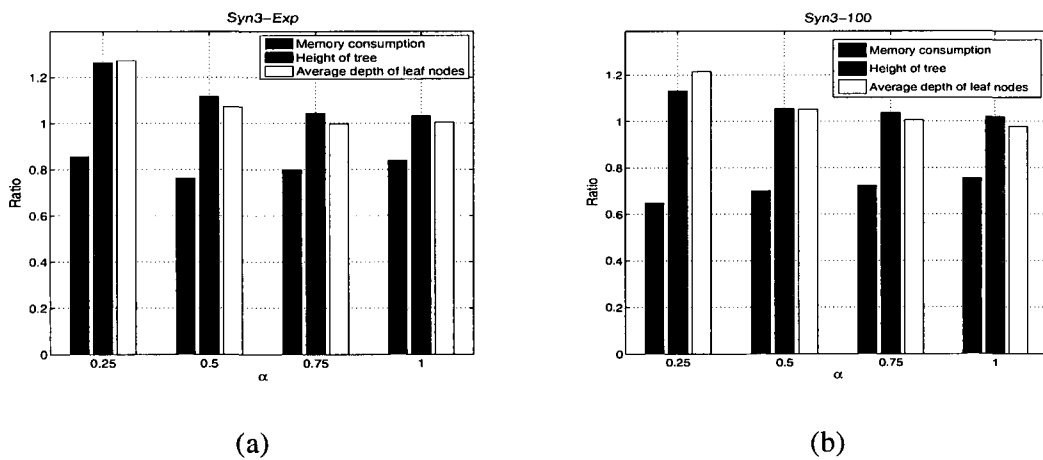


Figure 4.27 : Shared HyperCuts trees V.S. separate HyperCuts trees: (a) Syn3-Exp (b) Syn3-100.

factors, (3) generating 1,000 training pairs, (4) training the DT, (5) predicting the goodness of all pairs to construct G , (6) clustering packet filters and (7) constructing the shared HyperCuts decision trees. As for implementation, steps (4)-(6) are implemented in Matlab and the other steps are implemented in the C++ language. The desktop machine used in our experiment has a 2.6 GHz AMD Opteron processor and 4 GB of main memory.

Data Sets	Class 1	Class 2	Generate training pairs	Train	Predict G	Cluster	Construct shared trees	Total
Purdue	1.7	4.9	6.9	0.29	0.01	0.52	2.6	16.92
Syn1-Exp	28.2	86.4	298.9	0.1	0.33	57.6	51.3	522.83
Syn1-100	24.7	69.3	138.6	0.3	0.7	62.0	21.1	316.7
Syn2-Exp	28.7	86.9	315.0	0.31	0.78	46.6	26.7	504.99
Syn2-100	23.5	74.5	143.9	0.15	0.33	7.7	48.6	298.68
Syn3-Exp	23.9	72.3	312.0	0.51	0.88	36.9	65.8	512.29
Syn3-100	24.5	74.2	141.1	0.41	0.75	31.7	35.3	307.96

Table 4.3 : Computation time breakdown (in seconds) for each step in the proposed approach.

Table 4.3 shows the detailed breakdown of time (in seconds) spent on each step for all 7 data sets. When performing the packet filter clustering step, we set $\alpha = 0.5$. As can be seen, the step of preparing the training data takes the most time for all 7 data sets. The reason is that we need to construct 2,000 separate HyperCuts tree and 1,000 shared HyperCuts trees. The time spent in clustering packet filters and constructing shared HyperCuts trees is relatively modest. Therefore, a network operator may want to run the filter clustering and shared tree construction steps a few times with different α values to select one α offering best performance. In summary, it takes our approach about 17 seconds to construct shared HyperCuts trees for 140 real packet filters and about 6.8 minutes on average to construct a set of shared HyperCuts trees for 1,000 synthetic packet filters.

4.5 Another Application of the Shared HyperCuts Decision Tree

Packet filters are widely used on network devices such as routers to perform various services including firewalling, quality of service (QoS), virtual private networks (VPNs), load balancing, traffic engineering, etc. Therefore, multiple packet filters serving different purposes may be deployed on a single physical router. With the emergence of virtual routers as a promising technology to provide network services, more packet filters belonging to different virtual routers need to be stored on a single physical host router. So far, we have shown that by using a shared HyperCuts tree to represent multiple packet filters, mem-

ory consumption can be considerably reduced while not increasing the height of the trees. Consequently, more packet filters can be deployed and more virtual routers can be more efficiently supported on a single physical router.

4.5.1 The Need for Multiple Packet Filters on a Single Router

Multiple packet filters may be deployed on a single router to support different network services such as firewalling, QoS, VPNs, load balancing and traffic engineering. Due to the complexity of the network services, each packet filter may be large and complex as well. For example, recent studies have shown that a complex packet filter on modern routers or firewalls can have as many as 50,000 rules [ZWG07].

Today router virtualization is already available in commercial routers from both Cisco [cisa] and Juniper [junb]. It is quickly emerging as a promising technology to support new network services such as router consolidation [Junc], customer-specific routing, policy-based routing [Cise], multi-topology routing [P.] [T.] and network virtualization [APST05] [BFH⁺06]. For example, with the help of router virtualization, network operators can now consolidate a large number of existing routers onto a newly-purchased router by running one virtual router instance for each existing router. When performing router consolidation, all the packet filters deployed on existing routers will be exported to the new router. A Juniper router today can be configured with as many as 128 virtual routers. Therefore, a modern router may need to support a large number of packet filters.

4.5.2 Challenges of Deploying Multiple Packet Filters on a Single Router

One key challenge of holding a large number of packet filters on a single physical router is memory consumption. As more packet filters are deployed, the memory requirement will also increase accordingly.

Ternary content addressable memory (TCAM) is the de facto industry standard for hardware-based fast packet classification. However, TCAM has a few limitations. Firstly, TCAM consumes lots of power. Secondly, TCAM chips are expensive. They are often

more expensive than network processors [Lek03]. Thirdly, due to its high power consumption and high cost, the capacity of TCAM on each router is usually restricted by system designers. What is worse, in order to represent a packet filter in TCAM, the packet filter rules have to be converted to the ternary format, which will lead to the range expansion problem. For example, Cisco 12000, a high-end Gigabit switch router designed for large service providers and enterprise networks, can only hold up to 20,000 rules in its TCAM. Although some recently proposed TCAM-based packet classifiers compression techniques [MLT09] [Cha09] may help alleviate this problem, the amount of memory required to store a large number of packet filters can still easily exceed the capacity of the installed TCAM on a physical router.

Therefore, software based packet classification using fast memory such as SRAM is still widely used on many routers including both edge routers such as Cisco 7200 series and core routers such as Cisco 12000 series. Although SRAM consumes less power and occupies smaller space, it is still costly. Therefore, the proposed shared HyperCuts decision tree can be applied here to help considerably reduce the memory consumption of storing multiple packet filters. The saved memory can be used to more efficiently hold more packet filters and to support more virtual routers.

4.6 Related Work

To the best of our knowledge, this thesis is the first to study how to construct efficient shared data structures for multiple packet filters. The HyperCuts [SBVW03a] decision tree is used in our study because it is one of the most efficient packet classification data structures.

Our work is inspired by Fu and Rexford [FR08a], who observed that the forwarding information bases (FIBs) of different virtual routers on the same physical router share a large number of common prefixes. They proposed to use a shared trie data structure to hold multiple FIBs. They also proposed a corresponding lookup algorithm to search the shared trie data structure. Their evaluation results show that by sharing a trie data structure, the memory requirement can be greatly reduced and the IP lookup time also decreases. However,

their work only focused on merging forwarding tables. How to construct efficient shared data structures for multiple packet filters is not studied. In addition, in their approach, all FIBs are always merged into a single shared FIB, while our approach can automatically classify packet filters into multiple shared HyperCuts decision trees.

Several packet classifier compression techniques (e.g., [MLT09] [Cha09] [DBW⁺06] [AGJ⁺07] [LMZ08]) for TCAM-based packet classification systems have been proposed. However, these techniques are specifically designed for optimizing TCAM-based systems. In addition, they all try to reduce TCAM memory usage by compressing each individual packet classifier, while the key idea of our approach is to save memory by allowing multiple packet filters to efficiently share data structures.

Chapter 5

System Design

5.1 Router Control State Collection

The control state of a router depends on three classes of information: the configuration file of the router (e.g., ACL filtering rules), the direct configuration commands from a controller (e.g. SNMP SET commands), and the dynamic control protocol messages (e.g., OSPF LSA updates) exchanged among routers. The methods for obtaining this information are described below.

Configuration file and configuration commands - Many router functions such as ACL filtering, TOS byte marking, tunneling, ingress filtering and Unicast Reverse Path Forwarding (uRPF) are specified in routers' configuration files. We assume routers' initial configuration files are provided to the state collector by the operator. However, a router's configuration may be updated either by the network operator or by other network management software. In order to enable accurate trajectory error detection, the state collector must know the up-to-date configuration of each router in a timely manner. Therefore, we require any entity that may dynamically change the configurations of routers to immediately notify the state collector about any changes made to a router's configuration. This requirement presents some engineering challenges but is certainly feasible.

Routing control messages - We first illustrate how the control messages of OSPF and BGP can be collected. Then, we extend the discussion to other protocols such as PIM, IGMP, RSVP, etc. In this section, we illustrate how the control message of OSPF and BGP can be collected.

OSPF uses a reliable link state announcement (LSAs) flooding mechanism to exchange LSAs among routers. In order to reduce the flooding overhead, the operator may divide the

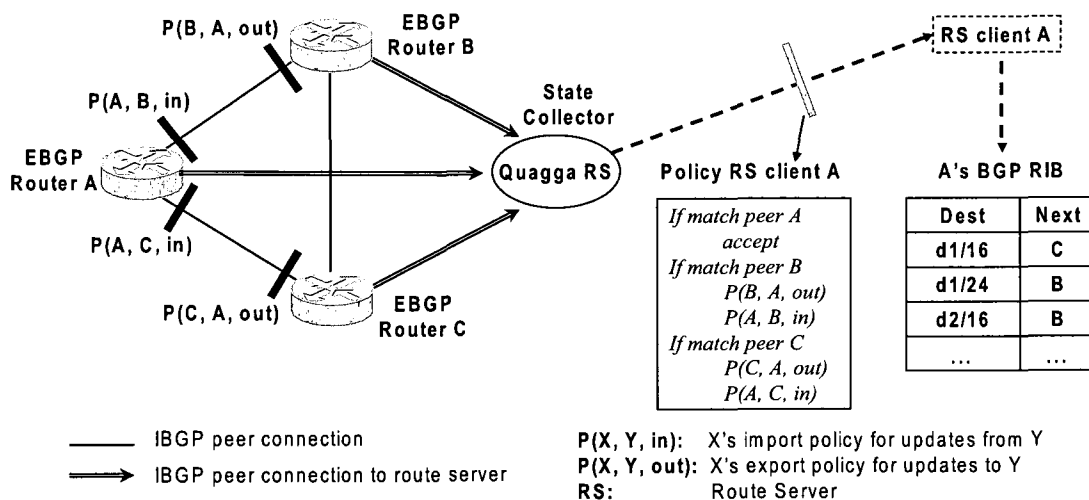


Figure 5.1 : Using a modified Quagga daemon to compute a router's BGP routing state.

whole network into a number of areas. Each OSPF area is an LSA flooding domain and all routers belonging to the same area will receive an identical link state database for that area. For routers belonging to multiple areas, their OSPF routing states are determined by all link state databases they receive. Therefore, in order to obtain the link state databases of each area, the state collector needs to establish an OSPF neighbor adjacency with at least one router from each OSPF area. This adjacency can be established through a virtual tunnel interface. The state collector needs to “speak just enough OSPF” to passively collect all OSPF LSAs. It will neither originate nor flood any LSAs. Once the collector has obtained all link state databases, it can calculate the OSPF routing tables for all routers. A similar approach can be applied to other link state routing protocols such as IS-IS [Cal90].

Collecting BGP updates can be easily accomplished by setting up an IBGP connection between each BGP router in the AS and the collector. However, just collecting all the updates is not enough to reproduce the BGP routing state for an individual router. Because of the import and export policies configured on the IBGP connections among routers, a router may not receive or accept all BGP updates. Therefore, the state collector needs to extract those import and export policies from BGP configuration files and apply them when

computing the BGP routing state of a router.

We have developed a convenient and practical way to do this computation by leveraging a modified version of the open source Quagga BGP daemon software. The basic idea is illustrated in Figure 5.1. First, the state collector runs our modified Quagga daemon configured as a route server.¹ We set up peering relations between the daemon and all the BGP routers in the AS so that the BGP routers will export all BGP updates to the daemon. Then, as shown in the figure, to compute the BGP routing state of a router A, we configure a *virtual route server client A*. This client is virtual because router A will never actually receive BGP updates from the daemon. The import policy of virtual client A is configured to simply duplicate all the policies between router A and its IBGP neighbors in the network as shown in the figure. Therefore, the virtual client A accepts the same BGP updates as router A. Now, by invoking Quagga's built-in BGP decision process on virtual client A, we can conveniently compute the BGP routing state of router A.

However, in a large network, it is very expensive to track the routing tables for all the routers all the time. For OSPF, when there is a link failure, it takes a significant amount of time to recompute the routing tables for all the routers. For BGP, one router's BGP table can be updated very frequently. To make our state collector scalable enough for large networks, we use an on-demand generation strategy to maintain routing tables. Specifically, in OSPF, the system maintains the most updated copy of the link state database in memory, and only computes the OSPF routing tables for the monitored router group when the trajectory error detector requests for them. In BGP, the BGP route server uses a master BGP table structure to maintain the latest updates received from BGP neighbors. Only when the detector requests for the BGP tables of the current router group, the route server generates all the requested BGP tables from the master BGP table in memory. By doing this, the state collector can track the routing state of a large network and efficiently generate the necessary routing tables for trajectory error detection. Once the needed routing tables are

¹The original purpose of a route server is to remove the need for full mesh peering among BGP routers at Internet exchange points.

generated, the shared data structure proposed in [FR08b] can be used to efficiently store them in the memory.

While OSPF and BGP are arguably the most widely used and critical control protocols in the Internet, there are other protocols of interest. For example, to detect multicast forwarding errors, the state collector must learn the multicast forwarding state of routers via multicast control protocol messages such as PIM and IGMP messages. However, unlike OSPF and BGP, PIM and IGMP provide no built-in mechanism for the state collector to reliably obtain a router's multicast state. Fortunately, by configuring the appropriate NetFlow input filters, Cisco's Flexible NetFlow capability [fle] can export the contents of PIM and IGMP messages received by routers (up to the first 1200 bytes of each packet, more than sufficient for PIM and IGMP messages) to an external collector. With copies of PIM and IGMP messages, it is feasible for the state collector to compute the multicast state of routers. Furthermore, since PIM and IGMP messages are periodic in nature (i.e. soft state), even if some messages are lost, the state collector will eventually receive refreshed copies to bring the router state up-to-date.

A similar Flexible NetFlow-based strategy could be applied to other soft state control protocols, for instance, DVMRP, RIP, RSVP, etc. To limit the processing overhead, the state collector can collect these control messages only from the subset of routers whose behavior is being monitored.

Although collecting router control state is feasible, maintaining all the changing control state is challenging. Ideally, we want to maintain a history of state instances for each type of state (e.g., OSPF RIB, BGP RIB, configuration file) on a per router basis. Each state instance is associated with an ideal accurate timestamp, indicating when this state instance started taking effect on the corresponding router. Whenever a new event (e.g., a new OSPF LSA) is received by the state collector, the collector should compute new state instances efficiently and append them to the history. Although the basic idea is conceptually clear, there are challenges to be addressed.

Deciding when a state instance took effect: We assume that routers' clocks are loosely synchronized. Synchronization accuracy of 100ms or better is achievable with NTP. The accuracy of the timestamp associated with a state instance will significantly affect the false trajectory error detection rate of the system. Calculating timestamps of state instances involves multiple factors such as the network topology, propagation delays of links, protocol specific parameters (e.g., Minimum Route Advertisement Interval timer in BGP, LSA hold down timer in OSPF), router software and hardware speed (e.g., how fast the router can write one entry to the forwarding table), packet loss and retransmission (e.g., the retransmission of one lost BGP update will delay BGP convergence). Therefore, it is very challenging to precisely model the above process and calculate an accurate timestamp for each state instance.

Our approach is to associate each state instance with an uncertainty period. We use the uncertainty period to specify a time interval during which the state might be in transition from the old instance to the new one on the router. Therefore, during its uncertainty period, a state instance will not be used for any trajectory error detection because it may produce a false detection with a high probability. Similarly, all collected behavioral evidence for that router within the uncertainty period will not be used either.

Take OSPF control state as an example, it is necessary to estimate different uncertainty periods for LSA updates originating from different routers. For example, given one update originated at router R_i , we may first determine the farthest router R_j that needs to receive the update. Assume the round-trip delay of the path between R_i and R_j is d_{ij} and there are m routers (including R_i and R_j) on the path from R_i to R_j . Then a rough estimate of the length of the uncertainty period P is: $P = \alpha \times d_{ij} + m \times MaxRouterProcessing + m \times ProtocolTimers$, where $MaxRouterProcessing$ stands for the estimated maximum processing time for the corresponding update and $ProtocolTimers$ stands for the protocol specific timer such as the LSA hold-down timer in OSPF. α coarsely represents the effect of packet losses.

Observe that some false detections may exist if the uncertainty period is estimated

poorly. However, we can always cross validate and confirm a detected trajectory error by actively probing the network using, for instance, Cisco IOS IP Service Level Agreements (IPSLA) [ips]. Instances of false detections can thus provide a source of feedback for adapting the uncertainty period estimation algorithm. We will continue to investigate how to leverage such feedback information to improve the estimation of the uncertainty period as our future work.

Efficiently maintaining state instances: We investigated several directions for improving state maintenance efficiency. Recall that a new state instance is inserted to the state history. However, some protocols such as BGP feature frequent updates. Specifically, BGP is in charge of learning routes to all destinations outside of the network, so it typically receives updates from its neighbors frequently. Therefore, it is possible that the BGP routing table will always be in the uncertainty period due to frequent updates. What is worse, the BGP table on a backbone router could be as big as a few megabytes. Thus, generating a new table for every update will consume a lot of resources. Fortunately, one useful observation is that BGP may receive update messages frequently but actual changes to BGP tables are narrow (i.e., only updating a small number of prefixes). Therefore, one optimization is to associate the uncertainty period to a finer-grained object, an IP prefix, instead of the whole BGP table. Then we do not need to generate a new BGP table every time a new update is adopted, instead we only append the new update associated with its uncertainty period to the current BGP table. By doing this, we can minimize the uncertainty period of the whole BGP table and improve the memory utilization. The next problem is that every time when we need to supply the BGP table to the trajectory error detector, we have to start from the initial table and process all update logs to generate an up-to-date table, which could be very inefficient if the trajectory error detector requests BGP tables frequently. Therefore, we propose to periodically process the logged updates to produce a new table, and then we only keep the new table in memory and store all the old processed updates to disk.

When a complete history for each state is maintained by the detector, there might exist

a lot of redundancies in those states across different routers. For example, BGP RIBs of different routers may be very similar. Another example is that routers within the same network may use similar firewall policies. In order to minimize resource consumption, we propose to classify the same type of states from different routers into a small number of groups, where each group contains a set of similar states of the same type. We can then compress the storage of states in the same group by exploiting their similarity. To be specific, let us again take BGP as an example. We first extract all the same BGP table entries from all BGP tables in the same group and store these common entries in a base table. Then for each BGP table, we can create a delta table storing the rest of its BGP table entries that are not in the base table. When one BGP table is updated afterward, we only update the corresponding delta table while still keeping all the same entries in the base table. Periodically, we may need to run the classification algorithm again to re-group the BGP tables in the network. For the multiple delta tables, the shared forwarding table data structure can be used to more efficiently store them.

All the packet filters and access control lists stored on the detector can be efficiently represented using the shared HyperCuts decision trees proposed in Section 4.

5.2 Traffic Trajectories Monitoring and Collection

We leverage NetFlow or its equivalents such as Flexible NetFlow and IPFIX to collect network behavioral evidence from routers. NetFlow or its equivalents are widely supported by commercial routers from Cisco, Juniper, and other vendors for network traffic analysis and monitoring purpose. The NetFlow facility on a router can generate records for traffic flows that go through the router's interfaces in both the inbound and outbound directions. Furthermore, it has the necessary property that inbound flows are recorded before inbound ACL filtering is applied, and outbound flows are recorded after outbound ACL filtering is applied. The collected flow records can be exported to a NetFlow collector host.

A TCP or a UDP flow is defined by the source IP address, destination IP address, source port number, destination port number, and protocol number. Other flows, such as ICMP

flows, are defined by source, destination IP addresses and protocol number. A NetFlow flow record contains a wide variety of accounting information about a flow, such as flow timestamps, number of bytes and packets observed in the flow, IP layer header of packets and TCP flags. Therefore, by analyzing the NetFlow records collected at a target router's neighbors' interfaces, it is possible to determine whether the target router has maliciously dropped a flow, let a flow bypass ACL filtering, mis-forwarded the flow, marked the TOS byte of a flow to steal higher quality service or degrade the flow's service, initiated a new unwanted flow, etc.

NetFlow's filter based sampling feature further allows us to record the full flow information for specific kinds of flows (e.g. specific protocol, specific source, specific port number, etc.). However, we must be careful about how to set up the sampling filters and coordinate these filters on different routers. On one hand, we want the sampling filters to cover all flows going through the routers so that we can detect trajectory errors that affect any flow. On the other hand, if a sampling filter samples a large fraction of traffic on a router, it will bring significant computation overhead on the router. Our strategy to handle this is to set up the sampling filters to cover only a small range of destination port numbers. For destination port numbers that are extremely popular, such as 80 for HTTP traffic, we further use a source port number and destination port number pair to set up the sampling range. By doing this, we can make sure that a sampling filter only cover a small fraction of flows. We then vary the range of port numbers to eventually cover all flows in the network.

We have analyzed the Internet2 NetFlow traces. Based on this analysis, we find that it is easy to restrict the percentage of flows sampled. To give some examples, for unpopular port numbers, specifying the destination port number range from 6000 to 8000 will cover about 1% of all the traffic. On the other hand, pairing a popular destination port number 80 together with source port number range from 3000 to 4200 will also cover 1% of the total traffic. Furthermore, we find that flows that do not have port numbers (e.g. ICMP) account for only a very small fraction (0.23% in our analysis) of the flows. Therefore, our strategy is to sample all such flows.

NetFlow records assembly - Another issue we want to point out here is the need for flow record assembly. The NetFlow facility on routers uses a flow cache to store temporary flow records before exporting them to collectors. When a flow is complete, the record will be exported. NetFlow will also export partial records of active flows to the collector when the cache becomes full to make room for new flows. NetFlow also has an active timer and an inactive timer to control the exporting of flow records. A flow record will be exported if the flow is inactive for a certain time, or if the flow is long lived and lasts greater than the active timer. The consequence is that, a router can export several partial flow records for a network flow. Different routers can export the partial flow records at different time. Furthermore, NetFlow can only provide best-effort service for exporting traffic information. NetFlow supports both UDP and SCTP [Ste07] for the transport of flow records to collectors. However, a flow record can be lost due to packet loss (in case of UDP) or a connection failure (in case of SCTP). The unaligned partial flow records and potential missing records make it a challenging problem to maintain behavioral evidence.

For connection-oriented TCP flows, flows start from SYN packets and finish at FIN or RST packets. Since NetFlow records contain TCP flow flags, a complete flow record for a TCP flow will have both SYN and FIN/RST bit set in TCP flow flags. Partial flow records may only have SYN bit, or FIN/RST bit, or neither of them set. To provide behavioral evidence for TCP flows, the collector must assemble all the related partial flow records into complete flow records. So, given a TCP flow, we can detect whether or not any router misbehaves on this flow by matching complete flow records collected from different routers.

On the other hand, connection-less flows do not have special packets to determine the start and end of flows. Given a UDP flow, what the collector receives is just a sequence of flow records with timestamps. The problem is that, given a flow record at an upstream router, how can we determine which downstream flow record represents the same set of packets? Since different routers may export its flow records at very different time, the alignment of flow records from routers is a challenging problem.

We use the following strategy to address the problems. First, to prevent missing flow records caused by accidental packet loss, we use SCTP to transport NetFlow records. In this case, flow records can only be lost when the SCTP connections between routers and the collector fail. When a connection fails, the flow records from the corresponding router are potentially lost. The collector therefore discards all the partial TCP flow records received from this router. Under normal circumstances, the collector buffers all the partial TCP flow records and assemble them into a complete record after it sees a flow record with the FIN/RST bit set. For connection-less flows, we propose the following heuristic mechanism to align flow records on different routers. The basic idea is that, assuming the network delay jitter is bounded, if the flow experiences a long idle period at an upstream router, it will also experience a long idle period at a downstream router. We can use these long idle periods to align flow records. More specifically, we set both the active timer and the inactive timer to T seconds (e.g. 30 seconds). In this case, as long as a flow is active, NetFlow will export a flow record for it every T seconds. If the collector keeps receiving records for a flow from both the upstream and the downstream routers every T seconds, it will assemble them. If it does not receive records from both upstream and downstream routers for $2T$ seconds, this means the flow has been idle for at least T seconds on both upstream and downstream routers. In this case, the collector will stop assembling and make available the accumulated records for trajectory error detection. Finally, for connection-less flows that are continuously active for a long time, the collector periodically (period $\gg 2T$) stop assembling and make available the accumulated records. In this case, because of the potential misalignment, a difference between the upstream record and the downstream record may not indicate any actual trajectory errors. Instead, we will monitor whether the difference is *diverging* over multiple detection periods to identify a trajectory error.

5.3 Router Trajectory Error Detector

With the collected router state and the network behavioral evidence observed for a monitored router group, the trajectory error detector's task is to periodically process the accu-

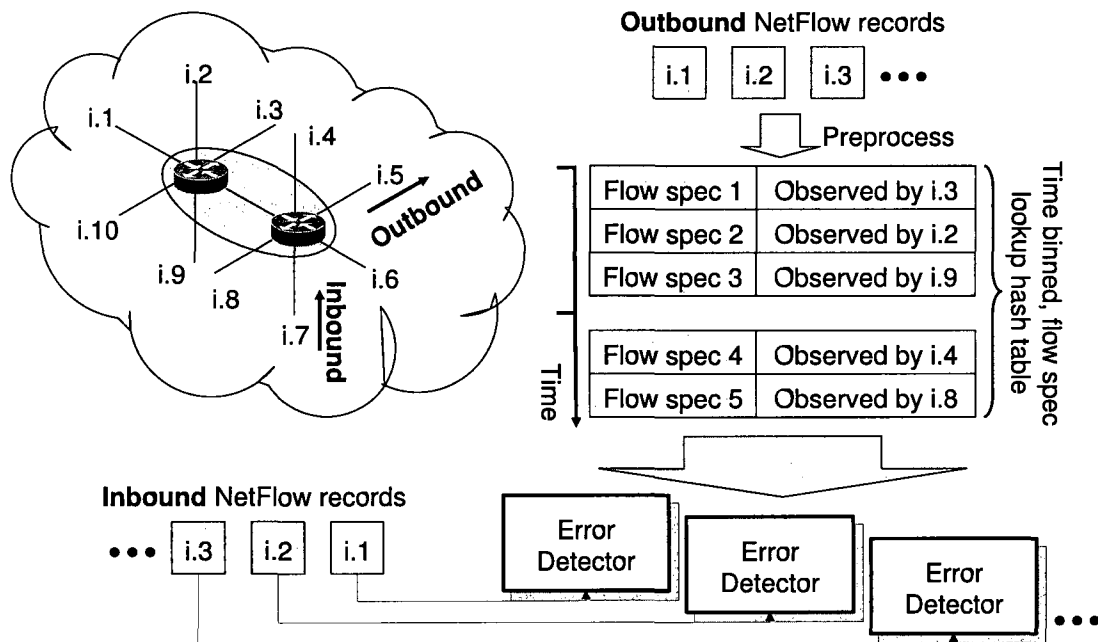


Figure 5.2 : Preprocessing and supply of behavioral evidence to traffic trajectory error detectors.

ulated evidence and report discovered trajectory errors. The main challenge is to design the detector to achieve a high processing throughput. This challenge can be decomposed into two parts. First, the detector must be able to efficiently search through the NetFlow records in the evidence to find out what in reality happened to a particular flow. Second, the detector must be able to efficiently compute based on router state what should have happened to that flow.

We address the first part by preprocessing the outbound NetFlow records into an efficient searchable data structure and by parallelizing the processing of inbound NetFlow records. The high level organization of this approach is presented in Figure 5.2. In the illustrative example, 10 interfaces, i.1 to i.10, are monitored. Each interface reports NetFlow records for both inbound flows (i.e. flows heading into the monitored region) and outbound flows (i.e. flows leaving from the monitored region). Outbound NetFlow records from all monitored interfaces are then gathered and sorted by their timestamps (we assume

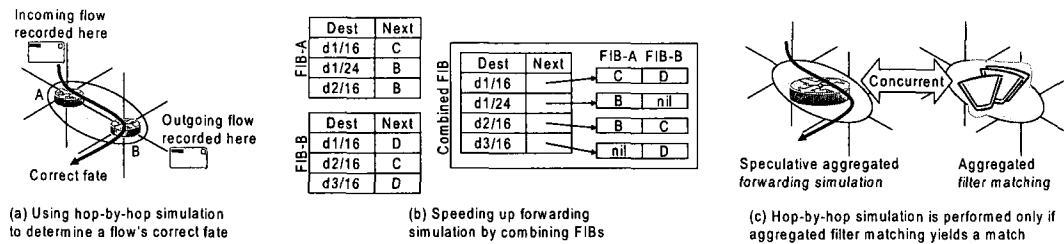


Figure 5.3 : Trajectory error detection mechanisms. (a) The baseline mechanism is to simulate a monitored router group hop-by-hop based on the routers' state. (b) By combining the FIBs in a router group, simulating the aggregated forwarding behavior requires only one longest address prefix match operation. (c) Advanced router behaviors are specified by packet filters. By aggregating the filters in a router group and performing an aggregated match, the detector can efficiently decide whether hop-by-hop simulation can be avoided.

router clocks are loosely synchronized). Those records that belong to the same time period ($t_i, t_{i+1} = t_i + \text{constant}$) are grouped together into a bin. Then, the records in each bin are inserted into a hash table to facilitate the search operation.

In contrast, the processing of the inbound NetFlow records needs no special precomputation. Also, since the processing of each inbound NetFlow record is independent, the computations can be highly parallelized to increase throughput. In the illustrative example, one detector instance is responsible for processing the inbound records from one interface. With the outbound records preprocessed, given an observed inbound NetFlow record at time t_j , the detector simply needs to index into the corresponding time bin (and also the adjacent time bin if t_j is too close to the bin boundary) and search the data structure for the flow specification of the expected outbound flow to determine what happened to the flow. Note that the outbound flow specification may be different from that of the inbound flow if the routers inside the region are configured to modify the flow (e.g. mark the TOS byte, tunnel the flow, etc.). Finally, the detectors can gather all the outbound flow records that are not matched to any inbound flow record and check whether those outbound flows were maliciously fabricated.

To address the second part of the challenge, we add several detector performance im-

provement mechanisms as illustrated in Figure 5.3. First of all, observe that since most network traffic is simply best-effort forwarded, in the majority of the cases, the fate of a flow only depends on routers' FIBs. In the minority of the cases where a flow is specially processed by a router (e.g. filtering, tunneling, TOS marking etc.), the flow must match a certain configured filter of the router. Therefore, in general, filter matching and FIB lookup are two fundamental steps in computing the fate of a flow.

As illustrated in Figure 5.3(a), the baseline mechanism the detector relies on to compute the correct fate of an inbound flow is a hop-by-hop full simulation of the routers' behaviors. However, our goal is to minimize the need for such hop-by-hop simulation to improve performance. Since most flows only require hop-by-hop FIB lookups to determine their fates, it is important to improve the performance of such repeated FIB lookups. We can improve the performance of the FIB lookups by preprocessing the FIBs of routers in a monitored region into a combined FIB as outlined in Figure 5.3(b). The combined FIB makes it possible to determine the outbound interface of a flow by performing only one address prefix match operation, which is the most costly step in a FIB lookup. For instance, suppose an inbound flow with destination address $d1$ arrives at router A . A prefix match in the combined FIB will yield two matching entries $d1/16$ and $d1/24$. As usual, the longest prefix matched is considered first and only if the corresponding FIB entry is "nil" then the shorter prefix matched is considered. Thus, the path of this flow A, B, D can be determined efficiently. Existing FIB aggregation techniques proposed by Fu and Rexford [FR08b] can be directly applied.

A hop-by-hop simulation may still be required if the routers in the region need to apply complex processing to the flow. Operationally, a router matches a received packet against its packet filters to decide whether it needs complex processing. To determine whether such complex processing is necessary in a monitored region, we preprocess all filters of routers in the monitored region into one aggregated filter. Then, for each inbound flow, we perform one filter matching operation against the aggregated filter while at the same time speculatively perform a forwarding lookup in the combined FIB. If the flow does not match

the aggregated filter, then the fate of the flow is determined by the result of the combined FIB lookup. Otherwise, we will fall back on hop-by-hop simulation to determine the fate of the flow.

5.3.1 Discussion

In our design, the detector is composed of a number of key components including collecting control states, collecting traffic trajectories and trajectory error detection logic. Obviously, the implementation of the detector must be more bug-free as possible. Now let us compare the implementations of the detector and a full-fledged router.

A full-fledged router is a complex system composed of a control plane and a data plane. The control plane is responsible for learning how to process data packets and it consists of a large number of complex software modules including routing protocols (e.g., OSPF, IS-IS, RIP, BGP, PIM), signaling protocols (e.g., RSVP, LDP), route management utilities (e.g., route filter, route redistribution), router management interface (e.g., SNMP, router configuration utilities, packet filter configuration) and so on. On the other hand, the data plane is responsible for actual packet processing and it is usually a combination of complex software and hardware. The data plane performs many functions such as forwarding, filtering, NAT, IP option handling, TTL decrement, checksum computation, etc. The control plane and data plane together determine the observable packet processing behavior of a router. Therefore, the implementation of a router is really complex.

Fortunately, implementing the detector is easier and less error-prone than implementing a full router because of the following reasons: Firstly, the detector only handles sampled flow records so it does not need to run at line speed. Thus, instead of using sophisticated algorithms, it can employ simpler and less error-prone algorithms to reduce the number of bugs. For example, in order to check whether the target router filters packets correctly, the verifier controller needs to implement packet classification. Router vendors usually use complex algorithms or even specialized hardware such as TCAMs to achieve high classification speed. However, the detector can just implement the simple HyperCuts algorithm

which is very unlikely to introduce many bugs. Secondly, the detector does not handle data packets directly, instead it only receives the aggregated flow records. As we have shown in the Introduction, a lot of bugs are triggered by data packets. By avoiding handling data packets, the detector is safer. Thirdly, to detect traffic trajectory errors, it is sufficient for the detector to only selectively implement a small set of key packet processing functions, which also helps minimize the implementation complexity of the detector. For example, our detector only detects three types of typical trajectory errors. Fourthly, the detector may only implement the needed protocols instead of all possible protocols in the full-fledged router. Even for each supported, protocol, the detector only needs to implement the required components. For example, to obtain OSPF control messages, the detector only acts as a passive neighbor, it does not need to announce any LSAs nor forward LSAs. Fifthly, the detector is just an application level program and it can run on commodity hardware. It does not require customized fancy hardware and software support. Due to wide availability of the commodity hardware, it is less likely that they have many bugs. Lastly, because the code base of the detector is much smaller than the full implementation of a full-fledged router, it is more feasible for us to apply software testing and formal verification techniques to more thoroughly test and verify the correctness of the implementation of the detector.

Chapter 6

Prototype Evaluation

6.1 Prototype System Implementation

We have implemented a complete prototype system. It has three major parts: control state collector, the NetFlow evidence collector and the trajectory error detector. We leverage the open source Quagga routing software (version 0.98.6) [quab] to implement the control state collector. We modify the Quagga ospfd and bgpd route server code to support on-demand OSPF and BGP table generation. We use the open source flow-tools [floa] collector as our NetFlow evidence collector. The flow-tools collector will receive NetFlow record packets from routers' NetFlow sampling facility and manage them in the flow record files. The trajectory error detector requests the control state data and NetFlow records from the state collector and the evidence collector, and checks routers' behaviors. The detector uses a HyperCuts decision tree [SBVW03b] based packet classification algorithm to encode filters and a trie data structure to encode forwarding tables to speed up the filter match and forwarding table lookup procedures. Our current prototype implementation only supports hop-by-hop simulation to determine a router group's behavior.

6.2 Emulab Testbed Setup

We set up a testbed with the Internet2 topology on Emulab. As shown in Figure 6.1, there are 9 routers in the testbed emulating the 9 core routers in Internet2. To closely emulate the real Internet2 routers, we set up these routers using Juniper JUNOS [juna] version 8.5 running on FreeBSD. This use of JUNOS is also widely known as Olive [oli]. Olive can precisely emulate a Juniper router. The only difference between an Olive and a Juniper

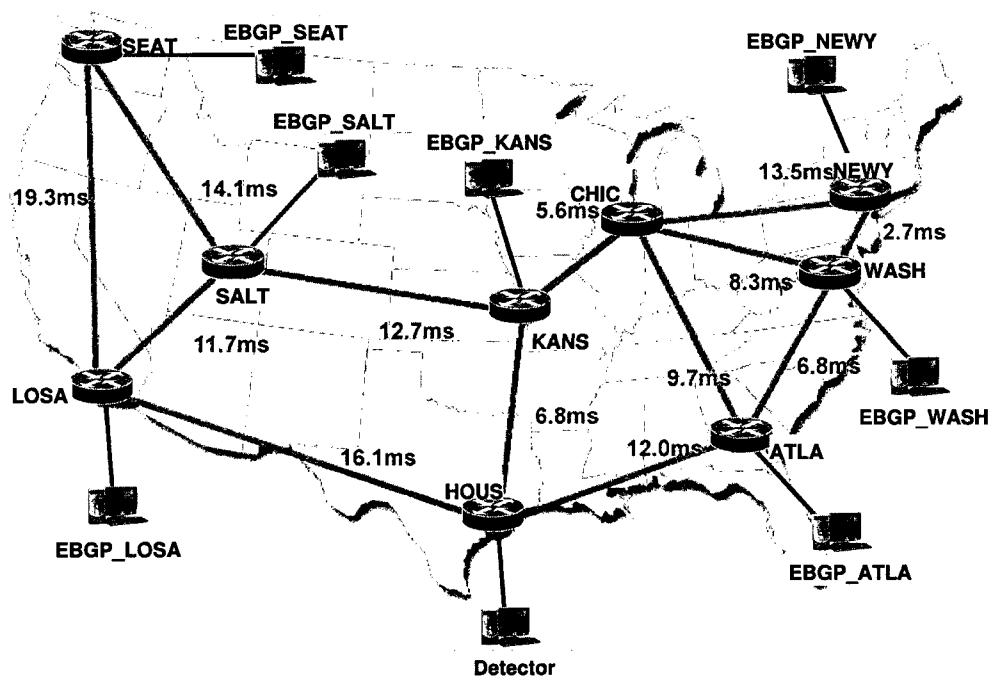


Figure 6.1 : Emulated Internet2 testbed on EmuLab.

router is that, a real router runs JUNOS on a special hardware packet forwarding engine, while Olive runs JUNOS on a commodity PC. We use Emulab PC3000 machines to set up the Olive routers. PC3000 are the machines with Intel Xeon 3.0GHz 64bit CPU, 2GB RAM and 5 Ethernet NICs. We first set up the QEMU (version 0.9.1) [qem] virtualization environment on the PC3000 to provide the particular NIC models required by Olive, and install the Olive software router inside QEMU virtual environment. QEMU then runs on top of a host Linux system.

The configurations of these 9 Olive routers are based on the real configuration files of Internet2 routers. Since Quagga does not support IS-IS, we translate Internet2's IS-IS configuration into an equivalent OSPF configuration. We use one PC directly connected with each Olive router to emulate its EBGP neighbor and inject BGP routes into it. To inject BGP routes, we built a simple tool to load BGP update messages from real routers'

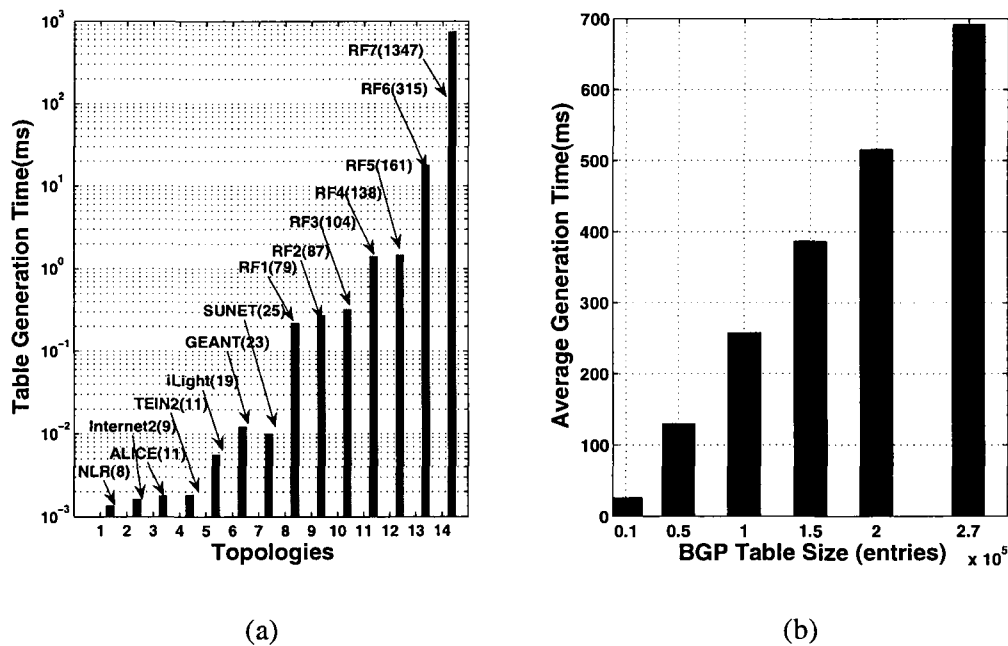


Figure 6.2 : The performance of OSPF and BGP control state collection. (a) OSPF. (b) BGP.

BGP updates trace, set up a BGP connection with the Olive router, and announce these update messages to it. We tested the forwarding performance of Olive routers. For an Olive router with 270k routing table entries and no packet filter, it can forward 5300 packets/s without packet loss. After we enable filters on the router, its throughput drops to 4700 packets/s. Since Olive routers do not support NetFlow sampling, we use the IPCAD (version 3.7.3) [ipc] tool to sample packets on the QEMU interfaces to emulate the NetFlow on routers. We deploy our detection system on a PC3000 machine connected to the HOUS router.

6.3 Performance of Router Control State Collection

To evaluate the performance of our router control state collector, we measure the time it takes to generate OSPF and BGP routing tables of different sizes. For OSPF routing tables,

we use a simulation experiment to inject LSAs for networks with different topologies and sizes into the Quagga based OSPF state collector. These topologies include the real network topologies we have used for previous experiments. To study the OSPF table computation time for larger networks, we also include Rocketfuel (RF) topologies [SMW02] in this experiment. Figure 6.2(a) shows the average time it takes to compute an OSPF table with different sizes of networks. The sizes of these networks are shown in the label text. We can see that the state collector can compute OSPF tables quickly. Even in a large network with 1347 routers and 6225 links, it only takes 700 ms to generate the OSPF table for one router. For the Internet2 network, it only takes 0.02 ms to generate the OSPF routing tables of all the 9 routers.

To evaluate the performance of the BGP routing state collector, we inject different numbers of BGP routes into the Internet2 routers in our testbed. The BGP update messages are from the RouteViews router traces. We use our BGP route server to collect all the updates from the Internet2 routers, and reproduce the BGP tables of all the 9 Internet2 routers. Based on our experiment, the BGP route server can process BGP update messages very efficiently. Even after the route server has collected the whole 270,000 BGP routes, it can still process an update message in 76 μ s on average. Figure 6.2(b) shows the average time for the route server to generate one BGP table at different sizes. We can see that, the BGP route server can generate BGP tables very quickly. Even for a BGP table with all the 270,000 entries for the whole Internet, it takes 690 ms to generate one table on average. The Internet2 BGP table has only 12000 entries, and it only takes 275 ms to generate the BGP tables for all the 9 Internet2 routers.

6.4 Performance of Traffic Trajectories Collection

We use the open source flow-tools (version 0.68) software as the behavioral evidence collector. We design the following experiment to test how many flow records the flow-tools collector can handle in one second. First we run the flow-capture daemon as a collector in our testbed. We use the flow-send daemon to load flow records from the Internet2 NetFlow

traces and send the flow records to the collector. To test the limit of the flow collector, we simultaneously run the flow-send daemon on 9 nodes. All the nodes keep loading flow records from trace files and send them to the collector. We found that each node can send 700 flow report packets per second without packet loss, where each packet contains 22 flow records. This means the collector can handle 138,600 flow records per second from all 9 senders and manage them in the disk archives without flow record loss. Considering in reality, all the 9 Internet2 routers only generate 1,200 flow records per second together (Internet2 routers enable random NetFlow sampling at the rate of one out of 100 packets), our results show that the flow-tools collector can easily handle a large volume of NetFlow record data and effectively collect behavioral evidence from large networks.

6.5 Performance of Trajectory Error Detection

Three major components in the trajectory error detector are critical to the detection performance. The forwarding table trie and filter decision tree components decide how fast the detector can compute the expected behavior of a flow. The flow hash table component decides how fast the detector can find out the flow's real behavior from a large number of NetFlow records. To understand the performance of the detector, we first evaluate the performance of each major component individually. We first test how much time it takes to build a large trie, a large decision tree and a flow hash table. We found that it takes 2.3s to build a trie with 260K prefixes, and it takes 3.3s to build a HyperCuts decision tree for a filter with 10,000 rules generated by ClassBench [TT05a]. Building a flow hash table is much faster. Even for a flow hash table with 100K records, it only takes 0.14s to build the hash table. Note that building these data structures is a one time operation before a router group is monitored. We expect a router group is monitored for several minutes before the system switches to a different router group. Thus, needing tens of seconds of preparation time is still acceptable.

Figure 6.3 shows the average lookup time of the trie, the decision tree and the flow hash table. We can see that it is very efficient to perform lookup operations on them. Even for

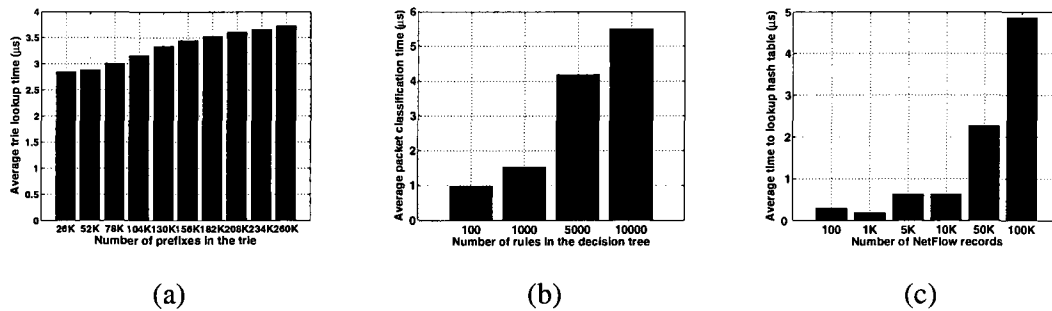


Figure 6.3 : The performance of detector components. (a) Trie lookup time. (b) Decision tree lookup time. (c) Flow hash table lookup time.

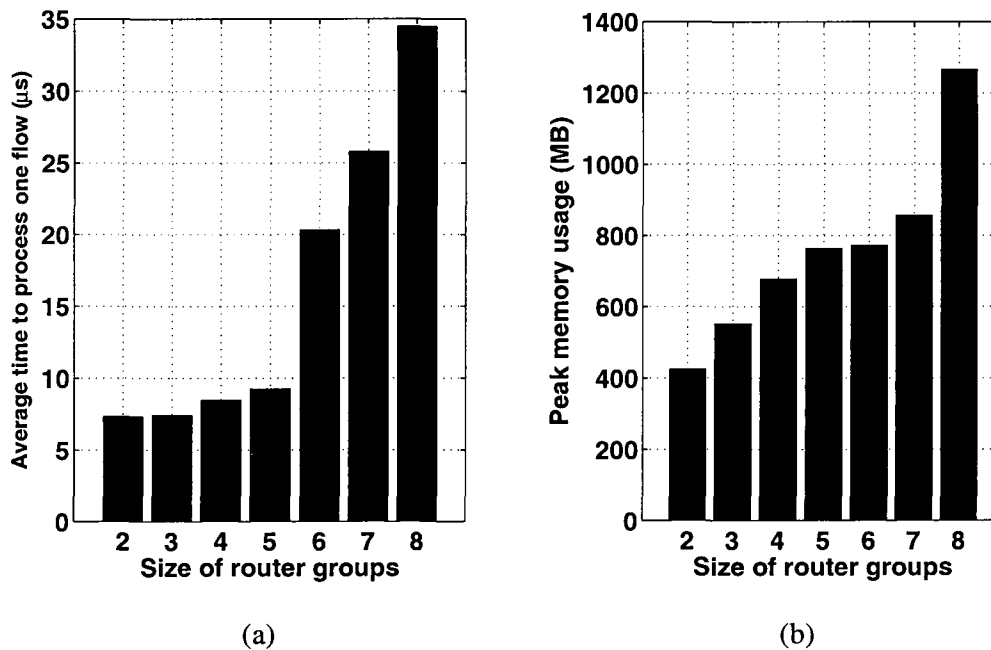


Figure 6.4 : The overall performance of trajectory error detection. (a) Average processing time for each flow. (b) Peak memory usage during the detection.

a trie with 260K prefixes, it only takes $3.7\mu s$ to lookup a prefix on average. For a filter decision tree with 10,000 rules, it only takes $5.5\mu s$ to classify a packet on average. It takes $4.8\mu s$ to lookup a flow record from 100K records.

We conduct a complete experiment in our testbed to evaluate the overall performance of the detector. In the experiment, each router has 270K prefixes in its routing table. A simple synthetic 10-rule filter generated by [TT05a] is enabled as the input filter on each interface. Real Internet2 traces are used to generate flows. We vary the router group size and measure the average time for processing each flow record for trajectory error detection and the peak memory usage. The results are shown in Figure 6.4. As can be seen, the detector can process a large number of flows quickly. For example, even if each router group contains 8 routers, the average processing time for each flow is around $34 \mu\text{s}$, i.e., 30,000 flows per second. For comparison, currently the 9 real Internet2 routers only generate a total of 1200 flow records per second.

6.6 Integrated Trajectory Error Detection Demonstration

We set up an integrated experiment to demonstrate the use of our system to monitor our emulated Internet2 network. Based on the Internet2 network topology, we select router groups using the group selection algorithm described in Section 3.3 with $MaxInterface = 10$, $MaxRouter = 8$, and $\alpha = 0.5$. Four router groups are returned by the selection algorithm: $G1 = \{CHIC, HOUS, LOSA, SEAT, KANS, WASH\}$, $G2 = \{CHIC, HOUS, LOSA, SALT, NEWY, ATLA\}$, $G3 = \{HOUS, SEAT, SALT, KANS, NEWY, WASH, ATLA\}$ and $G4 = \{CHIC, LOSA, SEAT, SALT, KANS, NEWY, WASH, ATLA\}$. We monitor each router group for 150 seconds. All four router groups are monitored repeatedly. We inject traffic from a host connecting to ATLA by playing back real Internet2 traces. We artificially introduce 15 forwarding errors across all routers by adding static routes in the Olive routers while not informing the detector about these configuration changes. We selectively add those static routes to only affect long-lived flows so that we can simulate persistent trajectory errors.

The four router groups can detect 6, 2, 7 and 2 of the 15 trajectory errors respectively. All 15 trajectory errors are detected eventually. Some trajectory errors are detected by more than one router group. Table 6.1 shows the time when each trajectory error is detected for

Trajectory error:	1	2	3	4	5
Time(s):	152.10	152.11	152.12	152.22	152.33
Trajectory error:	6	7	8	9	10
Time(s):	152.86	304.03	304.03	456.01	456.02
Trajectory error:	11	12	13	14	15
Time(s):	456.04	456.12	456.13	608.52	608.53

Table 6.1 : Timestamp of each trajectory error first detected.

the first time since the detector started. In the current implementation, the detector only starts detection when one monitoring period is completed. In a future version, we will eliminate this limitation and perform live detection so that the detection times will all be reduced by 150s.

In addition to quickly detecting artificially introduced trajectory errors, our system also detected a trajectory error that was caused by bugs in our BGP update injector and our router configuration. Specifically, because we are emulating Internet2 on Emulab, in order to avoid confusion, we do not allow the BGP update injectors to announce any IP prefixes belonging to the University of Utah and the real Internet2. However, a bug in our BGP update injector caused one Internet2 prefix 64.57.8.1/31 to be announced into the testbed. In addition, in the Olive configuration, we accidentally gave routes learned from BGP a higher preference. Consequently, this BGP route was chosen by the Olive routers, and all traffic destined to that prefix was routed to the BGP update injector running on a host connecting to SALT, creating a trajectory error. On the other hand, the detector correctly calculated its route to this prefix based on LSAs flooded by OSPF. The detector accurately detected this routing inconsistency and helped us discover the bug in our BGP update injector and router configuration!

Chapter 7

Conclusion and Future Work

Traffic trajectory errors are serious problems to an operational network because they may disrupt network services, cause network applications to fail and create security loopholes for network intruders to exploit. Therefore, traffic trajectory errors must be detected quickly and efficiently when they are triggered in the field. There has been recent work on designing efficient traffic trajectory error detection systems; nonetheless they have seen limited deployment in operational networks. This thesis presents important contributions towards making traffic trajectory error detection systems more efficient and more attractive to network operators. First, we present an efficient trajectory monitoring technique called router group monitoring. The proposed technique can greatly reduce the monitoring overhead and increase the error detection speed. Second, we propose a novel shared data structure to efficiently store and lookup a large number of packet filters inside a router group. Third, we have built a complete prototype system based on the first two contributions. Unlike many existing trajectory error detection systems that require modifications to existing routers' hardware and software, our prototype is completely compatible with Juniper's JUNOS, which makes it immediately deployable in a real network.

We now present a summary of the main results of this thesis and directions for future work.

7.1 Summary of Contributions

Our first contribution is the router group monitoring technique. The idea started with a simple observation: To detect a traffic trajectory error in a network, it is unnecessary to monitor all network interfaces. However, how to exploit this observation was not entirely

obvious. This thesis has explored one class of strategy called router group monitoring. To understand the potential of this strategy, we have studied numerous real network topologies and found that router group monitoring is surprisingly effective. To make this idea practical, we have derived an analytical model to predict the effectiveness of a router group based on three identified important factors that affect router groups' error detection performance. In addition, we have also designed an efficient algorithm for selecting sets of router groups with complete error coverage and fast error detection under monitoring resource constraints. The analytical model provides key insights on the factors that determine the error detection rate. Our router group selection algorithm, when applied to Trajectory Sampling, can improve detection speed by up to a factor of 4, and when applied to Fatih, can reduce the communication overhead by up to 85%. Interestingly, router group monitoring is just one of possibly many interface selection strategies that remain to be explored.

The second contribution of the thesis is to exploit the feasibility of efficiently representing multiple packet filters using a shared data structure. Concretely, our thesis is the first to study how to construct an efficient shared data structure based on the HyperCuts tree for multiple packet filters. We have identified a set of important factors that can affect the performance of the constructed shared HyperCuts decision trees. We then propose a novel approach to clustering packet filters into shared HyperCuts decision trees. Our evaluation using both real packet filters and synthetic packet filters shows that our shared HyperCuts decision trees can reduce up to 50% of the memory consumption while keeping the average height of trees the same as the separate trees. In addition, the shared HyperCuts decision trees enable concurrent lookup of multiple packet filters sharing the same tree. We also show that the proposed approach is practical. It only takes a few minutes to finish clustering 1,000 packet filters and to construct the corresponding shared HyperCuts decision trees.

The third contribution is the proof-of-concept implementation of a prototype trajectory error detection system. The prototype system is completely compatible with Juniper's JUNOS. Our micro-benchmark experiments show that the system can monitor a real net-

work with ease. Our live system demonstration shows that the system is realistic and immediately deployable.

7.2 Future Work

We have shown that the router group monitoring is a powerful technique to improve the efficiency of the traffic trajectory monitoring. However, our heuristic router group selection algorithm is by no means an optimal solution. Instead, it only serves as the first step towards designing an optimal router group selection algorithm. To continue to improve the efficiency of traffic trajectory monitoring, more efforts are needed to find the optimal router group selection algorithm that provides both full coverage and fastest error detection. In addition, the current router group selection algorithm still lacks of flexibility to cope with some practical constraints. For example, it implicitly assumes that each link carries the same traffic volume, so we always assume the detector cannot monitor more than *MaxInterface* interfaces simultaneously. However, in a real network, links could be carrying very different traffic volumes. Thus, the router group selection algorithm needs to incorporate this constraint. In addition, the current selection algorithm tends to use interfaces on high-degree nodes to monitor multiple router groups simultaneously. Therefore, those high-degree nodes might need to monitor many of its interfaces simultaneously, which may potentially overload high-degree nodes. It would be important to develop a more general router group selection algorithm that can cope with as many practical constraints as possible.

After a trajectory error is detected inside a router group, how to quickly determine the actual misbehaving router remains a problem. Some straight-forward approaches do exist. For example, we can randomly split the router group into two halves and then monitor each half respectively to determine which half contains the misbehaving router and then continue the binary search on the half containing the misbehaving router. We can also monitor all the routers along the expected forwarding path one by one. It would be important to determine the optimal strategy for localizing the misbehaving router. One potential first step is to

leverage the forwarding states and router configurations inside the router group to more quickly narrow down the list of suspicious routers. Given different types of router errors, different fault localization algorithms may be needed.

The shared HyperCuts tree is shown to be effective in efficiently maintaining multiple packet filters. However, in practice, packet filters may be updated frequently. Therefore, how to efficiently cope with the dynamics of packet filters is a problem that needs to be carefully studied. That is, there is a need for efficient mechanisms for incrementally updating the shared decision trees when some packet filters are changed. It would be also very useful to study whether our proposed technique can be applied to other data structures that can represent packet filters (e.g., the decision diagram [GL04]).

Bibliography

- [ACL00] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proceedings of ACM Symposium on Theory of Computing*, pages 171–180, 2000.
- [AGJ⁺07] D. Applegate, G. Galinescu, D. Johnson, H. Karloff, K. Ligett, and J. Wang. Compressing Rectilinear Pictures and Minimizing Access Control Lists. In *ACM SODA*, 2007.
- [AHNRR02] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens. An on-demand secure routing protocol resilient to byzantine failures. In *Proceedings of ACM Workshop on Wireless Secure*, 2002.
- [AKWK04] I. Avramopoulos, H. Kobayashi, R. Wang, and A. Krishnamurthy. Highly secure and efficient routing. In *Proc. IEEE INFOCOM*, March 2004.
- [AMCS04] K. Argyraki, P. Maniatis, D. Cheriton, and S. Shenker. Providing packet obituaries. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Networking*, November 2004.
- [APST05] T. Anderson, L. Peterson, S. Shenker, and T. Turner. Overcoming the Internet Impasse Through Virtualization. In *IEEE Computer*, vol. 38, no.4, May 2005.
- [AR06] I. Avramopoulos and J. Rexford. Stealth probing: Efficient data-plane security for IP routing. In *USENIX Annual Technical Conference*, 2006.
- [BCP⁺98] K. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R. Olsson. Detecting Disruptive Routers: A Distributed Network Monitoring Approach. In *IEEE Network*, 1998.

- [BFH⁺06] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In vini veritas: realistic and controlled network experimentation. In *Proc. ACM SIGCOMM*, September 2006.
- [BHBR01] S. Basagni, K. Herrin, D. Bruschi, and E. Rosti. Secure pebblenets. In *Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, pages 156–163. ACM New York, NY, USA, 2001.
- [Cal90] R. Callon. *RFC 1195 - Use of OSI IS-IS for routing in TCP/IP and dual environments*, 1990.
- [CFR05] C. Chaudet, E. Fleury, and H. Rivano. Optimal Positioning of Active and Passive Monitoring Devices. In *ACM CoNEXT*, October 2005.
- [Cha09] Chad R. Meiners and Alex X. Liu and Eric Torng. Topological Transformation Approaches to Optimizing TCAM-Based Packet Processing System. In *ACM SIGMETRICS*, 2009.
- [Che97] S. Cheung. An efficient message authentication scheme for link state routing. In *13th Annual Computer Security Applications Conference*, pages 90–98. IEEE Computer Society, 1997.
- [CIB⁺06] G. Cantieni, G. Iannaccone, C. Barakat, C. Diot, and P. Thiran. Reformulating the monitor placement problem: optimal network-wide sampling. In *ACM CoNEXT*, December 2006.
- [cisa] Cisco Logical Routers. http://www.cisco.com/en/US/docs/ios_xr_sw/iosxr_r3.2/interfaces/command%2Freference/hr321r.html.
- [cisb] Cisco Security Advisories and Notices. http://www.cisco.com/en/US/products/products_security_advisories_listin%2Fg.html.

- [cisc] Cisco Systems Inc. <http://www.cisco.com/>.
- [cisd] Cisco Traffic Anomaly Detection and Mitigation Solutions. http://www.cisco.com/en/US/prod/collateral/vpndevc/ps5879/ps6264/ps5887%/prod_bulletin0900aecd800fd124_ps5888_Products_Bulletin.html.
- [Cise] Cisco, Inc. Policy-based routing, white paper. http://www.cisco.com/warp/public/732/Tech/plicy_wp.pdf.
- [CR08] M. Caesar and J. Rexford. Building Bug-tolerant Routers with Virtualization. *Proceedings of PRESTO'08*, 2008.
- [DBW⁺06] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla. Packet Classifiers in Ternary CAMs Can Be Smaller. In *ACM SIGMETRICS*, 2006.
- [Den87] D. Denning. An intrusion detection model. *IEEE Transaction on Software Engineering*, February 1987.
- [DG00] Nick G. Duffield and Matthias Grossglauser. Trajectory sampling for direct traffic observation. In *Proc. ACM SIGCOMM*, pages 271–282, 2000.
- [EKMV04] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a Better NetFlow. In *ACM SIGCOMM 2004*, August 2004.
- [FB05] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. *NSDI*, 2005.
- [fle] Cisco IOS Flexible NetFlow Technology White Paper. http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6555/ps660%1/ps6965/prod_white_paper0900aecd804be1cc.html.
- [floa] Flow-tools. <http://www.splintered.net/sw/flow-tools/>.

- [flob] FlowMon Probe. <http://www.invea-tech.com/products/flowmon-probes>.
- [FR01] A. Feldmann and J. Rexford. IP Network Configuration for Intradomain Traffic Engineering. *IEEE Network*, 2001.
- [FR08a] J. Fu and J. Rexford. Efficient IP-Address Lookup with a Shared Forwarding Table For Multiple Virtual Routers. In *ACM CoNEXT*, 2008.
- [FR08b] J. Fu and J. Rexford. Efficient ip-address lookup with a shared forwarding table for multiple virtual routers. In *Proceedings of CoNEXT'08*, 2008.
- [FSBK03] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred. Statistical approaches to ddos attack detection and response. *Proceedings of DARPA Information Survivability Conference and Exposition*, 2003.
- [GL04] M. G. Gouda and A. X. Liu. Firewall Design: Consistency, Completeness and Compactness. In *Proceedings of 24th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2004.
- [GM99] P. Gupta and N. McKeown. Packet Classification Using Hierarchical Intelligent Cuttings. In *Hot Interconnects*, 1999.
- [GXT⁺08] S. Goldberg, D. Xiao, E. Tromer, B. Barak, and J. Rexford. Path quality monitoring in the presence of adversaries. In *Proc. ACM SIGMETRICS*, June 2008.
- [HAB00] J. Hughes, T. Aura, and M. Bishop. Using conservation of flow as a security mechanism in network protocols. In *IEEE Symposium on Security and Privacy*, May 2000.
- [HARD09] Andreas Haeberlen, Ioannis Avramopoulos, Jennifer Rexford, and Peter Druschel. Netreview: Detecting when interdomain routing goes wrong. In *NSDI 2009*, April 2009.

- [HJP03] Y.C. Hu, D.B. Johnson, and A. Perrig. SEAD: secure efficient distance vector routing for mobile wireless ad hoc networks. *Ad Hoc Networks*, 1(1):175–192, 2003.
- [HK00] A. Herzberg and S. Kutten. Early detection of message forwarding faults. *SIAM J. Comput.*, (4), 2000.
- [HKD07] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. In *ACM SOSP 2007*, October 2007.
- [HLO03] J. Horton and A. Lopez-Ortiz. On the number of distributed measurement points for network tomography. In *USENIX IMC*, November 2003.
- [HPT97] R. Hauser, A. Przygienda, and G. Tsudik. Reducing the Cost of Security in Link State Routing. In *Symposium on Network and Distributed Systems Security (NDSS 97)*, pages 93–99, 1997.
- [ibm] IBM ISS Network Anomaly Detection and Behavior Analysis. <http://www-935.ibm.com/services/us/iss/pdf/proventia-network-anomaly-detection-system-ss.pdf>.
- [ipc] IPCAD: IP accounting daemon. <http://lionet.info/ipcad/>.
- [ipf] IP Flow Information Export (ipfix). <http://www.ietf.org/html.charters/ipfix-charter.html>.
- [ips] Cisco IOS IP Service Level Agreements. http://www.cisco.com/en/US/technologies/tk648/tk362/tk920/technologies_%white_paper0900aec8017f8c9.html.
- [JMS⁺07] A. Jackson, W. Milliken, C. A. Santivanez, M. Condell, and W. Strayer. A Topological Analysis of Monitor Placement. In *IEEE Sixth International Symposium on Network Computing and Applications*, July 2007.

- [juna] Juniper JUNOS Software: Network Operating System. http://www.juniper.net/products_and_services/junos/.
- [junb] Juniper Logical Routers. <http://www.juniper.net/techpubs/software/junos/junos85/feature-guide-85%/id-11139212.html>.
- [Junc] Juniper Networks, Inc. Intelligent Logical Router Service. www.juniper.net/solutions/literature/white_papers/200097.pdf.
- [Jund] Juniper Networks, Inc. Security Notices. http://www.juniper.net/support/security/security_notices.html.
- [KE05] R. Kompella and C. Estan. The Power of Slicing in Internet Flow Measurement. In *USENIX IMC 2005*, October 2005.
- [KLMS00] S. Kent, C. Lynn, J. Mikkelsen, and K. Sen. Secure border gateway protocol (secure-bgp). *IEEE J. Selected Areas in Communications*, (4), April 2000.
- [Kre92] M. Krentel. Generalizations of opt p to the polynomial hierarchy. In *Theor. Compu. Sci.* 97 (2):183-198, 1992.
- [Kum93] B. Kumar. Integration of security in network routing protocols. *ACM SIGSAC Review*, 11(2):18–25, 1993.
- [KYCR09] Eric Keller, Minlan Yu, Matthew Caesar, and Jennifer Rexford. Virtually eliminating router bugs. *Proceedings of ACM CoNext'09*, 2009.
- [LCD04] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *SIGCOMM*, 2004.
- [LCD05] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. In *SIGCOMM*, 2005.

- [Lek03] Panos Lekkas. *Network Processors - Architectures, Protocols, and Platforms*. 2003.
- [LMZ08] Alex X. Liu, Chad R. Meiners, and Yun Zhou. All-match based complete redundancy removal for packet classifiers in TCAMs. In *IEEE INFOCOM*, 2008.
- [LWK06] S. Lee, T. Wong, and H. S. Kim. Secure split assignment trajectory sampling: a malicious router detection system. In *IEEE DSN'06*, 2006.
- [LX01] W. Lee and D. Xiang. Information theoretic measure for anomaly detection. *IEEE Symposium on Security and Privacy*, May 2001.
- [MB96] S. L. Murphy and M. R. Badger. Digital signature protection of the ospf routing protocol. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS'96)*, February 1996.
- [MCMS05] A. Mizrak, Y. Cheng, K. Marzullo, and S. Savage. Fatih: Detecting and Isolating Malicious Routers. In *DSN*, 2005.
- [MCMS06] A. Mizrak, Y. Cheng, K. Marzullo, and S. Savage. Detecting and Isolating Malicious Routers. In *IEEE Transaction on Dependable and Secure Computing*, 2006.
- [Mit97] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.
- [Miz07] Alper T. Mizrak. *Detecting Malicious Routers*. PhD thesis, University of California, San Diego, September 2007.
- [MLT09] Chad R. Meiners, Alex X. Liu, and Eric Torng. Bit Weaving: A Non-prefix Approach to Compressing Packet Classifiers in TCAMs. In *IEEE ICNP*, 2009.

- [MND05] M. Molina, S. Niccolini, and N.G. Duffield. A Comparative Experimental Study of Hash Functions Applied to Packet Sampling. In *Proc. of International Teletraffic Congress (ITC)*, 2005.
- [MSM08] A. Mizrak, S. Savage, and K. Marzullo. Detecting compromised routers via packet forwarding behavior. *IEEE Network Magazine*, March/April 2008.
- [nev] Nevis Networks. <http://www.nevisnetworks.com/>.
- [new] NewBridge Networks. <http://www.alcatel-lucent.com/>.
- [oli] Juniper Olive Router. <http://juniper.cluepon.net/index.php/Olive>.
- [P.] P. Psenak and S. Mirtorabi and A. Roy and L. Nguyen and P. Pillay-Esnault. Multi-Topology (MT) Routing in OSPF. IETF RFC 4915, 2007.
- [Per88] R. Perlman. *Network Layer Protocols with Byzantine Robustness*. PhD thesis, Dept. of EECS, MIT, 1988.
- [Per00] R. Perlman. *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley Professional, 2000.
- [PS03] V. Padmanabhan and D. Simon. Secure Traceroute to Detect Faulty or Malicious Routing. In *CCR*, 2003.
- [PSS⁺01] C. Patridge, A. C. Snoeren, W. T. Strayer, B. Schwartz, M. Condell, and I. Castineyra. Fire: Flexible intra-as routing environment. *IEEE J. Selected Areas in Communications*, 19(3), 2001.
- [PST⁺02] A. Perrig, R. Szewczyk, JD Tygar, V. Wen, and D.E. Culler. SPINS: Security Protocols for Sensor Networks. *Wireless Networks*, 8(5):521–534, 2002.
- [qem] QEMU: Open Source Processor Emulator. <http://bellard.org/qemu/>.

- [quaa] Quagga Bugzilla. <http://bugzilla.quagga.net/>.
- [quab] Quagga Software Routing Suite. <http://www.quagga.net/>.
- [RMR07] R. K. Rajendran, V. Misra, and D. Rubenstein. Theoretical Bounds on Control-plane Self-monitoring in routing protocols. *Proceedings of SIGMETRICS'07, 2007*.
- [SBVW03a] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In *ACM SIGCOMM, 2003*.
- [SBVW03b] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *ACM SIGCOMM 2003, August 2003*.
- [SG04] A. Shaikh and A. Greenberg. OSPF Monitoring: Architecture, Design and Deployment Experience. In *USENIX NSDI, 2004*.
- [SGKT05] K. Suh, Y. Guo, J. Kurose, and D. Towsley. Locating network monitors: Complexity, heuristics and coverage. March 2005.
- [SMGLA97] B.R. Smith, S. Murthy, and JJ Garcia-Luna-Aceves. Securing Distance-Vector Routing Protocols. In *NDSS, February 1997*.
- [SMW02] Neil Spring, Ratul Mahajan, and David Wetheral. Measuring ISP topologies with RocketFuel. In *SIGCOMM, August 2002*.
- [sne] Sampled Netflow. http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newf%t/120limit/120s/120s11/12s_sanf.htm.
- [spr] Sprint IP Data Analysis Research Project. <https://research.sprintlabs.com/packstat/packetoverview.php>.

- [SRS⁺04] L. Subramanian, V. Roth, I. Stoica, S. Shenker, and R. H. Katz. Listen and whisper: Security mechanisms for bgp. In *Proc. Networked Systems Design and Implementation*, March 2004.
- [SRW⁺08] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. cSAMP: A System for Network-Wide Flow Monitoring. In *USENIX NSDI*, 2008.
- [SRXM08] Y. Sung, S. Rao, G. Xie, and D. Maltz. Towards Systematic Design of Enterprise Networks. In *ACM CoNEXT*, 2008.
- [Ste07] R. Stewart. *Stream Control Transmission Protocol*. Internet Engineering Task Force, Sep 2007. RFC 4960.
- [T.] T. Przygienda and N. Shen and N. Sheth. Multi-Topology (MT) Routing in Intermediate System to Intermediate Systems (IS-ISs). IETF RFC 5120, 2008.
- [Tay05] David E. Taylor. Survey and Taxonomy of Packet Classification Techniques. In *ACM Computing Surveys*, vol. 37, no 3, 2005.
- [TT05a] David Taylor and Jonathan Turner. Classbench: A packet classification benchmark. In *INFOCOM 2005*, March 2005.
- [TT05b] David E. Taylor and Jonathan S. Turner. ClassBench: A Packet Classification Benchmark. In *IEEE INFOCOM*, 2005.
- [WAAR06] Dan Wendlandt, Ioannis Avramopoulos, David Andersen, and Jennifer Rexford. Don't Secure Routing Protocols, Secure Data Delivery. In *Proc. 5th ACM Workshop on Hot Topics in Networks (Hotnets-V)*, Irvine, CA, November 2006.

- [WEO04] T. Wan, K. Evangelos, and P. C. Van Oorschot. S-rip: A secure distance vector routing protocol. *Applied Cryptography and Network Security (ANCS 2004)*, 2004.
- [Whi03] R. White. Securing bgp through secure origin bgp. *The Internet Protocol Journal*, (3), 2003.
- [WJ02] J. Winick and S. Jamin. Inet-3.0: Internet topology generator. Technical Report UM-CSE-TR-456-02, University of Michigan, 2002.
- [Woo00] T. Woo. A Modular Approach to Packet Classification: Algorithms and Results. In *IEEE INFOCOM*, 2000.
- [WWV⁺97] S. F. WU, F. Wang, B. M. Vetter, R. Cleaveland, Y. F. Jou, F. Gong, and C. Sargor. Intrusion detection for link-state routing protocols. *IEEE Symposium on Security and Privacy*, May 1997.
- [xor] XORP. <http://www.xorp.org/>.
- [ZCB96] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to Model an Inter-network. In *Proceedings of IEEE INFOCOM*, March 1996.
- [ZGC03] D. Zhu, M. Gritter, and D. R. Cheriton. Feedback based routing. In *ACM SIGCOMM Computer Communication Review*, 2003.
- [ZH99] L. Zhou and ZJ Haas. Securing ad hoc networks. *Network, IEEE*, 13(6):24–30, 1999.
- [ZN05] Hui Zang and Antonio Nucci. Optimal NetFlow Deployment in IP Networks. In *Proc. of International Teletraffic Congress (ITC)*, Aug 2005.
- [ZWG07] Charles Zhang, Marianne Winslett, and Carl Gunter. On the Safety and Efficiency of Firewall Policy Deployment. In *IEEE Symposium on Security and Privacy*, May 2007.